Compilado de Preguntas

Estructuras de Datos y Algoritmos – IIC2133

1 Algoritmos de Ordenación **2**

2 Árboles **73**

3 Hashing **191**

4 Grafos **220**

5 Backtracking **264**

6 Algoritmos Codiciosos **281**

7 Programación Dinámica **339**

1 Algoritmos de Ordenación

En esta sección se encuentran los siguientes contenidos:

- SelectionSort
- InsertionSort
- QuickSort
- MergeSort
- HeapSort

2020-2-I1-P1–QuickSort (1/3)



Pontificia Universidad Católica de Chile
Escuela de Ingeniería
Departamento de Ciencia de la Computación

IIC2133 — Estructuras de Datos y Algoritmos
2020 - 2

Interrogación 1

Pregunta 1

a) Supongamos que al ejecutar Quicksort sobre un arreglo particular, la subrutina partition hace siempre el mayor número posible de intercambios; ¿cuánto tiempo toma Quicksort en este caso? ¿Qué fracción del mayor número posible de intercambios se harían en el mejor caso? Justifica

b) El algoritmo quicker-sort llama a la subrutina pq-partition, que utiliza dos pivotes p y q $(p < q)$ para particionar el arreglo en 5 partes: los elementos menores que p, el pivote p, los elementos entre p y q, el pivote q, y los elementos mayores que q. Escribe el pseudocódigo de pq-partition. ¿Es quicker-sort más eficiente que quick-sort? Justifica.

Solución Pregunta 1a)

La subrutina partition para arreglos que vimos en clases:

```
partition(A, i, f):
    x ← un indice aleatorio en [i, f]
    p ← A[x]
    A[x] ⇄ A[f]
    j ← i
    for k ∈ [i, f − 1]:
        if A[k] < p:
            A[j] ⇄ A[k]
            j ← j + 1
    A[j] ⇄ A[f]
    return j
```

2020-2-I1-P1–QuickSort (2/3)

Podemos ver que hay dos intercambios que siempre se hacen, que son poner el pivote al final del arreglo y luego moverlo a la posición que separa los menores de los mayores.

Los demás intercambios se hacen solo cuando encontramos una clave que es menor al pivote. Por lo tanto, en el caso de que se hagan la mayor cantidad de intercambios es cuando todas las claves del subarreglo son menores al pivote.

Esto significa que si el subarreglo tenía $\boldsymbol{m}$ elementos, se hacen $\boldsymbol{m+1}$ intercambios, y el subarreglo se separa en dos subarreglos disparejos, uno de largo $\mathbf{0}$ y el otro de largo $\boldsymbol{m-1}$.

Esto significa que el tiempo que toma Quicksort para un arreglo inicial de $\boldsymbol{n}$ elementos es:

$$\boldsymbol{T(n) = n + (n-1) + (n-2) + \cdots + 1}$$

$$\boldsymbol{T(n) \in \mathcal{O}(n^2)}$$

El cual es el peor caso de Quicksort.

Por otro lado, sabemos que el mejor caso de Quicksort se produce cuando partition separa el subarreglo en dos mitades de el mismo largo (o lo más parejo posible).

Esto significa que la mitad de los elementos del subarreglo deben ser mayores al pivote y otra mitad menores al pivote. Ya vimos que los intercambios se hacen sólo para los elementos menores al pivote, por lo que en este caso tendríamos **la mitad** de intercambios que en el caso anterior.

[**1pt**] Por identificar que implican los intercambios en partition

[**1pt**] Por identificar correctamente el peor caso

[**1pt**] Por identificar correctamente fracción en el mejor caso

[**Era posible interpretar esta pregunta como *cuál es la fracción de intercambios que se efectúan en Quicksort*, en lugar de cada llamada a partition. Esta interpretación es válida y se evalúa de manera correspondiente. No basta con el resultado final, debe tener los pasos intermedios correctos.**]

2020-2-I1-P1–QuickSort (3/3)

Solución Pregunta 1b)

Ya que la definición dice que $p < q$, entonces es posible asumir que no hay datos repetidos. De todos modos, esto no afecta el análisis.

Lo más simple es implementar partition con listas ligadas como vimos en clases:

```
1:  procedure PQ-PARTITION(lista ligada L)
2:      p, q ← dos nodos de L tal que p < q. Quitar estos nodos de L.
3:      A, B, C ← listas vacias   ▷ Los elementos menores a p, entre p y q y mayores a q respectivamente
4:      for nodo x ∈ L do
5:          if x < p then
6:              agregar x al final de A
7:          else if x < q then
8:              agregar x al final de B
9:          else
10:             agregar x al final de C
11:         end if
12:     end for
13:     return A, p, B, q, C
14: end procedure
```

El paso en la linea 2 es fácil de hacer en $\mathcal{O}(1)$, basta con extraer los primeros dos elementos de L, e intercambiarlos si $p > q$.

Recordemos que el peor caso de Quicksort se produce cuando partition separa una secuencia de m datos en dos secuencias disparejas, de 0 y $m-1$ datos cada una. En ese caso la complejidad para una secuencia inicial de n datos es de:

$$T(n) = n + (n-1) + (n-2) + \cdots + 1$$

$$T(n) \in \mathcal{O}(n^2)$$

En este caso esto también puede suceder, solo que se separa en 3 secuencias disparejas, de 0, 0 y $m-2$ elementos cada una. En este caso la complejidad para una secuencia inicial de n datos es de:

$$T(n) = n + (n-2) + (n-4) + \cdots + 1$$

$$T(n) \in \mathcal{O}(n^2)$$

Ambos algoritmos son iguales en el peor caso, así que no podemos afirmar que quickersort sea mejor que quicksort.

[1pt] Por el pseudocódigo

[1pt] Por calcular la complejidad de quicker-sort

[1pt] Por argumentar cual de los dos es mejor [varía según el argumento]

2020-1-I1-P1–HeapSort (1/3)

IIC2133 – Estructuras de Datos y Algoritmos

Interrogación 1

Hora inicio: 14:00 del 8 de abril del 2020

Hora máxima de entrega: 13:00 del 9 de abril del 2020

0. Formalidad. Responde esta pregunta en papel y lápiz, incluyendo tu firma al final.

    a. ¿Cuál es tu nombre completo?
    b. ¿Te comprometes a no preguntar ni responder dudas de la prueba a nadie que no sea parte del cuerpo docente del curso, ya sea de manera directa o indirecta?

1. A la hora de ordenar usando $HeapSort$, es necesario primero que el arreglo esté estructurado como un heap. Considera la siguiente función que recibe un arreglo A de n elementos:

    $preparar\ para\ HeapSort(A, n)$:

        $for\ i = \left\lfloor \frac{n}{2} \right\rfloor .. 0$:

            $sift\ down(A, i)$

    a. Demuestra que $preparar\ para\ HeapSort$ deja el arreglo A estructurado como un heap.
    b. Justifica, mediante cálculos, que la complejidad de este algoritmo es $O(n)$.

2020-1-I1-P1–HeapSort (2/3)



Pontificia Universidad Católica de Chile
Escuela de Ingeniería
Departamento de Computación
IIC2133 – Estructuras de datos y algoritmos
Primer Semestre 2020

Pauta Interrogación 1

Problema 1

a)

Idea central: El algoritmo comienza desde el ultimo padre escalando hacia arriba del árbol ordenando cada subárbol como un heap (las hojas de estos son trivialmente heaps de un solo elemento). Esta propiedad es utilizada hasta llegar al primer elemento completando lo pedido en el enunciado. **(0.2 puntos)**

Demostración formal:

La presente demostración es parecida a una inducción. Se asume una proposición B. Se expone su caso base, y se muestra que si en un caso general (ciclo n) si la proposición B es verdadera en el siguiente caso (ciclo n+1) también lo será.

Proposición B: Todos los elementos mayores a i son la raíz un heap.

Concepto de termino y correctitud: Al llegar al primer elemento, como sus hijos son heaps el algoritmo dejará la totalidad del array estructurado como uno. **(proposición + concepto de termino y correctitud = 0.2 puntos)**

Caso base: Todos los elementos mayores a $\lfloor \frac{n}{2} \rfloor$ son hojas y trivialmente heaps. **(0,2 puntos)**

Mantenimiento: Como todos los hijos de i tienen una numeración mayor, debido a la proposición B tienen que ser raíces de heaps. Esta es precisamente la condición necesaria para que SiftDown cree un heap. Preservando la proposición B, para el siguiente ciclo: Todos los elementos mayores a $(i-1)$ son la raíz un heap. **(0.4 puntos)**

b)

La idea principal de por qué toma tiempo lineal (i.e. $\mathcal{O}(n)$) se basa en que la cantidad de pasos que toma la operación `siftDown` depende del nivel del árbol en el que estamos situados. El algoritmo toma $\mathcal{O}(log(n))$ cuando estamos en la raíz y $\mathcal{O}(1)$ cuando estamos en la penúltima hoja del árbol.

Sabemos que la cantidad de nodos por nivel está dado por la siguiente fórmula

$$nodos_h = \left\lceil \frac{n}{2^{h+1}} \right\rceil$$

donde h es la altura del árbol ($log(n)$ para la raíz). Además, sabemos que la cantidad de pasos que tomará `siftDown` en el nivel de altura h será, en el peor caso, de h pasos. **(0.3 puntos)**

Teniendo lo anterior, podemos demostrar que la complejidad de nuestro algoritmo está dada por

2020-1-I1-P1–HeapSort (3/3)

$$\sum_{h=0}^{\lfloor log(n) \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil \mathcal{O}(h)$$

Es decir, la cantidad de nodos por nivel, multiplicados por la complejidad de aplicar el método `siftDown` a esos nodos nos dará la complejidad total de nuestro algoritmo. Es importante notar que podemos partir desde $h = 0$ dado que, si bien comenzamos desde la penúltima hoja, en la última hoja la expresión se multiplica por $h = 0$ (la cantidad de pasos de `siftDown` son 0). Resolviendo obtenemos

$$= \mathcal{O}\left(n \cdot \sum_{h=0}^{\lfloor log(n) \rfloor} \frac{h}{2^{h+1}} \right)$$

(0.3 puntos)

$$= \mathcal{O}\left(\frac{n}{2} \cdot \sum_{h=0}^{\lfloor log(n) \rfloor} \frac{h}{2^{h}} \right)$$

Dado que estamos en notación asintótica, podemos escribir la expresión anterior como

$$= \mathcal{O}\left(\frac{n}{2} \cdot \sum_{h=0}^{\infty} \frac{h}{2^{h}} \right)$$

La suma converge a 2, por lo que nos queda

$$= \mathcal{O}\left(\frac{n}{2} \cdot 2 \right) = \mathcal{O}(n)$$

(0.4 puntos)

2020-1-I1-P2–QuickSort (1/3)

2. En ocasiones, como vimos con ***QuickSort***, los algoritmos pueden tener una componente aleatoria. Esto puede ser muy útil, ya que en el caso de ***QuickSort*** significa que no podemos forzar el peor caso intencionalmente. En otros algoritmos, por ejemplo, puede ser un problema. Considera el siguiente algoritmo de ordenación que recibe un conjunto A de n elementos:

   BogoSort(A,n):

   while(***true***):

   Sea B un arreglo vacío

   while A aún tenga elementos:

   Extraer aleatoriamente un elemento de A y ponerlo al final de B

   if B está ordenado:

   return B

   Devolver todos los elementos de B a A

   a. Demuestra que ***BogoSort*** no es correcto según la definición vista en clases. ¿Cuál es la lógica detrás de este algoritmo? ¿Qué habría que modificar en el algoritmo para que este sea correcto, manteniendo esta lógica?

   b. ¿Qué hace que ***QuickSort*** sea correcto, pese a que toma decisiones aleatorias?

2020-1-I1-P2–QuickSort (2/3)

Problema 2

a)

Finitud: Como se extraen elementos de forma aleatoria de A a B, aun que un orden esté mal, esto no asegura que en un intento futuro no se valla a repetir el mismo orden. Por lo tanto no hay manera de demostrar que hay un avance en dirección al final. Por lo tanto el algoritmo falla en este ámbito.**(Max 0.4 puntos)**

(0.2 puntos) Menciona que puede no terminar.
(0 puntos) No menciona que puede no terminar.

(0.2 puntos) Menciona que ordenes incorrectos se pueden repetir.
(0.1 puntos) Menciona probabilidades(correcta o incorrectamente) y/o loops infinitos.
(0 puntos) Sin explicación.

Lógica: La lógica detrás de este algoritmo es generar permutaciones aleatorias de los datos, con la esperanza de obtener un arreglo ordenado. **Max 0.3 puntos**

(0.3 puntos) Menciona "permutaciones" explicitamente, y el probarlas.
(0.2 puntos) No menciona "permutaciones" pero tiene la idea correcta.
(0.1 puntos) Habla de aleatoriedad.
(0 puntos) Si no habla de la lógica del algoritmo.

2020-1-I1-P2–QuickSort (3/3)

Arreglo: La manera de mantener la aleatoriedad pero arreglar el problema de la finitud sería asegurar de alguna forma que una misma permutación, de los elementos del conjunto A, no se genere genere dos veces. En otras palabras, que no se repita la misma permutación más de una vez.**Max 0.3 puntos**

(0.3 puntos) Habla de que se use cada permutación solo una vez. (Sigue lógica de Bogosort)
(0.2 puntos) Usa otras lógicas, pero logra resolver el problema.
(0 puntos) Si no arregla el problema de bogosort.
(0 puntos) Si usa lógicas de ordenación conocidas (Inserionsort, Selectionsort) randomizadas.

Si el alumno le falta puntaje en esta pregunta puede conseguir estos puntos extra, mencionando ambos criterios(0.1 pto.) y explicando por que si cumple su fucnión cuando termina(0.1 pto.):

Criterios: Para que un algoritmo de ordenación sea correcto necesitamos ver que se cumplan dos cosas:

- ◇ Finitud: Termina en una cantidad finita de pasos.
- ◇ Correctitud: Cumple con su función al terminar. Como es un algoritmo de ordenación: Que al retornar B, este esté ordenado. **(0.1 puntos)**

Correctitud: Como sabemos que para retornar B se tiene que cumplir que "B está ordenado", entonces en esta parte el algoritmo revisa si B está ordenado. Si está ordenado, retorna B ordenado, por lo que cumple con su función. Si no está ordenado, no retorna, devuelve los elementos de B a A y vuelve a correr el while. Por lo tanto sabemos que si B es retornado, estará ordenado.**(0.1 puntos)**

b)

La aleatoriedad de Quicksort se encuentra en la elección del pivote para realizar la partición. Una vez que un elemento es elegido como pivote, éste queda en su posición correcta y no puede volver a ser elegido como pivote. Lo anterior implica que en cada iteración el número de elementos a ordenar va disminuyendo, por lo tanto, con un número de iteraciones finitas, el algoritmo termina para cualquier caso, lo que no sucede con Bogosort. **(1 punto)**

2020-1-I1-P3–MergeSort (1/5)

3. $MergeSort$ utiliza la estrategia "dividir para conquistar" dividiendo los datos en 2 y luego resolviendo el problema recursivamente. Considera una variante de $MergeSort$ que divide los datos en 3 y los ordena recursivamente, para luego combinar todo en un arreglo ordenado usando una variante de $Merge$ que recibe 3 listas.

   a. Escribe la recurrencia $T(n)$ del tiempo que toma este nuevo algoritmo para un arreglo de n datos. ¿Cuál es su complejidad, en notación asintótica?

   b. Generaliza esta recurrencia a $T(n, k)$ para la variante de $MergeSort$ que divida los datos en k. ¿Cuál es la complejidad de este algoritmo en función de n y k? Considera que la cantidad de pasos que toma $Merge$ para k listas ordenadas, de n elementos en su totalidad, es $n \cdot \log_2(k)$. Por ejemplo, si $k = 2$, $Merge$ toma n pasos, ya que $\log_2(2) = 1$.

   Finalmente, ¿Qué sucede con la complejidad del algoritmo cuando k tiende a n?

2020-1-I1-P3–MergeSort (2/5)

Problema 3

a)

Sabemos que $Merge$ funciona en $O(n)$, y que *MergeSort* funciona en $O(1)$ para un solo elemento, y que para un input n, esta variable llamará recursivamente a *MergeSort* tres veces, con inputs $\lceil \frac{n}{3} \rceil$, $\lfloor \frac{n}{3} \rfloor$ y $n - \lfloor \frac{n}{3} \rfloor - \lceil \frac{n}{3} \rceil$para después unir las 3 con *Merge*. Por lo tanto, la ecuación de recurrencia quedaría:

$$T(n) = \begin{cases} 1 & \text{if n} = 1 \\ T\Big(\lceil \frac{n}{3} \rceil\Big) + T\Big(\lfloor \frac{n}{3} \rfloor\Big) + T\Big(n - \lceil \frac{n}{3} \rceil - \lfloor \frac{n}{3} \rfloor\Big) + n & \text{if n} > 1 \end{cases}$$

Alternativamente:

$$T(n) \leq \begin{cases} 1 & \text{if n} = 1 \\ 3 * T\Big(\lceil \frac{n}{3} \rceil\Big) + n & \text{if n} > 1 \end{cases}$$

(0.5 pts)
Se descuenta 0.1 si solo indica $\frac{n}{3}$, sin resolver la divisón entera, si indica mal la recursión o si indica la cota superior como igualdad.

2020-1-I1-P3–MergeSort (3/5)

Para la complejidad asintótica tenemos dos opciones, utilizar el teorema maestro, o resolver la recurrencia reemplazando recursivamente.
El teorema maestro resuelve recurrencias de la forma:

$$T(n) = a \cdot T\Big(\frac{n}{b}\Big) + f(n)$$

Donde:

- n es el tamaño del problema.
- a es el número de subproblemas en la recursión.
- $\frac{n}{b}$ el tamaño de cada subproblema.
- $f(n)$ es el costo de dividir el problema y luego volver a unirlo.

En este caso, podemos acotar la recurrencia por arriba, sabiendo que cada subllamada tendrá a lo más $\lceil \frac{n}{3} \rceil$ elementos, por lo que podemos decir que:

$$T(n) \leq 3 * T\Big(\lceil \frac{n}{3} \rceil\Big) + n$$

Aquí tenemos que $a = b = 3$, y $f(n) = n$, y tenemos que $f(n) \in \Theta(n^{log_b a}) = \Theta(n^{log_3 3}) = \Theta(n)$, por lo tanto, según el teorema maestro (caso 2),

$$T(n) \in O\big(n \cdot log(n)\big)$$

(Opción 1: 0.5 pts)

Para resolver esta recurrencia reemplazando recursivamente buscamos un k tal que $n \leq 3^k < 3n$. Se cumple que $T(n) \leq T(3^k)$. Como $\lceil \frac{3^k}{3} \rceil = \lfloor \frac{3^k}{3} \rfloor = 3^k - \lceil \frac{3^k}{3} \rceil - \lfloor \frac{3^k}{3} \rfloor$, podemos entonces, reescribir la recurrencia de la siguiente forma:

$$T(n) \leq T(3^k) = \begin{cases} 1 & \text{if k} = 0 \\ 3^k + 3 \cdot T(3^{k-1}) & \text{if k} > 0 \end{cases}$$

Expandiendo la recursión:

$$\begin{aligned} T(n) \leq T(3^k) &= 3^k + 3 \cdot [3^{(k-1)} + 3 \cdot T(3^{k-2})] && (1) \\ &= 3^k + [3^k + 3^2 \cdot T(3^{k-2})] && (2) \\ &= 3^k + 3^k + 3^2 \cdot [3^{k-2} + 3 \cdot T(3^{k-3}) && (3) \\ &= 3^k + 3^k + 3^k + 3^3 \cdot T(3^{k-3}) && (4) \\ &... && (5) \\ &= i \cdot 3^k + 3^i \cdot T(3^{k-i}) && (6) \end{aligned}$$

cuando $i = k$, por el caso base tenemos que $T(3^{k-i}) = 1$, con lo que nos queda

$$T(n) \leq k \cdot 3^k + 3^k \cdot 1$$

Ahora, tenemos que volver a nuestra variable inicial n. Por construcción de k:

$$3^k < 3n$$

Tenemos entonces que

$$T(n) \leq k \cdot 3^k + 3^k < \log_3(3n) \cdot 3n + 3n$$

Por lo tanto

$$T(n) \in \mathcal{O}\big(n \cdot \log_3(n)\big) = \mathcal{O}\big(n \cdot \log(n)\big)$$

(Opción 2: 0.5 pts)
Se descuenta 0.1 si no utiliza la base del *log* en el reemplazo o el desarrollo, ya que es la diferencia de profundidad con MergeSort normal.

2020-1-I1-P3–MergeSort (4/5)

b)

Para esta pregunta hay mas de una solución ya que no era necesario realizar una demostración formal, igualmente en esta solución se incluye una explicación mas formal.

Primera solución

Una de las soluciones para generalizar la recurrencia de $\mathbf{T(n,k)}$ seria indicar en primer lugar que la función que modela la recurrencia para este caso sería para n > 1

Se divide el arreglo en k arreglos de al menos $\lceil \frac{n}{k} \rceil$ elementos.

$$T(n,k) \leq \underbrace{log_2(k) \cdot n}_{\text{Costo de realizar merge para k arreglos ordenados}} + \overbrace{T\left(\left\lceil \frac{n}{k} \right\rceil, k\right) + T\left(\left\lceil \frac{n}{k} \right\rceil, k\right) + ... + T\left(\left\lceil \frac{n}{k} \right\rceil, k\right)}$$

Y para n= 1

$$T(1,k) = 1$$

Ahora bien, esto es equivalente a decir

$$T(n,k) \leq log_2(k) \cdot n + k \cdot T\left(\left\lceil \frac{n}{k} \right\rceil, k\right)$$

Si se reemplaza n por $n \leq k^y < k \cdot n$ quedara

$$T(n,k) \leq T(k^y,k) = log_2(k) \cdot k^y + k \cdot T\left(k^{y-1}, k\right)$$

Y de manera recursiva quedara

$$T(k^y,k) = log_2(k) \cdot k^y + k \cdot (log_2(k) \cdot k^{y-1} + k \cdot T\left(k^{y-2}, k\right))$$

Quedando finalmente

$$T(k^y,k) = log_2(k) \cdot k^y + k \cdot (log_2(k) \cdot k^{y-1} + k \cdot (log_2(k) \cdot k^{y-2} + ... + (k^{y-y} \cdot log_2(k) + k \cdot T\left(1,k\right))))$$

Que en otras palabras es

$$T(k^y,k) = y \cdot k^y \cdot log_2(k) + k^y$$

Y por la condicion que se establecio en la definición de k^y, notar que

$$k^y < k \cdot n / log_k$$

$$y < log_k(k \cdot n) = \frac{log(kn)}{log(k)}$$

Por tanto quedara

$$T(n,k) \leq T(k^y) < \left(\frac{log_2(n)}{log_2(k)} + 1\right) \cdot n \cdot k \cdot log_2(k) + n \cdot k$$

Reordenando

$$T(n,k) < log_2(n) \cdot n \cdot k + n \cdot k \cdot (log_2(k) + 1)$$

A partir de esto se puede concluir que

2020-1-I1-P3–MergeSort (5/5)

$$\therefore T(n,k) \in O(k \cdot n \cdot log(n))$$

Tambien es valido haber indicado como posible orden

$$T(n,k) \in O(k \cdot n \cdot log(n \cdot k))$$

- **0.75 pts** por explicación y/o mostrar de manera correcta el orden de complejidad
- **0.6 pts** Por explicación y/o mostración correcta pero orden de complejidad incorrecto.
- **0.3 pts** Por explicación y/o mostración con errores mayores
- **0 pts** Por explicación y/o mostración incorrecta

Segunda solución

Se explica a traves de un desarrollo correcto que el orden de complejidad es $O(n * log(n))$
Como por ejemplo

$$\sum_{i=0}^{log_k(n)} log_2(k) \cdot \frac{n}{k^i} + k^{i+1} T(\frac{n}{k^{i+1}}, k)$$

$$\frac{log(k)}{log(2)} n \frac{log(n)}{log(k)} + k^{log_k(n)+1}$$

$$n * log_2(n) + n * k$$

.

- **0.75 pts** por explicación y/o mostrar de manera correcta el orden de complejidad
- **0.6 pts** Por explicación y/o mostración correcta pero orden de complejidad incorrecto.
- **0.3 pts** Por explicación y/o mostración con errores mayores
- **0 pts** Por explicación y/o mostración incorrecta

Para el caso de la complejidad del algoritmo para el caso que k tienda a n, es claro que la complejidad tendera a converger a $O(n \cdot log(n))$. Es claro si se reemplaza en la ecuación de recursión T(n,n).

- **0.25 pts** Por explicación correcta.
- **0.1 pts** Por explicación con intuición correcta pero equivocada respecto al orden de complejidad.
- **0 pts** Por explicación incorrecta.

2020-1-Ex-P1–Justificación, complejidad de algoritmo de ordenación (1/3)

IIC2133 – Estructuras de Datos y Algoritmos

Examen

Hora inicio: 9:00 del 8 de julio del 2020

Hora máxima de entrega: 15:00 del 8 de julio del 2020

0. Responde esta pregunta en papel y lápiz, incluyendo tu firma al final. Nos reservamos el derecho a no corregir tu prueba según tu respuesta a esta pregunta. Puedes adjuntar esta pregunta a cualquiera de las preguntas que subas a SIDING.

    a. ¿Cuál es tu nombre completo?
    b. ¿Te comprometes a no preguntar ni responder dudas del examen a nadie que no sea parte del cuerpo docente del curso, ya sea de manera directa o indirecta?

Responde sólo 3 de las 4 preguntas a continuación. Si respondes 4, se escogerá arbitrariamente cuales 3 corregir, y la otra se considerará como no entregada.

1. Se tiene el siguiente algoritmo de ordenación para un arreglo $\boldsymbol{A}$ de $\boldsymbol{n}$ pares $(\boldsymbol{key}, \boldsymbol{value})$, donde las $\boldsymbol{key}$ son números **naturales**:

```
IndexSort(A, n):
    min ← el mínimo elemento entre las claves de A
    max ← el máximo elemento entre las claves de A
    range ← max − min + 1
    INDEX ← un arreglo de largo range lleno con ceros
    for a in A:
        Incrementar INDEX[a.key − min] en 1
    for x in 1 ... range − 1:
        INDEX[x] ← INDEX[x] + INDEX[x − 1]
    S ← un arreglo de largo n para guardar el arreglo ordenado
    for i in n − 1 ... 0:
        Disminuir INDEX[A[i].key − min] en 1
        index ← INDEX[A[i].key − min]
        S[index] ← A[i]
    return S
```

a) Justifica por qué *IndexSort* es correcto. No es necesario hacer una demostración formal.
b) Calcula su complejidad. Considera que encontrar el mínimo y el máximo toma $\boldsymbol{O(n)}$.
c) Identifica al menos 2 diferencias fundamentales que tiene este algoritmo con los algoritmos vistos en clases.

2020-1-Ex-P1–Justificación, complejidad de algoritmo de ordenación (2/3)

IIC2133 - Pauta Examen 2020-1

Problema 1

Parte A

Podemos ver en el primer for que lo que hace el algoritmo es contar, en $INDEX[key - min]$, la cantidad de tuplas que tienen key como clave.

Luego, en el segundo for, va sumando hacia adelante estos contadores, de manera que $INDEX[key - min]$ tiene la cantidad de tuplas con clave menor o igual a key.

Por ultimo, el último for recorre el arreglo de atrás para adelante, y para cada elemento de clave key, mira cuantos elementos tienen clave menor o igual a él y lo pone en esa posición en el arreglo solución S, y disminuye ese contador en uno para poder posicionar correctamente el siguiente elemento con esa clave. Eso si esta disminución se hace antes de posicionar el elemento ya que los arreglos parten en 0.

Por lo tanto, si tenemos k elementos con claves menor a key, y m elementos de clave key, y estos van a quedar entre las posiciones k y $k + m$. Por lo tanto el arreglo S está ordenado.

[1.5 pts] Por identificar lo que hace el algoritmo en cada paso.
[0.5 pts] Por justificar por qué eso hace que ordene.

La pregunta pedía justificar. Si demuestran también se considera correcto.

Parte B

En primer lugar, encontrar el mínimo y el máximo elemento toma $O(n)$.

Luego recorremos A, que tiene n elementos. Como cada operación del for toma $O(1)$ ya que es acceso a un arreglo, este for aporta $O(n)$ a la complejidad.

El siguiente for recorre desde 1 hasta range - 1, y cada operación nuevamente es $O(1)$, por lo que ese for aporta $O(range)$.

El último for recorre A. Nuevamente, todas las operaciones adentro son $O(1)$, por lo que este for aporta $O(n)$ a la complejidad.

Por lo tanto, la complejidad total del algoritmo es

$$O(n) + O(n) + O(range) + O(n)$$

$$O(n + range)$$

El término $range$ no es una constante ya que depende de los datos, por lo que no lo podemos simplificar.

[1.5pt] Por calcular correctamente la complejidad.
[0.5pt] Por NO simplificar el $range$

2020-1-Ex-P1–Justificación, complejidad de algoritmo de ordenación (3/3)

Parte C

El puntaje se recibe por identificar diferencias específicas de la siguiente lista. Se eligen las dos mejores.

A diferencia de los algoritmos vistos en clases:

- ◇ **[1pt]** Este algoritmo no compara los elementos: todos los algoritmos que vimos ordenaban mediante comparación de elementos (se puede ver incluso que el algoritmo no tiene ningun if)
- ◇ **[1pt]** La complejidad de este algoritmo depende de sus datos, además del n. Los algoritmos que vimos en clases podían depender del orden en que venían los datos (QuickSort, InsertionSort), pero jamás de qué números se estaban ordenando.
- ◇ **[1pt]** Este algoritmo es lineal, asumiendo que el *range* no depende de n. Los algoritmos más rapidos que estudiamos en clases ordenaban en tiempo $O(nlog(n))$.
- ◇ **[0.5pt]** Este algoritmo solo funciona si las claves son números enteros positivos. Esto recibe menos puntaje ya que es algo que viene en el enunciado.
- ◇ **[0.5pt]** (Si analizaron el algoritmo como estaba originalmente) Este algoritmo ordena los datos de mayor a menor. Recibe menos puntaje ya que es una diferencia trivial y es fácilmente solucionable en $O(n)$.

2019-2-I1-P2–InsertionSort, SelectionSort, QuickSort (1/1)

2. Tienes a tu disposición 4 ejecutables compilados E_1, E_2, E_3, E_4 correspondientes a 4 algoritmos de ordenación: InsertionSort, SelectionSort, QuickSort con pivote aleatorio, QuickSort con pivote del primer elemento. El problema es que no sabes cual archivo corresponde a que algoritmo. Para descubrir a qué algoritmo corresponde cada archivo creas 3 archivos con 1 millón de números para ordenar: El primer archivo tiene los números ya ordenados, el segundo tiene los números de manera aleatoria y el tercero tiene los números ordenados en el orden inverso. Al ejecutar cada programa con cada set de números obtienes la siguiente tabla de tiempos:

Archivo	Datos ordenados	Datos aleatorios	Datos al revés
E_1	44.88s	48.72s	43.35s
E_2	40.87s	9.97s	43.22s
E_3	7.35s	25.47s	46.11s
E_4	10.05s	9.81s	10.56s

[6pt] Diga cuál archivo corresponde a qué algoritmo y justifique cómo llegó a esa conclusión a partir de los datos obtenidos.

Solución: De los algoritmos dados sabemos sus mejores y peores casos con sus complejidades respectivas. En particular sabemos que:

i. InsertionSort toma tiempo lineal cuando los datos están ordenados, toma tiempo cuadrático cuando los datos están en el orden inverso y toma tiempo cuadrático en el caso promedio. Además, sabemos que el caso en el que realiza más intercambios es cuando el arreglo está al revés ya que es el caso en el que todos los pares están invertidos.

ii. SelectionSort toma tiempo cuadrático en todos los casos ya que siempre revisa todos los datos del arreglo en cada iteración.

iii. QuickSort aleatorio: El caso promedio toma tiempo $O(n \cdot \log n)$ y el peor caso tiempo cuadrático. Ya que el pivote es aleatorio lo más probable es que el tiempo en los 3 inputs sea similar y que el número de comparaciones sea proporcional a $n \cdot \log n$.

iv. QuickSort con pivote primer elemento: El peor caso es cuadrático y se da cuando el escoge como pivote el mayor o el menor elemento del intervalo. En el caso de que el arreglo esté ordenado normalmente o al revés se dará el peor caso. En el caso aleatorio se espera que el tiempo sea proporcional a $n \cdot \log n$.

Por lo tanto, $E_2 =$ QuickSort con pivote primer elemento ya que toma mucho más tiempo cuando el arreglo está completamente ordenado y cuando está completamente ordenado en el orden contrario. $E_1 =$ SelectionSort ya que siempre toma mucho tiempo ordenar con el algoritmo independiente del orden. $E_3 =$ InsertionSort ya que hay una clara tendencia a que mientras más ordenado, menos tiempo toma. $E_4 =$ QuickSort con pivote aleatorio ya que en todos los casos tuvo un comportamiento eficiente independiente del orden, lo cual es lo esperable.

Distribución de puntajes:

1.5 puntos por identificar correctamente todos los algoritmos.

1.5 puntos por cada justificación (solo son necesarias 3 ya que se puede dejar uno por descarte)

2019-1-C1–QuickSort modificado (1/2)

Estructuras de Datos y Algoritmos – iic2133

Control 1

20 de marzo, 2019

Nombre: _____

1) Escribe el algoritmo *quicksort3*, que, en lugar de particionar el arreglo A en dos, como lo hace *quicksort*, lo particiona en tres: datos menores que el pivote, datos iguales al pivote, y datos mayores que el pivote. Puedes suponer que las particiones van a parar a listas diferentes o bien al mismo arreglo —especifica. Usa una notación similar a la usada en las diapositivas.

Solución: Pueden haber muchos algoritmos correctos. Las distribución de puntaje se hizo de la siguiente forma:

- [1 pto.] Que se haya hecho en pseudocódigo y sin funciones específicas de ningún lenguaje, es decir, usando la notación que se usa en las diapositivas.
- [3 ptos.] Haya separación correcta entre los menores, iguales y mayores y que se vaya reordenando la lista según corresponda.
- [1 pto.] La nueva implementación de partition retorne dos pivotes.
- [1 pto.] Que el algoritmo termine de forma correcta.

2) Considera el siguiente algoritmo de ordenación, para ordenar el arreglo A de largo n :

$\boldsymbol{sort}(A, n)$:
    $\boldsymbol{for}\ g \in \{5,\ 3,\ 1\}$:
        $\boldsymbol{for}\ i \in [g, n[$:
            $j \leftarrow i$
            $\boldsymbol{while}\ (j \geq g\ \wedge\ A[j] < A[j-g])$:
                $A[j] \rightleftarrows A[j-g]$
                $j \leftarrow j - g$

Demuestra que este algoritmo es correcto según los **dos** criterios vistos en clases.

Solución: El algoritmo es finito y es correcto.

- Es finito:
  [0.5 pto.] El primer "for" termina ya que recorre un conjunto finito

  [0.5 pto.] El segundo "for" termina ya que recorre un conjunto finito

  [2 pto.] La segunda condición no podemos predecirla. Por otro lado, como J es monótonamente decreciente y G es finita, tenemos que el "while" siempre va a terminar ya que la primera condición se va a romper (J >= G).

- Es correcto:
  - ❑ Opción 1:

  [3 pto.] Con G = 1, el algoritmo es igual a Insertion Sort. Como sabemos que Insertion Sort es correcto y siempre se va a ejecutar el algoritmo con G = 1, el algoritmo Sort() es correcto.

2019-1-C1–QuickSort modificado (2/2)

- ❑ Opción 2:

Solo considerando con G = 1. Los otros valores de G no aportan a la demostración.

Invariante: Luego de la iteración i, el arreglo está ordenado hasta el índice i.

Lo demostramos por Inducción.

[1 pto.] Caso Base. i = 1. El primer elemento del arreglo.

Un arreglo de largo uno está siempre ordenado.

[0.5 pto.] Hipótesis Inductiva. Tras la iteración i, A está ordenado hasta el índice i.

[1.5 pto.] En la iteración i+1 existen dos casos:

- a_i <= a_{i+1} -> A está ordenado hasta el índice i+1
- a_i > a_{i+1} ->
  Llamemos a_j= a_{i+1}

  Como ya estaba ordenado hasta a_i, se tiene

  a_1 <= a_2 <= … <= a_{i-1} <= a_i > a_j

  en cada paso el elemento a_j se cambia de posición con el anterior, dejando ordenado a ambos lados:

  a_1 <= a_2 <= … > a_j <= … <= a_{i-1} <= a_i -> while continúa.

  a_1 <= a_2 <= … <= a_j <= … <= a_{i-1} <= a_i -> while termina, y los elementos están ordenados hasta el índice i+1.

  Por inducción, después de la iteración n el arreglo está ordenado hasta el índice n, por lo tanto, está completamente ordenado.

2019-1-Ex-P2–Comparación de algoritmos, InsertionSort, QuickSort, HeapSort (1/2)

2. Supongamos que tienes tres códigos compilados (es decir, códigos producidos por el compilador de C) S_1, S_2 y S_3, correspondientes a tres algoritmos de ordenación: *insertion sort*, *quick sort*, y *heap sort*. **Solo que no sabemos cuál código corresponde a cuál algoritmo**. Explica cómo identificar experimentalmente cuál código corresponde a cuál algoritmo, justificando tus decisiones en base a las propiedades de los algoritmos.

[Durante el examen se dijo que QuickSort escoge como pivote siempre al primer elemento de la secuencia, y que para hacer sus experimentos pueden ejecutar el programa con un input dado y medir cuánto tiempo toma]

Solución:

Para identificar los algoritmos experimentalmente podemos darles inputs específicos y medir cuánto demoran en resolverlos para determinar empíricamente su complejidad.

Las complejidades que buscamos son:

	Peor Caso	Caso Promedio	Mejor Caso
Insertion Sort	$O(n^2)$	$O(n^2)$	O(n)
Quick Sort	$O(n^2)$	O(n log n)	O(n log n)
Heap Sort	O(n log n)	O(n log n)	O(n log n)

[2pt por mencionar las complejidades. El caso promedio no es necesario]

El mejor caso de insertion sort se da cuando los datos vienen en orden. **[1pt]**

El peor caso de quick sort (con el primer elemento como pivote) también es cuando los datos vienen en orden. **[1pt]**

Heap sort demora lo mismo en todos los escenarios: es insensible al orden del input.

Complejidad con los datos ordenados:

Insertion Sort	O(n)
Heap Sort	O(n log n)
Quick Sort	$O(n^2)$

Una Posible estrategia:

2019-1-Ex-P2–Comparación de algoritmos, InsertionSort, QuickSort, HeapSort (2/2)

Hay que medir cuánto demora cada algoritmo en ordenar un set de *n* datos **ordenados**, probando con distintos *n* para poder ver graficar los tiempos para cada *n*, trazando las curvas correspondientes. Es decir, si graficamos los 3 algoritmos juntos, la curva superior corresponde a QuickSort, la de al medio a Heap Sort, y la de abajo a Insertion Sort. **[2pts]**

Otra Posible estrategia:

Es posible identificar los tiempos de cada algoritmo comparándolos consigo mismos. En este caso basta con identificar dos y el tercero sale por descarte.

Para todo el procedimiento es similar al anterior, pero para cada *n* hay que probar con datos ordenados y aleatorios. Nuevamente graficando los resultados, pero esta vez tenemos un gráfico para cada algoritmo, y las curvas para cada gráfico son las de datos ordenados y datos aleatorios. **[1.5pt por razonamiento]**

- Quicksort es el que se demora más con los ordenados que con los aleatorios.
- Insertionsort es el que se demora más con los aleatorios que con los ordenados.
- Heapsort es el que se demora lo mismo con los aleatorios y los ordenados.

[0.5 pts por identificarlos]

2018-2-I1-P3–InsertionSort, cálculo de complejidad (1/2)

3. Ordenación

Una observación que hicimos en clase sobre los algoritmos de ***ordenación por comparación de elementos adyacentes***, p.ej., *insertionSort*( ), es que su debilidad (en términos del número de operaciones que ejecutan) radica justamente en que sólo comparan e intercambian elementos adyacentes. Así, si tuviéramos un algoritmo que usara la misma estrategia de *insertionSort*( ), pero que comparara elementos que están a una distancia > 1 entre ellos, entonces podríamos esperar un mejor desempeño.

a) Calcula cuántas comparaciones entre elementos hace *insertionSort*( ) para ordenar el siguiente arreglo **a** de menor a mayor; muestra que entiendes cómo funciona *insertionSort*( ): **a** = **[**11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1**]**.

insertionSort( ) coloca el segundo elemento ordenado con respecto al primero, luego el tercero ordenado con respecto a los dos primeros (ya ordenados entre ellos), luego el cuarto ordenado con respecto a los tres primeros (ya ordenados entre ellos), etc. En el caso del arreglo **a**, *insertionSort*( ) básicamente va moviendo cada elemento, 10, 9, …, 1, hasta la primera posición del arreglo. Para ello, el 10 es comparado una vez (con el 11), el 9 es comparado dos veces (con el 11 y con el 10), el 8 es comparado tres veces (con el 11, el 10 y el 9), y así sucesivamente; finalmente, el 1 es comparado 10 veces. Luego el total de comparaciones es 1 + 2 + 3 + ... + 10 = **55**.

2018-2-I1-P3–InsertionSort, cálculo de complejidad (2/2)

b) Calcula ahora cuántas comparaciones entre elementos hace el siguiente algoritmo *shellSort*( ) para ordenar el mismo arreglo **a**. Muestra que entiendes cómo funciona *shellSort*( ); en particular, ¿qué relación tiene con *insertionSort*( )?

```
shellSort(a):
   gaps[] = {5,3,1}
   t = 0
   while t < 3:
      gap = gaps[t]
      j = gap
      while j < a.length]:
         tmp = a[j]
         k = j
         while k >= gap and tmp < a[k-gap]:
            a[k] = a[k-gap]
            k = k-gap
         a[k] = tmp
         j = j+1
   t = t+1
```

Notemos que las comparaciones entre elementos de ***a*** *se dan sólo en la comparación* ***tmp < a[k-gap]****; el algoritmo realiza* ***11*** *de estas comparaciones con resultado* ***true*** *y otras* ***17*** *con resultado* ***false****; en total,* ***28****.*

Primero, realiza insertionSort entre elementos que están a distancia 5 entre ellos (según las posiciones que ocupan en ***a****, no en cuanto a sus valores): el 6 con respecto al 11, el 5 c/r al 10, el 4 c/r al 9, el 3 c/r al 8, el 2 c/r al 7, el 1 c/r al 11, y el 1 c/r al 6.*

Luego, realiza insertionSort entre elementos que están a distancia 3 (nuevamente, según sus posiciones en el arreglo): el 2 c/r al 5 y el 7 c/r al 10.

Finalmente, realiza insertionSort entre elementos que están a distancia 1: el 3 c/r al 4 y el 8 c/r al 9; estos son los dos únicos pares de valores que aún están "desordenados" al finalizar el paso anterior.

c) **[**Independiente de a) y b)**]** Tenemos una lista de N números enteros positivos, ceros y negativos. Queremos determinar cuántos tríos de números suman 0. Da una forma de resolver este problema con complejidad mejor que $O(N^3)$.

Se puede hacer en tiempo $O(n^2 \log n)$: Primero, ordenamos la lista de menor a mayor, en tiempo $O(n \log n)$. Luego, para cada par de números, sumamos los dos números y buscamos en la lista ya ordenada, empleando búsqueda binaria, un número que sea el negativo de la suma; si lo encontramos, entonces incrementamos el contador de los tríos que suman 0. Hay $O(n^2)$ pares (los podemos generar sistemáticamente con dos loops, uno anidado en el otro) y cada búsqueda binaria se puede hacer en tiempo $O(\log n)$.

2018-2-I1-P4–MergeSort (1/1)

4. *Merge* y *heaps*

En clase estudiamos el procedimiento *merge*, que recibe como input dos secuencias ordenadas de datos, S_1 y S_2, y produce como output una nueva secuencia ordenada con todos los datos de S_1 y S_2. Vimos que si la cantidad total de datos es n, entonces la complejidad de *merge* es O(n).

a) ¿Cómo funcionaría un procedimiento *merge-3*, que en lugar de recibir dos secuencias ordenadas, recibe tres secuencias ordenadas, S_1, S_2 y S_3, e igualmente produce como output una nueva secuencia ordenada con todos los datos de S_1, S_2 y S_3 ? Calcula cuál es la complejidad de *merge-3*, si la cantidad total de datos es n.

Al igual que al hacer merge con 2 secuencias, simplemente tomo el elemento menor de las secuencias y lo agrego a la secuencia final. Elegir el elemento menor toma a lo más 2 comparaciones, por lo que en total el algoritmo toma O(n) pasos.

b) Explica cuidadodamente cómo funcionaría un procedimiento *merge-k*, que recibe k secuencias ordenadas y produce una nueva secuencia ordenada con todos los datos de las k secuencias de input. Calcula cuál es la complejidad de *merge-k*, si la cantidad total de datos es n, en términos de n y k.

Esta vez tomar el menor elemento de las secuencias no es tan fácil. Para hacerlo utilizo un min heap de tamaño k en el cual agrego las cabezas de las secuencias. Hasta que no me queden elementos voy sacando la cabeza del heap y los remplazo por el siguiente elemento de la secuencia correspondiente. Esto toma tiempo O(nlogk).

Esto también se puede resolver en el mismo tiempo haciendo merge-2 entre pares de secuencias. Luego a las secuencias restantes se les aplica la misma operación hasta que solo queda una secuencia.

2018-2-I3-P1–QuickSort (1/1)

Estructuras de Datos y Algoritmos – IIC2133
I3

1. Quicksort

Considera la siguiente descripción recursiva de *quicksort* para ordenar un arreglo S **sin elementos repetidos**:

1. Si el número de elementos en S es 0 o 1, entonces *return*.
2. Elige cualquier elemento v en S; a este elemento le llamamos el pivote.
3. Particiona $S - \{v\}$ en dos grupos disjuntos: $S_1 = \{ x \in S - \{v\} \mid x < v \}$ y $S_2 = \{ x \in S - \{v\} \mid x > v \}$.
4. *return* {quicksort(S_1) seguido de v seguido de *quicksort*(S_2)}.

a) Si S tiene n elementos, ¿cuál es la complejidad de la partición —el paso 3? Justifica.

Respuesta: [2 ptos.] La complejidad es O(n), ya que se deben comparar los n - 1 elementos con el v y cada comparación es O(1).

b) Considera la estrategia de elegir como pivote siempre el primer elemento de S. Demuestra —es decir, argumenta rigurosamente— que esta estrategia puede producir que *quicksort* tenga un desempeño cuadrático con respecto al número, n, de elementos de S.

Respuesta: [2 ptos.]

Dada un lista ordenada S de largo n y v su primer elemento.

Se particiona $S - \{v\}$ en los grupo: $S_1 = \{ x \in S - \{v\} \mid x < v \}$ y $S_2 = \{ x \in S - \{v\} \mid x > v \}$ y como sabemos que $\forall x \in S\ (v < x)$, entonces $|S_1| = 0\ y\ |S_2| = n - 1$.

Como se sabe de a), cada partición es $O(n)$, entonces en cada iteración se tendrán que hacer la siguiente cantidad de iteraciones: $n + (n - 1) + \ldots + 1 + 0 = \sum_{i=0}^{n} (n - i) = \frac{n(n-1)}{3} = O(n^2)$.

Este corresponde al peor caso de quicksort.

c) Explica cómo modificar *quicksort* para convertirlo en un algoritmo para encontrar el k-ésimo elemento más pequeño de S. Este algoritmo debe tomar tiempo O(n) en el caso promedio.

Respuesta: [2 ptos.]

1. Si el número de elementos en S es 0 o 1, entonces *return*.
2. Elige cualquier elemento v en S; a este elemento le llamamos el pivote.
3. Particiona $S - \{v\}$ en dos grupos disjuntos: $S_1 = \{ x \in S - \{v\} \mid x < v \}$ y $S_2 = \{ x \in S - \{v\} \mid x > v \}$.
4. Si la posición de v es igual a k, entonces *return* v.
5. Si es menor a k, entonces $S = S_2$ y $k = k - |S_1|$ y volver a 2.
6. Si es mayor a k, entonces $S = S_1$ y volver a 2.

También estaban correctos otros cambias, pero que fueran O(n).

2018-2-Ex-P2–Análisis Algoritmo CountSort (1/1)

2. Ordenación

Queremos ordenar n datos, que vienen en un arreglo a, en que cada dato es un número entero en el rango 0 a k. Considera el siguiente algoritmo:

```
countSort(a, n, k):
    sea count[0 … k] un arreglo de enteros
    for i = 0 … k: count[i] = 0
    for j = 0 … n–1: count[a[j]] = count[a[j]] + 1
    for i = 0 … k:
        if count[i] > 0:
            for p = 1 … count[i]: print(i)
```

a) ¿Cuál es el *output* del algoritmo?

Una lista con los n números enteros de a, pero ordenados de menor a mayor.

b) ¿Cuál es la complejidad del algoritmo en función de n y k?

El primer **for** i es O(k); el **for** j es O(n); y el siguiente **for** i es O(n), ya que esencialmente imprime los n datos de a, una vez cada uno. **Luego, el algoritmo es O($k + n$)**.

c) Si k es O(n), ¿cuál es la complejidad del algoritmo?

O(n).

d) Esto pareciera contradecir la demostración que vimos en clase de que los algoritmos de ordenación por comparación tienen complejidad O(n log$n$). Explica por qué no hay una contradicción.

Efectivamente, es un algoritmo de ordenación, **pero no ordena por comparación**: en ninguna parte del algoritmo aparece una condición tal como ***a***[i] < ***a***[j]. (O tal vez pdría argumentarse que la asignación *count*[*a*[j]] = *count*[*a*[j]] + 1 hace una comparación de un valor contra otros n–1 en tiempo O(1).)

2018-1-I1-P3–InsertionSort (1/1)

3. Ordenación por comparación de elementos adyacentes

Sea a un arreglo de números distintos. Si $j < k$ y $a[j] > a[k]$, entonces el par (j, k) se llama una *inversión* de a.

a) ¿Cuál es exactamente el número promedio de inversiones que puede tener un arreglo a de n números distintos? (Hint: Considera el arreglo a y el arreglo a totalmente invertido, que llamamos a'; entonces el par (j, k) es una inversión de a o es una inversión de a'.) Justifica.

Respuesta

Dado cualquier arreglo a y su inverso a' el pal de elementos (i, j) es una inversión en alguno de los dos arreglos. En total hay n*(n-1)/2 pares de elementos, por lo que en promedio hay n*(n-1)/4 inverisones.

b) A partir de (a), justifica que cualquier algoritmo de ordenación que ordena intercambiando elementos adyacentes —por ejemplo, *insertionSort*— requiere tiempo $\Omega(n^2)$ en promedio para ordenar n elementos.

Respuesta

Al intercambiar dos elementos adyacentes, sólo resolvemos una inversión a la vez. Como en promedio hay $n(n\text{-}1)/4$ inversiones, el algoritmo necesita realizar $n(n\text{-}1)/4 = \Omega(n^2)$ intercambios en promedio.

c) ¿Cuál es la relación entre el tiempo de ejecución de *insertionSort* y el número de inversiones del arreglo de entrada? Justifica.

Respuesta

Dado un arreglo a de entrada de n elementos, siempre debe iterar sobre el arreglo completo, por lo que como mínimo toma tiempo $\Omega(n)$. Luego, por cada inversión insertion sort hace un intercambio. Por lo tanto, el tiempo de insertion sort es O(n + inversiones), lo que en el mejor caso es O(n) y en peor caso es $O(n^2)$.

2018-1-I1-P4–MergeSort, HeapSort (1/2)

4. mergeSort y quickSort

a) Siguiendo con las inversiones, definidas en la pregunta 3, describe un algoritmo, basado en *mergeSort*, que determine el número de inversiones de un arreglo de n números distintos en tiempo proporcional a $n\log n$. Explica la complejidad de tu algoritmo.

Respuesta

El número de inversiones de un arreglo es el número de inversiones que hay en su mitad izquierda, más el número de inversiones que hay en su mitad derecha, y más el número de inversiones que hay entre los elementos de la mitad izquierda con respecto a los elementos de la mitad derecha. Sea **a** el arreglo, y **e** y **w** los índices extremos de la parte de **a** en que vamos a calcular el número de inversiones:

```
invCount(int[] a, int e, int w):
     if (e < w):
          int m = (e+w)/2
          return invCount(a, e, m) + invCount(a, m+1, w) + countInvs(a, e, m, w)
     else return 0

int countInvs(int[] a, int e, int m, int w):
     int[] b = new int[a.length]
     int j = e, k = m+1, p = 0, count = 0
     while (j <= m && k <= w):
          if (a[j] > a[k]):
               b[p] = a[k]; k++; count = count + (m-j+1)
          else:
               b[p] = a[j]; j++
          p++
     while (j <= m): b[p] = a[j]; j++
     while (k <= w): b[p] = a[k]; k++
     p = 0;
     for (j = e; j <= w; j++): a[j] = b[p]; p++
     return count
```

b) Como vimos en clase, el desempeño de *quickSort* depende fuertemente de cómo resultan las particiones, lo que a su vez depende de cuál elemento del (sub)arreglo se elige como pivote.

- Explica cuál es la relación entre el resultado de las particiones y el desempeño de *quickSort*.
- Explica los pro y los contra de elegir como pivote el elemento de más a la derecha del (sub)arreglo, como es el caso de la versión del algoritmo que vimos en clase.
- Otro método para elegir el pivote es el llamado “la mediana de 3”: se elige como pivote la mediana entre el elemento de más a la izquierda del (sub)arreglo, el elemento al medio del (sub)arreglo y el elemento de más a la derecha del (sub)arreglo. ¿Qué ventajas presenta este método frente al anterior?

Respuesta

Particiones vs desempeño: Es importante la forma en que se obtiene la partición en cada llamado a _quicksort_ pues si esta es muy desbalanceada, habrá un impacto sobre la complejidad del algoritmo completo. En efecto, lo ideal es que las partes obtenidas luego de particionar un subarreglo sean de tamaños parecidos para que se realicen O(nlog(n)) llamados en total. Para lograr este objetivo, no

2018-1-I1-P4–MergeSort, HeapSort (2/2)

basta fijar una posición del pivote para particionar, sino que además se debe considerar si el arreglo ya está ordenado/semi-ordenado antes de comenzar.

Pivote extremo-derecho: Si se toma siempre el elemento extremo-derecho de cada sub-arreglo el algoritmo toma O(n^2) cuando el arreglo original está casi ordenado (respectivamente, ordenado de mayor a menor). Esto ocurre debido a que al armar las particiones, muy pocos elementos serán mayores (resp. menores) que el pivote y por lo tanto, el llamado recursivo siguiente se hará para un sub-arreglo de tamaño casi igual al actual. El peor caso es cuando el arreglo ya está ordenado, en cuyo caso cada llamado se hará para un sub-arreglo con un elemento menos que el paso anterior.

Mediana de 3: La ventaja inmediata es que este método no cae en los peores casos de usar como pivote el extremo-derecho o el extremo-izquierdo. Si el arreglo está ordenado/ordenado de mayor a menor, como el elemento central es la mediana de los tres elementos escogidos, no se hará un llamado con un sub-arreglo de largo similar al actual. En cualquier otro caso, el método no presenta mayor ventaja que usar un pivote aleatorio.

2018-1-Ex-P6–QuickSelect (1/1)

6. Ordenación y estadísticas de orden

Se quiere hacer un estudio de salud sobre la población chilena. Para esto, se registraron varias métricas sobre N personas elegidas aleatoriamente; una de estas métricas es la estatura de las personas. Para eliminar valores muy extremos (*outliers*), se decidió considerar solo las estaturas desde la i-ésima persona más alta hasta la j-ésima persona más alta; lamentablemente, los valores de las estaturas están desordenados.

Dados un arreglo *datos* con las N estaturas y los valores i y j, escribe un algoritmo en pseudocódigo que tenga tiempo esperado O(N) y que retorne las estaturas en el rango [i, j] (no importa si las retorna desordenadas).

P.ej., si *datos* = [1.80, 1.65, 1.79, 1.56, 1.57, 1.70, 1.76, 1.66, 1.86, 1.92], i = 3, j = 7, entonces el output del algoritmo debe ser 1.65, 1.79, 1.70, 1.76, 1.66.

Respuesta:

Si utilizamos QuickSelect para encontrar el elemento en la posición i del arreglo vamos a tener todas las alturas mayores a la derecha y las menores a la izquierda. Dado esto podemos usar QuickSelect en el sub-arreglo datos[i+1:] y encontrar la posición j. Esto a su vez hace que los elementos a su izquierda sean menores a datos[j]. Por lo tanto, los elementos entre las posiciones i y j son los elementos buscados **[4ptos]**. La complejidad esperada del algoritmo es O(N) ya que QuickSelect toma tiempo esperado O(N) y estamos usando ese algoritmo 2 veces **[2ptos]**.

2017-1-I1-P1–QuickSort, MergeSort, HeapSort (1/2)

PONTIFICIA UNIVERSIDAD CATÓLICA DE CHILE
ESCUELA DE INGENIERÍA
DEPARTAMENTO DE CIENCIA DE LA COMPUTACIÓN

IIC 2133 — Estructuras de Datos y Algoritmos
Interrogación 1
Primer Semestre, 2017

Duración: 2 hrs.

1. *a*) ¿Es correcto que si la rutina *Partition* siempre realiza el *mínimo* número de intercambios posibles, se garantiza que *QuickSort* ejecuta en el menor tiempo posible (es decir, está en el mejor caso)? Argumente.

   b) Escriba el pseudocódigo de una versión inestable (pero correcta) de Merge Sort. Ahora diga cómo transformarla en estable.

   c) Suponga que insertará los objetos $o_1, o_2, \ldots, o_n$, en orden, a un Min-Heap, usando la operación de inserción estándar. ¿Cómo podrían ser las claves de estos objetos para obtener el mejor caso? ¿Y para obtener el peor caso? Diga, con precisión, cuántos intercambios se producen en cada caso si se usan las rutinas vistas en clases.

Solución

a) Falso (**0.5 puntos**). Los peores casos de `QuickSort` son aquellos en los que ocurren muchas particiones desbalanceadas. Una partición desbalanceada es cuando el pivote, luego de `Partition`, queda al inicio o al final del arreglo, y esto no tiene relación alguna con el número de intercambios realizados (puede haber una partición desbalanceada con muchos intercambios así como una sin ningún intercambio). Si ocurren muchas particiones desbalanceadas en `QuickSort`, entonces, cada llamada recursiva procesará un arreglo de $n-1$ elementos (arreglo original menos el pivote), por lo que se llamará n-veces hasta llegar a un arreglo de largo 1 (el Heap de llamadas se vuelve lineal) (**1.5 puntos** por argumentar).

Otra forma sería mostrar un contraejemplo. Por ejemplo, ordenar un arreglo ordenado con `QuickSort` (**2 puntos** si el contraejemplo está correcto y bien fundamentado).

b) Un algoritmo estable es aquel que mantiene el orden del input original cuando el algoritmo de comparación no es capaz de distinguir dos o más elementos. Va a garantizar que cuando dos objetos son iguales, los ordenará de la misma forma en la que estaban originalmente.

En el algoritmo de `Merge`, se comparan los objetos del arreglo de la derecha y de la izquierda, poniendo primero en el arreglo original el que sea mas pequeño. Ahora, si cambiamos la comparación de $\leq$ por solo $<$ en el algoritmo, podemos obtener una versión inestable de `MergeSort`, pues cuando hayan dos objetos iguales no entrará en la primera condición del `If` y entrará en el `Else`, que es donde se ubica el objeto del arreglo de la derecha primero (**1 punto** por escribir el pseudocódigo inestable).

El cambio se encuentra en la linea 9 del siguiente pseudocodigo:

```
1: function MERGESORT(A, p, q, r)
2:     i ← 0, j ← 0
3:     L nuevo arreglo de tamaño q − p + 2
4:     R nuevo arreglo de tamaño r − q + 2
5:     L[0...q − p] ← A[p...q]
6:     R[0...r − q − 1] ← A[q + 1...r]
7:     L[q − p + 1] ← R[r − q] ← ∞
```

2017-1-I1-P1–QuickSort, MergeSort, HeapSort (2/2)

```
8:      for k ← p to r do
9:          if L[i] < R[j] then
10:             A[k] ← L[i]
11:             i+ = 1
12:         else
13:             A[k] ← R[j]
14:             j+ = 1
15:         end if
16:     end for
17: end function
```

Para hacerlo estable, se cambia $<$ por $\leq$ en la linea 9 (**1 punto** por decir cómo hacerlo estable).

c) Al insertar un elemento en un Min-Heap, este se coloca al final del arreglo y luego ”sube”hasta que sea mayor o igual que su Padre. `DecreaseKey`, toma en el peor caso una complejidad de $\mathcal{O}(log(n))$ (con n el tamaño del heap) y esto ocurre cuando el elemento insertado llega al inicio del arreglo (pues es el más pequeño). El algoritmo tomará el mejor caso cuando el elemento insertado es el mayor de todos, por lo que no debe modificar su posición y `DecreaseKey` se ejecuta en $\mathcal{O}(1)$.

Los intercambios se producen cuando el elemento insertado debe subir por el árbol (cuando es menor que su padre), por lo tanto, para obtener un máximo número de intercambios, todos los elementos insertados deben ser menores que todos los elementos sobre el, de modo de llegar hasta lo más arriba del heap a través de los intercambios. Por lo tanto, insertar n elementos de mayor a menor será el caso con mayor números de intercambios (**0.5 puntos**). El total será de (**0.5 puntos** por el número correcto):

$$\sum_{i=1}^{n}(\lfloor \log(i) \rfloor)$$

Para obtener un número mínimo de intercambios, todos los elementos insertados deben ser mayores que los elementos sobre el, de modo de mantenerse en el lugar donde fueron insertados y no subir por el heap a través de intercambios. Por lo tanto, insertar n elementos de menor a mayor será el caso con menor número de intercambios (**0.5 puntos**). El total será de cero intercambios (**0.5 puntos**).

2017-1-Ex-P1-a—c–QuickSort, RadixSort (1/1)



PONTIFICIA UNIVERSIDAD CATÓLICA DE CHILE
ESCUELA DE INGENIERÍA
DEPARTAMENTO DE CIENCIA DE LA COMPUTACIÓN

IIC 2133 — Estructuras de Datos y Algoritmos
Examen
Primer Semestre, 2017

Duración: 3 hrs.

1. Para cada una de las siguientes afirmaciones, diga si es *verdadera* o *falsa*, **siempre justificando** su respuesta.

   a) Quick Sort es un algoritmo de ordenación estable. **Respuesta:** Falso. Basta con dar un contraejemplo.

   b) Sea e la segunda arista más barata de un grafo dirigido acíclico G con más de dos nodos y aristas con costos diferentes. Entonces e pertenece al árbol de cobertura de costo mínimo para G. **Respuesta:** Verdadero. Si usamos el algoritmo de Kruskal, la segunda arista más barata es siempre agregada al MST puesto que no puede formar un ciclo en el bosque construido hasta el momento.

   c) Radix Sort es un algoritmo de ordenación que puede ordenar n números enteros y cuyo tiempo de ejecución está siempre en $\Theta(n)$. **Respuesta:** El tiempo de ejecución de Radix Sort es $d(n + k)$, cuando los datos están en $[0, k]$. Basta entonces con que k sea suficientemente grande (por ejemplo, exponencial en n), para que el algoritmo no sea $\Theta(n)$

   d) Sea A un árbol rojo-negro en donde cada rama tiene n nodos negros y n nodos rojos. Si al insertar una clave nueva en A se obtiene el árbol A', entonces cada rama de A' tiene $n + 1$ nodos negros. **Respuesta:** Verdadero. Un árbol rojo negro como A corresponde a un árbol 2-4 "completamente saturado"; es decir, uno que contiene nodos con 3 claves. Al insertar una clave nueva, el árbol 2-4 aumentará su altura en 1. Esto significa que su equivalente rojo-negro debe tener un nodo negro más por cada rama.

   e) Si se tienen dos tablas de hash, una con direccionamiento abierto y otra cerrado, las dos del mismo tamaño m y el mismo factor de carga α, ambas ocupan la misma cantidad de memoria.

   f) Sea A un árbol AVL y ℓ_1 y ℓ_2 los largos de dos ramas de A. Entonces $|\ell_1 - \ell_2| \leq 1$. **Respuesta:** Falso. Basta dar un contraejemplo

   g) La operación de inserción en un árbol AVL con n datos realiza a lo más una operación *restructure* pero toma tiempo $O(\log n)$. **Respuesta:** Verdadero, puesto que la inserción debe revisar que el balance esté correcto a lo largo de la rama donde se insertó el dato. Y esa rama tiene tamaño $O(\log n)$.

   h) Si A es un ABB y n es un nodo de A, el *sucesor* de n no tiene un hijo izquierdo. **Respuesta:** Falso. La propiedad no se cumple en general cuando n no tiene un hijo derecho.

   i) Es posible modificar la implementación de la estructura de datos para conjuntos disjuntos vista en clases, de manera que permita des-unir dos conjuntos en $O(1)$.

   j) Como diccionario, una tabla de hash es más conveniente que un árbol rojo negro en cualquier aplicación. (Sin considerar la dificultad de implementación).

   k) Sea p una secuencia de dos o más números diferentes y sea q una permutación de p distinta de p. Adicionalmente, sean A_p y A_q los árboles binarios de búsqueda que resultan de, respectivamente, insertar en orden los elementos de p y q en árboles binarios de búsqueda vacíos. Entonces A_p y A_q son distintos. **Respuesta:** Falso.

   l) Re-hashing, el procedimiento que construye una nueva tabla de hash a partir de otra existente, toma tiempo $O(n)$ en una tabla con colisiones resueltas por encadenamiento de tamaño m que contiene n datos. **Respuesta:** Falso. El tiempo depende del tamaño de la tabla y del número de datos; específicamente es $O(m + n)$.

2017-1-Ex-P3–QuickSort (1/1)

3. Existen diversas maneras de implementar Quick Sort. Una de ellas es reemplazar *Partition* por una función *MedianPartition*, que es tal que *MedianPartition*(A, p, r) es el índice en donde se ubica la *mediana* del arreglo $A[p..r]$, una vez que éste está ordenado.

   a) Modifique el pseudo-código de QuickSort—mostrado abajo—de manera que use esta idea. Su pseudocódigo debe ser detallado y completo; es decir, no debe faltar ninguna función por implementar. Asegure, además, que el tiempo promedio de *MedianPartition* sea $O(n)$.

```
1  function Partition(A, p, r)
2      i ← p − 1
3      j ← p
4      while j ≤ r do
5          if A[j] ≤ A[r] then
6              i ← i + 1
7              A[i] ↔ A[j]
8          j ← j + 1
9      return i
10 procedure Quick-Sort(A, p, r)
11     if p < r then
12         q ← Partition(A, p, r)
13         Quick-Sort(A, p, q − 1)
14         Quick-Sort(A, q + 1, r)
```

   b) Argumente a favor de que el tiempo promedio de su función *MedianPartition* es $O(n)$.

   c) ¿Es posible alimentar a su versión de Quick Sort con un arreglo A para que tome tiempo $O(n^2)$? Argumente.

   d) ¿Cómo espera que su algoritmo funcione en la práctica? ¿Cree que una variante de esta idea podría funcionar mejor? ¿Cuál?

2016-2-I2-P4–QuickSort, MergeSort, RadixSort (1/1)

4. **a**) Escribe un algoritmo que ordene una lista doblemente ligada *L* usando *quickSort*. Puedes usar de todas las operaciones pertinentes en listas, incluyendo *crear* y *destruir*.

b) Digamos que existe un algoritmo que permite mezclar dos listas ordenadas, obteniendo una lista ordenada, en tiempo O(1). Analiza la eficiencia de *mergeSort* usando este algoritmo para mezclar las listas

c) Explica qué propiedad de *radixSort* permite garantizar la ordenación de los elementos.

Pauta:

a) Si el algoritmo no es quicksort. 0 pts.

Si el algoritmo es quicksort pero no está hecho para listas, por ejemplo usa lista[i], lo que en una lista ligada es O(n). 0 pts.

Si el algoritmo es quicksort y tiene muchos errores. 1 pt.

Si el algoritmo es quicksort y tiene pocos o no tiene errores: 2pts.

b) 1 punto por plantear la ecuación de recurrencia o el problema maestro o plantear una sumatoria que resuelva el problema.

1 punto por resolver el problema con lo anterior.

c) 2 puntos si dice estabilidad o si explica la propiedad de estabilidad con palabras.

0 puntos en caso contrario.

2016-2-I2-P5–InsertionSort, QuickSort, MergeSort, ShellSort, CountingSort (1/2)

5. Para cada una de las siguientes situaciones, indica qué algoritmo visto en clase es el que más se beneficia, y cuál, el que se ve más perjudicado, justificando el porqué. Además, propón un algoritmo específico para el caso, que sea especialmente eficiente, indicando su complejidad. Si no hay algoritmos que se beneficien o perjudiquen, indícalo.

0.5 por cual se beneficia
0.5 por cual se perjudica
0.5 por el algoritmo propuesto eficiente

i. Datos ordenados en sentido contrario.

Beneficiado: No hay algoritmo beneficiado

Perjudicado: Insertion Sort, Shell Sort, QuickSort

Propuesto: Dar vuelta los elementos del arreglo. O(n)

ii. Datos ordenados de a pares, en que cada par de elementos $L[2*i]$ y $L[2*i+1]$ en el arreglo L cumplen que $L[2*i] < L[2*i+1]$.

Beneficiado: Insertion Sort, Shell Sort.

MergeSort no se beneficia porque a pesar de que venga ordenado de a pares, MergeSort no lo sabe y hará las comparaciones igual.

Perjudicado: No hay algoritmo perjudicado

Propuesto: MergeSort pero solo haciendo merge, saltándose las comparaciones iniciales O(n*log(n))

Ordenar de a pares e impares y luego hacer merge de ambos resultados O(n*log(n))

iii. Datos en que cada uno es un número natural entre 1 y 100.

Beneficiado: No hay algoritmo beneficiado.

Perjudicado: No hay algoritmo perjudicado.

2016-2-I2-P5–InsertionSort, QuickSort, MergeSort, ShellSort, CountingSort (2/2)

Propuesto: Counting Sort. O(n)

iv. Datos ordenados excepto por un dato insertado de manera aleatoria.

Beneficiado: Insertion Sort, Shell Sort

Perjudicado: QuickSort

Propuesto: Ej. (Listas) Encontrar el elemento, sacarlo y luego ingresarlo haciendo búsqueda binaria.

(Arreglo) Encontrar el elemento y hacer swap hasta que quede en su posición correcta.

Insertion Sort hasta que ordene un elemento.

Todos estos son O(n).

2016-1-I1-P4–MergeSort, QuickSort (1/1)

4. **a**) [**2**] Tenemos una lista de N números enteros positivos, ceros y negativos. Queremos determinar cuántos tríos de números suman 0. ¿Qué tan eficientemente puede resolverse este problema? Justifica.

Se puede hacer en tiempo $O(n^2 \log n)$: Primero, ordenamos la lista de menor a mayor, en tiempo $O(n \log n)$. Luego, para cada par de números, sumamos los dos números y buscamos en la lista ya ordenada, empleando búsqueda binaria, un número que sea el negativo de la suma; si lo encontramos, entonces incrementamos el contador de los tríos que suman 0. Hay $O(n^2)$ pares (los podemos generar sistemáticamente con dos *loops*, uno anidado en el otro) y cada búsqueda binaria se puede hacer en tiempo $O(\log n)$.

b) [**2**] Considera la siguiente versión de **`mergeSort`**:

```
mergeSort(a,b,e,w)   —a es el arreglo a ordenar entre a[e] y a[w]; b es un arreglo auxiliar
    if e >= w:
        return
    m = (e+w)/2
    mergeSort(a,b,e,m)
    mergeSort(a,b,m+1,w)
    if a[m] > a[m+1]:
        merge(a,b,e,m+1,w)   —el procedimiento merge estudiado en clase
```

Demuestra que el número de comparaciones usadas por este algoritmo para ordenar un arreglo **`a`** que ya está ordenado es lineal con respecto al tamaño n del arreglo.

La versión es diferente a la estudiada en clase debido a la línea **`if a[m] ≥ a[m+1]:`**. Esta línea implica que el procedimiento **`merge`**, que es el que ocupa un tiempo proporcional a **`w`** – **`e`**, se va a ejecutar sólo cuando el dato más grande de la primera mitad del arreglo **`a`** sea mayor que el dato más pequeño de la segunda mitad, lo cual ***no ocurre nunca*** ya que el arreglo está ordenado.

Luego, la ecuación de recurrencia para este caso es $T(n) = 2T(n/2) + c$, con c = constante

(c representa la ejecución de `if e >= w:`, con resultado falso, seguida de `m = (e+w)/2`),

cuya solución es $T(n) = nT(1) + (n–1)c$; como $T(1) = 1$, por la línea **`if e >= w:`**, entonces $T(n) = n + (n–1)c = O(n)$.

c) [**2**] Supongamos que elegimos el dato en la posición del medio del arreglo como pivote, en vez de elegir el dato en el extremo derecho (como en el procedimiento `partition` estudiado en clase). ¿Es ahora menos probable que **`quickSort`** requiera tiempo cuadrático? Justifica.

No, es igual de probable: siempre podría ocurrir que ese pivote es el dato más grande (o más pequeño) del (sub)arreglo.

2015-2-I2-P3–CountingSort, InsertionSort, MergeSort (1/1)

3. **a**) Da un algoritmo que, dados n enteros en el rango 0 a k, los preprocesa en tiempo $\Theta(n+k)$ y luego responde cualquier consulta acerca de cuántos de los n enteros están en el rango [a .. b] en tiempo O(1).

Suponemos que los n enteros vienen en un arreglo A. *La idea es usar la primera parte de countingSort, es decir, los primeros tres ciclos* `for`*, que son los que arman el arreglo* C:

```
for i = 0 to k: C[i] = 0
for j = 0 to n–1: C[A[j]] = C[A[j]] + 1
for i = 1 to k: C[i] = C[i] + C[i–1]
```

Con esto, el número de enteros en el rango [a .. b] *es* C[b] – C[a–1].

b) Justifica que *insertionSort* toma tiempo $\Omega(n^2)$ en promedio para ordenar n elementos.

insertionSort *ordena por la via de intercambiar la posición de dos elementos adyacentes (es decir, corregir una inversión), repitiendo esta acción todas las veces que sea necesario. La pregunta es, ¿cuántas inversiones hay en promedio en un arreglo de n elementos? Si consideramos el arreglo* $[a_1, a_2, \ldots, a_n]$ *y su inverso,* $[a_n, \ldots, a_2, a_1]$*, dos elementos cualquiera,* a_i *y* a_j*, están desordenados (invertidos) en el primer arreglo o en el segundo; es decir, cada dos arreglos, tenemos todas las inversiones posibles entre n elementos:* $n(n-1)/2 = \Omega(n^2)$.

c) Escribe una versión de mergeSort que tome tiempo O(n) para ordenar un arreglo de n elementos ***que ya está ordenado***. Justifica tu respuesta.

`mergeSort()` *mezcla dos subarreglos ordenados —*r[e .. m] *y* r[m+1 .. w]*— en un tercer subarreglo ordenado —*tmp[e .. w]*— que luego copia de vuelta a* r[e .. w]: *el algoritmo* `merge()`. *Sin embargo, si los elementos de* r[e .. m] *son todos menores que los elementos de* r[m+1 .. w]*, entonces no es necesario realizar la mezcla, ahorrando una cantidad de tiempo de ejecución proporcional a* w – e*, la cantidad total de elementos en los dos subarreglos originales. Para esto, en* `mergeSort()` *basta con probar si* r[m] < r[m+1] *justo antes de llamar a* `merge()`*; la llamada sólo se hace si* r[m] > r[m+1]. *Es decir:*

```
mergeSort(r, tmp, e, w):
   if e < w:
      int m = (e+w)/2
      mergeSort(r, tmp, e, m)
      mergeSort(r, tmp, m+1, w)
      if r[m] > r[m+1]:
         merge(r, tmp, e, m+1, w)
```

Si hacemos este cambio en `mergeSort()`*, entonces, si* `r` *está inicialmente ordenado,* `mergeSort()` *no va a hacer llamadas a* `merge()` *y la ejecución de todo el algoritmo va a tomar un tiempo proporcional a* n: *primero, divide* `r` *hasta formar subarreglos de tamaño 1, y, luego, mezcla pares de subarreglos ordenados, sólo que sin llamar a* `merge()`.

2015-2-Ex-P4–QuickSort (1/2)

4. Un componente esencial del algoritmo de ordenación **`quicksort()`** es el algoritmo de partición. Considera el siguiente algoritmo **`partition()`**:

```
partition(a, p, r):
   x = a[r]
   i = p-1
   for j = p … r-1:
      if a[j] ≤ x:
         i = i+1
         exchange(a[i], a[j])
   exchange(a[i+1], a[r])
   return i+1
```

a) Muestra el funcionamiento de **`partition()`** en el arreglo **`a`** = [B, H, G, A, C, E, F, D].

Inicialmente, p = 0, r = 7. x = 'D', i = –1

Ejecutamos el ciclo for j, con j = 0, 1, …, 6:
j = 0 –> a[j] = 'B' < x = 'D' –> i = 0 e intercambia (los contenidos de) a[0] y a[0] –> a queda igual
j = 1 –> a[j] = 'H' > x = 'D' –> no pasa nada
j = 2 –> a[j] = 'G' > x = 'D' –> no pasa nada
j = 3 –> a[j] = 'A' < x = 'D' –> i = 1 e intercambia a[1] y a[3] –> a = ['B', 'A', 'G', 'H', 'C', 'E', 'F', 'D']
j = 4 –> a[j] = 'C' < x = 'D' –> i = 2 e intercambia a[2] y a[4] –> a = ['B', 'A', 'C', 'H', 'G', 'E', 'F', 'D']
j = 5 –> a[j] = 'E' > x = 'D' –> no pasa nada
j = 6 –> a[j] = 'F' > x = 'D' –> no pasa nada

Así, al salir del ciclo for j, i = 2, y a = ['B', 'A', 'C', 'H', 'G', 'E', 'F', 'D']

Por lo tanto, finalmente intercambia a[3] y a[7] –> **a = ['B', 'A', 'C', 'D', 'G', 'E', 'F', 'H'] y retorna i+1 = 3**

La explicación anterior es la muestra más detallada del funcionamiento de **`partition()`**. Al menos, hay que mostrar las líneas en las que el contenido de **`a`** cambia, y la línea final; es decir, las tres líneas que están en **bold**.

2015-2-Ex-P4–QuickSort (2/2)

b) Demuestra que en todo momento durante la ejecución de **`partition()`**, un arreglo **`a`** cualquiera está dividido en cuatro sectores: **`a[p]`** … **`a[i]`**, que son valores menores o iguales que el pivote; **`a[i+1]`** … **`a[j-1]`**, que son valores mayores que el pivote; **`a[j]`** … **`a[r-1]`**, que son valores aún no procesados; y **`a[r]`**, que es el pivote.

Los que importan son los sectores **`a[p]`** … **`a[i]`** y **`a[i+1]`** … **`a[j-1]`**; simplemente mirando el código, los otros dos sectores son "obvios".

La afirmación es inicialmente verdadera, ya que los sectores **`a[p]`** … **`a[i]`** y **`a[i+1]`** … **`a[j-1]` no tienen elementos**, debido a los valores iniciales de los índices (los *rangos* de los índices son vacíos).

El procesamiento del arreglo **`a`** lo hace el ciclo **`for j`**, que revisa en orden cada uno de los valores **`a[p]`** … **`a[r-1]`**, lo que de paso demuestra que el sector **`a[j]`** … **`a[r-1]`** corresponde a los valores aún no procesados.

La revisión de **`a[j]`** produce un efecto sólo si **`a[j]`** ≤ **`x`** (el pivote **`x`** = **`a[r]`** es constante a lo largo de todo el ciclo):

- el efecto es que el sector **`a[p]`** … **`a[i]`**, de los elementos menores o iguales que el pivote, aumenta su tamaño en 1 (la instrucción **`i = i+1`**) para recibir a **`a[j]`** (por la vía de intercambiarlo con **`a[i]`**);
- en cambio, si **`a[j]`** > **`x`**, entonces lo único que ocurre es que **`j`** aumenta en 1.

Como el sector **`a[p]`** … **`a[i]`** es inicialmente vacío, lo anterior significa que sólo crece para recibir valores menores o iguales que el pivote, demostrando esta parte de la afimación.

Además, como cada valor ≤ **`x`** va a parar al sector **`a[p]`** … **`a[i]`**, y el sector **`a[j]`** … **`a[r-1]`** contiene los elementos aún no procesados, entonces el sector **`a[i+1]`** … **`a[j-1]`** necesariamente contiene valores ya procesados y mayores que **`x`**, demostrando esta otra parte de la afimación.

c) ¿Qué valor devuelve **`partition()`** cuando todos los elementos del arreglo **`a`** tienen el mismo valor?

Si todos los elementos de **`a`** son iguales, entonces para cada **`j`**, efectivamente **`a[j] ≤ x`**. Por lo tanto, para cada **`j`**, se incrementa **`i`** (y se intercambian elementos de **`a`**, pero esto no produce cambios en cómo se ve **`a`**); como **`j`** va de **`p`** a **`r-1`**, entonces **`i`**, que parte en **`p-1`**, llega a valer **`r-2`**.

Y como **`partition()`** siempre devuelve **`i+1`**, entonces **`partition()`** devuelve **`r-1`** cuando todos los elementos de **`a`** son iguales.

2015-1-I2-P1–InsertionSort, ShellSort (1/2)

Estructuras de Datos y Algoritmos – IIC2133

I2

11 mayo 2015

1. Una observación que hicimos en clase sobre los algoritmos de **ordenación por comparación de elementos adyacentes**, p.ej., *insertionSort*( ), es que su debilidad (en términos del número de operaciones que ejecutan) radica en que sólo comparan e intercambian posiciones de elementos adyacentes. Así, si tuviéramos un algoritmo que usara la misma estrategia general de *insertionSort*( ), pero que comparara elementos que están a una cierta distancia > 1 entre ellos, podríamos esperar un mejor desempeño.

a) Considera el siguiente arreglo **a** y calcula cuántas comparaciones entre elementos hace *insertionSort*( ) para ordenarlo de menor a mayor; muestra que entiendes cómo funciona *insertionSort*( ):

a = [11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1]

insertionSort( ) coloca el segundo elemento ordenado con respecto al primero, luego el tercero ordenado con respecto a los dos primeros (ya ordenados entre ellos), luego el cuarto ordenado con respecto a los tres primeros (ya ordenados entre ellos), etc. En el caso del arreglo **a**, *insertionSort*( ) básicamente va moviendo cada elemento, 10, 9, …, 1, hasta la primera posición del arreglo. Para ello, el 10 es comparado una vez (con el 11), el 9 es comparado dos veces (con el 11 y con el 10), el 8 es comparado tres veces (con el 11, el 10 y el 9), y así sucesivamente; finalmente, el 1 es comparado 10 veces. Luego el total de comparaciones es 1 + 2 + 3 + ... + 10 = **55**.

2015-1-I2-P1–InsertionSort, ShellSort (2/2)

b) Considera ahora el siguiente algoritmo de ordenación, *shellSort*( ), y calcula cuántas comparaciones entre elementos hace para ordenar el mismo arreglo **a** de menor a mayor. Muestra que entiendes cómo funciona este algoritmo; en particular, ¿qué relación tiene con *insertionSort*( )?

```
void shellSort(a)
   gaps[] = {5,3,1}
   for (t = 0; t < 3; t = t+1)
      gap = gaps[t]
      for (j = gap; j < a.length; j = j+1)
         tmp = a[j]
         k = j
         for (; k >= gap && tmp < a[k-gap]; k = k-gap)
            a[k] = a[k-gap]
         a[k] = tmp
```

[Primero, notemos que un algoritmo de ordenación por comparación muy afortunado podría ordenar el arreglo **a** haciendo sólo 5 comparaciones (y los consiguientes intercambios): 11 con 1, 10 con 2, 9 con 3, 8 con 4, y 7 con 5; este sería el mejor caso.]

En el caso de *shellSort*( ), notemos que las comparaciones entre elementos del arreglo se dan sólo en la comparación **`tmp < a[k-gap]`**; el algoritmo realiza **11** de estas comparaciones con resultado **`true`** y otras **17** con resultado **`false`**; en total, **28**.

Primero, realiza *insertionSort* entre elementos que están a distancia 5 entre ellos (según las posiciones que ocupan en **a**, no en cuanto a sus valores): el 6 con respecto al 11, el 5 c/r al 10, el 4 c/r al 9, el 3 c/r al 8, el 2 c/r al 7, el 1 c/r al 11, y el 1 c/r al 6.

Luego, realiza *insertionSort* entre elementos que están a distancia 3 (nuevamente, según sus posiciones en el arreglo): el 2 c/r al 5 y el 7 c/r al 10.

Finalmente, realiza *insertionSort* entre elementos que están a distancia 1: el 3 c/r al 4 y el 8 c/r al 9; estos son los dos únicos pares de valores que aún están "desordenados" al finalizar el paso anterior.

2015-1-I2-P2–MergeSort (1/2)

2. **a**) Describe el algoritmo de ordenación *mergeSort*( ) con un nivel de detalle similar al visto en clase, y **calcula su complejidad** en notación O( ).

Esto está en los apuntes de clase. La descripción puede ser en prosa, pero tiene que referirse explícitamente a cada uno de los grandes temas del algoritmo: las dos llamadas recursivas, c/u sobre una mitad del arreglo; y la mezcla posterior y, muy importante, cómo se hace ésta (bastaría con explicar el primer `while`, que es lo básico de la mezcla).

```
mergeSort(a, tmp, e, w)
   if ( e < w )
      m = (e+w)/2
      mergeSort(a, tmp, e, m)
      mergeSort(a, tmp, m+1, w)
      merge(a, tmp, e, m+1, w)

merge(a, tmp, e, m, w)
   p = e, k = e, q = m
   while ( p <= m-1 && q <= w )
      if ( a[p].key < a[q].key )
         tmp[k] = a[p]; k = k+1; p = p+1
      else
         tmp[k] = a[q]; k = k+1; q = q+1
   while ( p <= m-1 )
      tmp[k] = a[p]; k = k+1; p = p+1
   while ( q <= w )
      tmp[k] = a[q]; k = k+1; q = q+1
   for ( k = e; k <= w; k=k+1 )
      a[k] = tmp[k]
```

El análisis también está en los apuntes de clase. Esta es una posibilidad, pero hay otras.

Sabemos que $T(1) = 0$ y $T(n) = 2T(n/2) + n$

Luego,

$T(n)/n = T(n/2)/(n/2) + 1$
$T(n/2)/(n/2) = T(n/4)/(n/4) + 1$
$T(n/4)/(n/4) = T(n/8)/(n/8) + 1$
... ...
$T(2)/2 = T(1)/1 + 1$

Si sumamos ambos lados del signo = y cancelamos los términos que aparecen a ambos lados, obtenemos

$T(n)/n = T(1)/1 + \log n$

2015-1-I2-P2–MergeSort (2/2)

b) Un paso fundamental de *mergeSort*( ) es mezclar dos subarreglos ya ordenados, de modo de formar un solo subarreglo ordenado con los elementos de ambos subarreglos. Esto requiere usar un arreglo auxiliar y finalmente copiar el contenido del arreglo auxiliar de vuelta al arreglo original (en las posiciones que correspondan). Sin embargo, si el contenido inicial de este arreglo original ya está ordenado, entonces no es necesario ejecutar la mezcla; **describe un cambio** (simple) que puedes hacer a tu algoritmo de (a) para tomar en cuenta esta situación.

De lo que se trata es de no hacer la mezcla cuando, después de las dos llamadas recursivas, tenemos que **`a[m] < a[m+1]`**; como en ese punto el subarreglo **`a[e]`**, ..., **`a[m]`** está ordenado y el subarreglo **`a[m+1]`**, ..., **`a[w]`** también está ordenado, entonces si **`a[m] < a[m+1]`** significa que todo el subarreglo **`a[e]`**, ..., **`a[w]`** está ordenado. Básicamente, hay que verificar que este *no sea el caso* justo antes de hacer la mezcla.

```
mergeSort(a, tmp, e, w)
   if ( e < w )
      m = (e+w)/2
      mergeSort(a, tmp, e, m)
      mergeSort(a, tmp, m+1, w)
      if ( a[m] >= a[m+1] )
         merge(a, tmp, e, m+1, w)
```

c) **Calcula la complejidad** en notación O( ) de tu algoritmo modificado de (b) *cuando se ejecuta sobre un arreglo que viene totalmente ordenado*.

Suponiendo que el arreglo tiene n elementos, la complejidad en este caso será **O(n)** (en lugar de O($n$log$n$)), que básicamente representa lo que cuesta reconocer que el arreglo está ordenado.

Lo que pasa, en esta versión levemente modificada de *mergeSort*, es que cuando las llamadas recursivas empizan a "volver", lo único que hacen con los elementos del arreglo es la comparación del nuevo **`if`**, y como estas siempre resultan falsas (ya que el arreglo está ordenado), entonces *cada instancia de una llamada recursiva* es **O(1)**.

¿Cuántas "instancias" hay? Las instancias más pequeñas (cuando la recursión se detiene) son cuando los subarreglos son de tamaño 1 (**`e == w`**); de éstas hay n, pero no involucran comparaciones entre elementos del arreglo. En el nivel inmediatamente anterior hay, aproximadamente, $n/2$; en el anterior, $n/4$; y así sucesivamente hasta el primer nivel, en que hay una llamada: $n/2 + n/4 + ... + 1 = n$.

2015-1-I2-P3–QuickSort (1/1)

3. **a**) Considera la siguiente afirmación respecto a un algoritmo de ordenación de strings del mismo largo:

> "Otra forma de establecer la demostración de que el algoritmo es correcto es pensar en el futuro: si los caracteres que aún no han sido examinados para un par de claves son idénticos, entonces cualquier diferencia entre las claves está restringida a los caracteres que ya han sido examinados, de modo que las claves han sido ordenadas apropiadamente y permanecerán así debido a la estabilidad. Si, por el contrario, los caracteres que no han sido examinados son diferentes, entonces los caracteres ya examinados no importan y una pasada posterior ordenará correctamente el par de claves basado en esas diferencias más significativas."

¿Exactamente a cuál algoritmo para ordenación de strings se refiere esta afirmación? **Justifica**.

Se refiere al algoritmo que en clase llamamos *radixSort*, que ordena los strings ordenándolos primero según el carácter menos significativo —el de más a la derecha—, luego según el siguiente carácter menos significativo —el que está inmediatamente a la izquierda del que está más a la derecha—, y así sucesivamente hasta finalmente ordenar los strings según el carácter más significativo —el de más a la izquierda. La ordenación de los strings según un carácter determinado debe hacerse usando un algoritmo de ordenación *estable*, p.ej., *countingSort*.

b) Otra forma de ordenar strings, especialmente cuando son de diferentes largos, es considerar los caracteres de izquierda a derecha y usar el siguiente método recursivo: Usamos *countingSort*( ) para ordenar los strings de acuerdo con el primer carácter; luego, recursivamente, ordenamos los strings que tienen un mismo primer carácter (excluyendo este primer carácter). Similarmente a *quickSort*( ), este algoritmo particiona el arreglo de strings en subarreglos que pueden ser ordenados independientemente para completar la tarea, sólo que particiona el arreglo en un subarreglo para cada posible valor del primer carácter, en lugar de las dos particiones de *quickSort*( ). **Escribe este algoritmo**.

```
sortString(a, e, w, k):
—ordena recursivamente el arreglo a de strings, desde a[e] hasta a[w], a partir del k-ésimo carácter
—convierte cada carácter a un dígito entre 0 y 127 mediante la función (ficticia) dgt
—usa arreglos auxiliares b y c
   if e < w:
      for i = e, ..., w:
         p = dgt(a[i][k])
         b[p+2] = b[p+2]+1
      for r = 0, ..., 128:
         b[r+1] = b[r+1] + b[r]
      for i = e, ..., w:
         p = dgt(a[i][k])
         c[b[p+1]] = a[i]
         b[p+1] = b[p+1]+1
      for i = e, ..., w:
         a[i] = c[i-e]
      for r = 0, ..., 127:
         sortString(a, e+b[r], e+b[r+1]-1, k+1)
—los cuatro primeros for implementan countingSort()
—el quinto for hace las llamadas recursivas sobre cada una de las particiones (strings con el mismo k-ésimo carácter)
```

2015-1-Ex-P2–MergeSort modificado (1/1)

2. *Mergesort natural* es una versión *bottom-up* de mergesort; su paso fundamental es el siguiente: primero encuentra un subarreglo ordenado (incrementa un puntero hasta encontrar un dato que sea menor que su predecesor en el arreglo), luego encuentra el próximo subarreglo ordenado, y luego los mezcla.

Escribe un algoritmo —lo más parecido a código posible— eficiente que implemente mergesort natural para ordenar una *lista ligada* (y no un arreglo; este es el método preferido para ordenar listas ligadas, porque no usa espacio adicional). ¿Qué complejidad tiene tu algoritmo? Justifica.

[**2/3**] Algoritmo (una posibilidad, hay otras):

[1/3] ***—identificar las sublistas a mezclar y la condición de terminación del algoritmo***

```
sea P = puntero al elemento vigente de la lista (inicialmente, al primer elemento)
a partir de P, recorrer la lista mientras los elementos no decrezcan en valor.
sea Q = puntero al último elemento "no decreciente"
sea R = puntero al siguiente elemento en la lista
if R = null
    terminar, la lista está ordenada
else
    a partir de R, recorrer la lista mientras los elementos no decrezcan en valor.
    sea S = puntero al último elemento no decreciente
    elemento vigente = S.next
    Mezclar la sublista de P a Q con la sublista de R a S (*)
    volver a la primera instrucción
```

[1/3] ***—hacer la mezcla (el algoritmo* merge *visto en clase no sirve, porque mezcla arreglos y no listas)***

```
(*) Mezclar la sublista de P a Q con la sublista de R a S:
    T = T3 = new(Node)
    T1 = P, T2 = R
    if T1 < T2
        T3 = T1, T1 = T1.next
    else
        T3 = T2, T2 = T2.next
    if T1 = Q.next
        agregar a a T3 la lista de T2 a S
    else
        if T2 = S.next
            agregar a T3 la lista de T1 a Q
        else
            repetir el primer if
    P = T
```

[**1/3**] Complejidad = O(n log$n$), ya que en el peor caso, cuando la lista está ordenada "al revés", primero se mezclan sublistas de largo 1; luego, de largo 2; luego, de largo 4; luego, de largo 8, etc. Como el largo de las sublistas que se mezclan crece al doble cada vez, a lo más hay log$n$ "pasadas" por la lista completa. En cada pasada, hay O(n) operaciones de mezcla, independientemente del largo de las sublistas que se mezclan, ya que de una manera u otra hay que mezclar todos los elementos de la lista. ¡ **Ojo: Se asigna puntaje sólo si está bien justificado** !

2014-2-I1-P2–QuickSort (1/2)

2. Desafortunadamente, un arreglo que contiene elementos con un misma clave repetida muchas veces lleva a QuickSort a desempeñarse cercano a su peor caso. En el extremo, con un arreglo con un único valor para la clave de ordenación, el algoritmo es cuadrático.

   a) Muestre cómo modificar $\mathrm{Partition}(A, p, r)$ visto en clases **sin agregar** un *keyword* de iteración adicional[1], de tal forma de evitar malos casos con muchas claves repetidas.

   Respuesta: Una modificación correcta es la que se describe a continuación.

```
1:  function PARTITION(A, p, r)
2:      x ← A[r]
3:      i ← p − 1
4:      mid ← ⌊(p + r)/2⌋
5:      for j ← p to mid do
6:          if A[j] ≤ x then
7:              i ← i + 1
8:              SWAP(A[i], A[j])
9:      for j ← mid + 1 to r − 1 do
10:         if A[j] < x then
11:             i ← i + 1
12:             SWAP(A[i], A[j])
13:     SWAP(A[i + 1], A[r])
14:     return i + 1
```

   Existen muchas otras soluciones correctas.

   b) Argumente por qué su algoritmo resuelve el problema y por qué es correcto.

   Respuesta: Este algoritmo resuelve el problema ya que cuando hay muchas claves repetidas, tenderá a hacer particiones más equitativas que el algoritmo tradicional.

[1] Esta restricción tiene sentido hacerla por eficiencia.

2014-2-I1-P2–QuickSort (2/2)

Consideremos el caso en que todas las claves son iguales. Cuando particionamos un arreglo de tamaño n, el algoritmo visto en clases lo separará en arreglos de tamaño 1 y $n-1$. Esto causará que QuickSort tome tiempo en $\Theta(n^2)$.

El algoritmo presentado, en cambio, particionará tal arreglo en particiones de tamaño $\lceil n/2 \rceil$ y $\lfloor n/2 \rfloor$. En este caso, QuickSort tomará tiempo en $\Theta(n \log n)$.

2014-1-I2-P1–InsertionSort (1/1)

Estructuras de Datos y Algoritmos – IIC2133

I2

19 mayo 2014

Ordenación

1) Explica cómo ordenar una baraja de naipes por pinta (en el orden espadas, corazones, diamantes y tréboles) y por rango dentro de cada pinta (as, rey, reina, *jack*, 10, 9, …, 2), con la restricción de que las cartas están dispuestas cara abajo en una fila, y las únicas operaciones permitidas son mirar los valores de dos cartas e intercambiar las posiciones de dos cartas (manteniéndolas cara abajo).

Respuesta:

insertionSort, pero con intercambios (también *selectionSort* con intercambios, que se convierte en algo así como *bubbleSort*).

2014-1-I2-P2–MergeSort (1/1)

2) *Mergesort natural* es una versión *bottom*-up de mergesort que aprovecha el orden que pueda haber en el arreglo original: encuentra un subarreglo ordenado (incrementa un puntero hasta encontrar un dato que sea menor que su predecesor en el arreglo), luego encuentra el próximo subarreglo ordenado, y luego los mezcla.

Escribe un algoritmo (en pseudo código) de mergesort natural para ordenar una *lista ligada*. (Este es el método preferido para ordenar listas ligadas porque no usa espacio adicional y su orden de crecimiento es de la forma $n\log n$.)

Respuesta:

El algoritmo tiene dos partes. En la primera, se busca dos sublistas (consecutivas) ordenadas, como se describe en el primer párrafo, y se las indentifica con punteros al inicio y al final de cada una. En la segunda parte, se hace la mezcla, usando la misma estrategia general de *mergesort*, pero mezclando listas ligadas en lugar de arreglos. Esto significa que no se van traspasando valores del arreglo original a un segundo arreglo, sino que se van revinculando los punteros de las listas.

2014-1-I2-P3–QuickSort (1/1)

3) Si ordenamos el archivo de empleados de la Universidad Católica por año de nacimiento, muy probablemente habrá muchas claves duplicadas. Si bien quicksort se desempeña bien en esta situación, es posible mejorar mucho su desempeño; p.ej., un subarreglo que consiste sólo de ítemes iguales (todos tienen la misma clave) no necesita seguir siendo procesado, pero quicksort sigue particionándolo en subarreglos más pequeños. Una posibilidad es particionar el arreglo en tres partes: una para los ítemes con claves menores que el pivote, otra para los ítemes con claves iguales al pivote, y otra para los ítemes con claves mayores que el pivote.

Escribe un algoritmo (en pseudo código) de *partición en 3* que haga una pasada de izquierda a derecha sobre el arreglo $a[e \ldots w]$, manteniendo tres punteros: un puntero *less* tal que $a[e \ldots less–1]$ son menores que el pivote v; un puntero *greater* tal que $a[greater+1 \ldots w]$ son mayores que v; y un puntero i tal que $a[less \ldots i–1]$ son iguales a v y $a[i \ldots greater]$ aún no han sido examinados.

Respuesta:

```
less = e
i = e+1
greater = w
while (i <= greater)
      if (a[i] < v)
            exchange(a[less], a[i])
            less = less+1
            i = i+1
      else
      if (a[i] > v)
            exchange(a[i], a[greater])
            greater = greater-1
      else
            i = i+1
return (less, greater)
```

2014-1-Ex-P3–MergeSort, estrategia Dividir y Conquistar (1/1)

3) Dada la secuencia de enteros A_1, A_2, …, A_N, posiblemente negativos, queremos encontrar el valor de la suma de aquella subsecuencia que sume más. P.ej., para la secuencia –2, 11, –4, 13,–5, –2, la respuesta es 20 (= 11 + (–4) + 13).

Da un algoritmo del tipo *dividir para conquistar* para resolver este problema en tiempo O(N log$N$); justifica.

Como sabemos, para que un algoritmo dividir para conquistar corra en tiempo O(N log$N$), normalmente tiene que dividir el problema en dos subproblemas (de la misma naturaleza y) del mismo tamaño y luego, al combinar las soluciones a los subproblemas para encontrar la solución al problema original, tiene que tomar tiempo O(N) (similarmente a *mergeSort*).

En este caso, la subsecuencia que suma más es una que está contenida completamente en la mitad izquierda de la secuencia, o bien una que está contenida completamente en la mitad derecha (hasta aquí, la división), o bien una que empieza en la mitad izquierda y termina en la mitad derecha (esta es la combinación). Buscar la subsecuencia que suma más en cada una de las mitades, izquierda y derecha, se puede hacer recursivamente, y es lo que aporta el factior log$N$ al tiempo de ejecución del algoritmo.

¿Cómo encontramos la subsecuencia que suma más y que cruza desde la mitad izquierda a la derecha? La parte izquierda de esta subsecuencia tiene que ser la subsecuencia de la mitad izquierda que sume más y que incluya el elemento en el extremo derecho de esta mitad; y, análogamente, la parte derecha tiene que ser la subsecuencia de la mitad derecha que sume más y que incluya el elemento en el extremo izquierdo de esta mitad. Ambas subsecuencias se determinan revisando linealmente —O(N)— la mitad correspondiente, de izquierda a derecha y de derecha a izquierda, respectivamente.

2013-2-I1-P3–HeapSort, Heap (1/1)

3. *Ordenación a base de heaps.* Si bien los datos almacenados en un *max-heap* cumplen una ordenación sólo parcial—**a**) [**1 pt.**] ¿cuál?—en la práctica es posible ordenarlos totalmente y de manera eficiente, tanto en términos de uso de memoria, como en cuanto al número de pasos ejecutados.

La clave almacenada en un nodo es mayor que las claves almacenadas en los hijos del nodo.

La observación que hay que hacer es la siguiente: si sacamos un dato del *max-heap*, ejecutando `xMax()`, sale el dato más grande almacenado en el *max-heap* (y el número de datos almacenados se reduce en uno, es decir, `heapSize = heapSize – 1`). Si a continuación sacamos otro dato, nuevamente sale el dato más grande (y nuevamente `heapSize = heapSize – 1`); este nuevo dato corresponde al segundo más grande de los datos almacenados originalmente.

b) [**3 pts.**] Aprovechando esta observación y suponiendo que el *max-heap* está almacenado en un arreglo `a`, escribe un método `heapSort()` que ordene totalmente, de menor a mayor, los datos del *max-heap*, sin usar un arreglo adicional (es decir, los datos quedan ordenados en el mismo arreglo `a`).

```
heapSort()
    while (heapSize > 0)
        x = xMax() —el método visto en clase
        a[heapSize+1] = x
```

c) [**2 pts.**] Calcula (y justifica) el número de pasos que ejecuta el método, en notación O( ) y en función del número n de datos almacenados originalmente en el *max-heap*.

Cada llamada a `xMax()` ejecuta O(log n) pasos, ya que a su vez hace una llamada a `heapify(1)`. Como después de cada llamada, el tamaño del heap se reduce en 1 (y no vuelve a aumentar), tenemos que el número total de pasos es

O(log n) + O(log n–1) + O(log n–2) + ... + O(log 1) = O(n log n)

2013-2-I2-P4–QuickSort (1/1)

4. *quickSort*.

a) ¿Qué valor de q (o j) devuelve `partition` cuando todos los elementos del arreglo `A[p…r]` son iguales? Justifica.

Respuesta
Devuelve $q = (p+r-1)/2$. Los `while`'s internos no se ejecutan, ya que los elementos del arreglo son iguales al pivote, no mayores ni menores. Por lo tanto, el efecto del `while` externo es que en cada iteración `j` se incrementa en 1 y `k` se decrementa en 1 (y el `exchange` dentro del `while` intercambia de posición dos elementos iguales). Así, `j` se "encuentra" con `k` en la mitad del rango `p`, …, `r`–1 (ya que `k` parte en `r`–1), y en ese punto se detiene el `while`. El `exchange` fuera del while no cambia el valor de `j`.

b) ¿Cuál es el tiempo de ejecución de `quickSort` cuando todos los elementos del arreglo `A` son iguales? Justifica.

Respuesta
O($n$log$n$). Como vimos en clase, cuando `partition` produce dos particiones de similar tamaño, `quickSort` tiene su mejor desempeño. Y de **a**), este es el caso cuando todos los elementos del arreglo son iguales.

c) Considera la siguiente versión de `quickSort`, que sólo hace la primera llamada recursiva:

```
quickSort'(A, p, r)
while (p < r)
   q = partition(A, p, r)
   quickSort'(A, p, q-1)
   p = q+1
```

Explica por qué `quickSort'` ordena correctamente el arreglo `A`.

Respuesta
`quickSort'` hace la misma partición que `quickSort`, y luego también se llama recursivamente con parámetros `A`, `p` y `q-1`. La diferencia es que en este punto `quickSort` se llama recursivamente por segunda vez con parámetros `A`, `q+1` y `r`; en cambio, `quickSort'` asigna `p = q+1` y hace otra iteración del `while`. Sin embargo, esto ejecuta las mismas operaciones que la segunda llamada recursiva, con `A`, `q+1` y `r`, ya que en ambos casos `A` y `r` tienen los mismos valores que antes y `p` tiene el valor antiguo de `q+1`.

2013-2-Ex-P3–Comparación de estrategias de ordenación (1/1)

3) Considera las siguientes tres formas de resolver el problema de selección, consistente en encontrar el k-ésimo dato más pequeño de un conjunto de n datos:

i) Ordena los datos de menor a mayor, y elige el k-ésimo.

ii) Coloca los datos en un *min-heap* y ejecuta k operaciones *xMin*.

iii) Usa el algoritmo `randomSelect()` —similar a `quicksort()`, pero que sólo hace recursión sobre una de las dos partes definidas por `randomPartition()`.

Explica las ventajas y desventajas de cada forma y bajo cuáles condiciones usarías unas u otras. Considera aspectos tales como tiempo de ejecución, cantidad n de datos, fracción k/n, frecuencia de ejecución de la selección, y conjunto fijo de datos frente a datos que cambian (se agregan unos y se eliminan otros) a lo largo del tiempo.

i) toma O(n log n) para ordenar los n datos; luego, ***cualquier*** *selección es O(1). Si los datos son fijos y hay que hacer muchas selecciones, éste es el método preferido.*

ii) toma O(n) para colocar los n datos en un heap; luego, cada xMin toma O(log n). Encontrar el k-ésimo dato más pequeño toma O(n) + O(k log n), por lo que funciona mejor para casos con $k/n \approx 0$. Por otra parte, sacar los k objetos del heap significa reconstruir el heap para otra consulta o tener una segunda copia guardada, lo que agrega complejidad en ejecución o en uso de memoria.

iii) toma O(n) en promedio para entregar el k-ésimo dato más pequeño. Si hay que hacer una sola selección, éste es el método preferido. Incluso si hay que hacer un núnero pequeño de selecciones (< logn), éste es el método preferido. Por otra parte, si los datos cambian entre una selección y otra, éste es el mejor método.

2013-2-Ex-B–Estabilidad de algoritmos de ordenación (1/1)

B) Con respecto a los siguientes algoritmos de ordenación,

i) *Insertionsort* ii) *Mergesort* iii) *Heapsort* iv) *Quicksort*

a) ¿Cuáles son estables y cómo se sabe que lo son? Recuerda que un algoritmo de ordenación es ***estable*** si los datos con igual valor aparecen en el resultado en el mismo orden que tenían al comienzo

[1] Insertionsort y Mergesort son estables.

[1] Insertionsort (diapositiva #19 de los apuntes) es estable porque al comparar las claves de b y a[j-1], "subimos" (o avanzamos hacia a[0]) solo si la clave de b es estrictamente menor que la de a[j-1].

[1] Mergesort (diapositiva # 26 de los apuntes) es estable porque al comparar el a[p], de la primera "mitad", con el a[q], de la segunda mitad, colocamos a[p] en el resultado temporal (tmp) solo si su clave es estrictamente menor que la de a[q].

b) Para los que no son estables, explica cómo se los puede hacer estables y a qué costo.

Heapsort y Quicksort no son estables.

[2] Una forma de hacerlos estables es guardar el índice de cada elemento (la ubicación del elemento al comienzo) junto con el elemento. Así, cuando comparamos dos elementos, los comparamos según sus valores (key) y, si son iguales, usamos los índi-ces para decidir.

[1] Por cada elemento, se necesita almacenar adicionalmente su índice original. Si los índices van entre 0 y n, cada uno se puede almacenar en log n bits. Así, en total, se necesita n log n espacio adicional.

2013-1-I2-P1–QuickSort (1/2)

Estructuras de Datos y Algoritmos – IIC2133

I2

10 mayo 2013

1. Considera el algoritmo de ordenación *quickSort* que estudiamos en clase; a continuación repetimos una de sus partes, el algoritmo *partition*.

```
int partition(T[] a, int e, int w) {
   int j = e-1, k = w
   T v = a[w]
   while (true) {
      j = j+1
      while (a[j] < v) j = j+1
      k = k-1
      while (a[k] > v) {
         k = k-1
         if (k == e) break
      }
      if (j >= k) break
      exchange(a[j],a[k])
   }
   exchange(a[j],a[w])
   return j
}
```

a) ¿Cuál es el tiempo de ejecución de *quickSort* cuando todos los elementos del arreglo a son iguales? Justifica tu respuesta.

Respuesta:

Supongamos que al hacer la llamada a *partition*, $e = 0$ y $w = n$. Al ejecutarse la primera iteración del *while* exterior, $j = 1$. Como $a[j]$ no es menor que v (todos los elementos de a son iguales), el primer *while* interior no se ejecuta, y j queda en 1. Similarmente, $k = n\text{-}1$ inicialmente, y, como $a[k]$ no es mayor que v, k queda en $n\text{-}1$. Al ejecutarse la segunda iteración del *while* exterior, $j = 2$ y $k = n\text{-}2$; y así sucesivamente. Es decir, al terminar una iteración del *while* exterior, $k = n\text{-}j$. Como el *while* exterior termina cuando $j = k$, tenemos que $j = n/2$. Por lo tanto, $O(n \log n)$.

2013-1-I2-P1–QuickSort (2/2)

b) Escribe una versión no recursiva (y, por lo tanto, iterativa) de *quickSort*. Supón que dispones de un *stack* de números enteros con las operaciones *push*( ), *pop*( ) y *empty*( ) habituales. La idea es que después de llamar a *partition*, en lugar de llamar recursivamente a *quickSort* dos veces, coloques en el stack los índices correspondientes a cada una de las dos partes. Así, tu versión iterativa debe iterar mientras el stack no esté vacío.

Respuesta:

```
void iterQuicksort(int[] a)
   Stack s = new Stack()
   s.push(0); s.push(a.length-1)          Iniciar el stack con los índices extremos del arreglo original.
   int e, m, w
   while (!s.empty())
     w = s.pop(); e = s.pop()             Sacar del stack los índices extrremos de la parte a procesar.
     if (e < w)                           Verificar si la parte tiene al menos dos elementos.
            m = partition(a, e, w)
            s.push(e); s.push(m-1)        Colocar en el satck los índices extremos de la parte izquierda.
            s.push(m+1); s.push(w)        Colocar en el stack los índices extremos de la parte derecha.
```

c) Con respecto a la pregunta **b**) anterior, ¿qué se puede hacer, al momento de colocar en el stack los índices correspondientes a cada una de las dos partes, de modo de minimizar la profundidad máxima del stack? Justifica tu respuesta.

Respuesta:

Hay que colocar primero en el stack los índices de la parte más grande. **Falta justificar.**

2012-1-I2-P1–Quicksort, SelectionSort, MergeSort, HeapSort (1/3)

IIC2133 - Estructuras de datos y algoritmos I-2012 – Interrogación 2

Estructuras de datos y algoritmos I-2012

Interrogación n° 2

Instrucciones generales

- Ingresa tu nombre en todas las hojas de respuesta
- Entrega 1 hoja por pregunta, independiente si no respondiste la pregunta
- Lee cuidadosamente cada pregunta
- Dispones de 120 minutos

Pregunta 1 – Ordenación (20 pts)

a) (8 pts) Escribe en seudo código el algoritmo de ordenación quicksort.

```
/** 5 pts */
void quicksort(int* arr, int from, int to) {
        if (to – from <= 0) return;
        int index_pivote = pivotear(arr,from,to);
        quicksort(arr,from,index_pivote -1);
        quicksort(arr,index_pivote+1,to);
}
// aquí el pivoteo es clave. Usaremos from como pivote
/** 3 pts */
int pivotear((int* arr,int from,int to) {
        int index_pivote = from;
        int valor_pivote = arr[index_pivote];
        for (int i=from+1; i<=to; i++){
                if (arr[i] < valor_pivote){
                        index_pivote++;
                        swap(arr,i,index_pivote);
                }
        }
        swap(arr,from,index_pivote);
        return index_pivote;
}
```

2012-1-I2-P1–Quicksort, SelectionSort, MergeSort, HeapSort (2/3)

IIC2133 - Estructuras de datos y algoritmos I-2012 – Interrogación 2

```
/* También podia hacerse con listas. */
/** 5 pts */
void quicksort(List& values) {
        if (values.size() <= 1) return;
        List menores;
        List mayores;
        int pivote = pivotear(values,menores, mayores);
        quicksort(menores);
        quicksort(mayores);
        values.clear(); // vacia la lista
        values.addAll(menores);
        values.add(pivote);
        values.add(mayores);
}
/** 3 pts */
int pivotear(List& values, List& menores, List& mayores) {
        Iterator it = values.iterator();
        int pivote = it.next(); //sabemos que hay al menos 1
        while (it.hasNext()) {
                int val = it.next();
                if (val <= pivote) menores.add(val);
                else mayores.add(val);
        }
        return pivote;
}
```

b) (6 pts) Aplica quicksort al arreglo [9, 5, 3, 1, 6, 21, 13, 11, 20, 25, 23], mostrando el estado del arreglo antes de elegir un pivote, marcando el pivote y mostrando el resultado de usar dicho pivote.

```
/* Puntaje: 0-4-6. 6 = Perfecto, 4 = Algún error menor, 0 = otros casos.*/
[9, 5, 3, 1, 6, 21, 13, 11, 20, 25, 23]
[9, 5, 3, 1, 6, 21, 13, 11, 20, 25, 23]
[5, 3, 1, 6, 9, 21, 13, 11, 20, 25, 23]
```

2012-1-I2-P1–Quicksort, SelectionSort, MergeSort, HeapSort (3/3)

[5, 3, 1, 6, 9, 21, 13, 11, 20, 25, 23]

[3, 1, 5, 6, 9, 13, 11, 20, 21, 25, 23]

[3, 1, 5, 6, 9, 13, 11, 20, 21, 25, 23]

[1, 3, 5, 6, 9, 11, 13, 20, 21, 23, 25]

[1, 3, 5, 6, 9, 11, 13, 20, 21, 23, 25]

[1, 3, 5, 6, 9, 11, 13, 20, 21, 23, 25]

c) (6 pts) Identifica el algoritmo de ordenación utilizado en cada uno de los siguientes casos:

Caso 1

```
[25, 8, 5, 1, 7,20]
[ 1,25, 8, 5, 7,20]
[ 1, 5,25, 8, 7,20]
[ 1, 5, 7,25, 8,20]
[ 1, 5, 7, 8,25,20]
[ 1, 5, 7, 8,20,25]
```

Selection sort (2 pts)

Caso 2

```
(parte 1)
[ 8, 3,14, 5, 4, 6]
[ 8, 3,14, 5, 4, 6]
[ 8, 3,14, 5, 4, 6]
[ 8, 3,14, 5, 4, 6]
(parte 2)
[ 3, 8,14, 4, 5, 6]
[ 3, 8,14, 4, 5, 6]
[ 3, 4, 5, 6, 8,14]
```

Merge sort (2 pts)

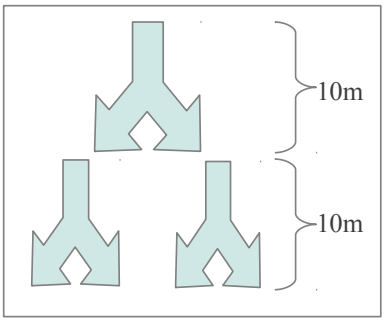
Caso 3

```
(parte 1)
[ 2,13, 5,25, 7,21]
[ 2,13, 5,25, 7,21]
[13, 2, 5,25, 7,21]
[13, 2, 5,25, 7,21]
[25,13, 5, 2, 7,21]
[25,13, 5, 2, 7,21]
[25,13,21, 2, 7, 5]
(parte 2)
[21,13, 5, 2, 7,25]
[13, 7, 5, 2,21,25]
[ 7, 2, 5,13,21,25]
[ 5, 2, 7,13,21,25]
[ 2, 5, 7,13,21,25]
[ 2, 5, 7,13,21,25]
```

Heap sort (2 pts)

2012-1-I3-P2-c–Identificación de solución de dividir para conquistar (1/1)

c) (5 pts) Acantilado submarino



En el fondo del mar hay una cueva que se divide en 4 cuevas más profundas, las cuales a su vez se dividen en 4 cuevas aún más profundas. Por fortuna, las cuevas están numeradas (para evitar que los turistas se pierdan) y siempre se dividen cada 10 metros. Encuentra la cueva más profunda.

Input: Primero el número de bifurcaciones, luego una línea por bifurcación con 5 números i j k l m, indicando que la cueva i se divide en las cuevas j, k, l y m; un valor igual a 0 indica que no hay una cueva abierta para ese lado. Tanto i, j, k, l y m son valores entre 1 y 100.

Ouput: El número de la cueva más profunda y su profundidad desde el lecho marino.

Ejemplo:

Input

```
12
31 3 0 0 0
15 31 0 0 13
13 89 0 8 0
89 0 12 0 0
14 15 11 1 37
11 0 10 0 0
10 0 0 0 27
1 0 7 41 0
7 0 0 0 9
41 0 0 0 17
37 0 0 4 0
4 47 42 48 43
```

Output

```
12 50
```

Solución propuesta

Representar las cuevas como un árbol; a su vez, el árbol lo representamos como un grafo dirigido en una matriz de adyacencia. A medida que leemos los datos, se van anotando las aristas de modo que la matriz [i][j] == 1 solo si desde j se puede ir a i. Una vez leido los datos, la cueva inicial es aquella que no posee aristas incidentes (no llegan aristas, sólo salen aristas).

Con el árbol construido, representado con un grafo, partiendo desde el vértice raiz aplicar recursivamente: encontrar profundidad máxima y vertice respectivo para cada subárbol (hijos de la raiz actual) y quedarse con la profundidad máxima entre éstas, con el vértice respectivo, sumar 10 y retornar; el caso base es un vértice que no tiene hijos, en cuyo caso se retorna la raiz con profundidad 10.

2011-2-I1-P1–HeapSort (1/1)

Estructuras de Datos y Algoritmos – IIC2133

I1 - pauta

31 agosto 2011

1. Supón que tenemos k fuentes de datos. Los datos de cada fuente vienen ordenados, p.ej., de mayor a menor. Queremos enviar la totalidad de los datos recibidos por un único canal de salida, también ordenados de mayor a menor. El dispositivo electrónico que debe hacer esta *mezcla ordenada de datos* tiene una capacidad limitada de memoria.

[**4 pts.**] Da un algoritmo para este dispositivo, que le permita hacer su tarea empleando un arreglo a de k casilleros, en que cada casillero tiene dos campos: en $a[j].data$ se puede almacenar un dato, y en $a[j].num$ se puede almacenar un número entero entre 1 y k. Para recibir un dato desde la fuente i, se ejecuta $receive[i]( )$, y para enviar un dato x por el canal de salida, se ejecuta $send(x)$.

La idea es armar un (max-)heap binario de tamaño k, inicialmente con el primer dato (el mayor) de cada fuente, de modo que en la raíz quede el mayor de todos los datos:

```
for (i = 0; i < k; i = i+1)
    x.data = receive[i]()
    x.num = i
    insertObject(x)
```

A continuación, hay que ir sacando el dato que está en la raíz, enviándolo por el canal de salida, y reemplazándolo en el heap por el próximo dato recibido de la misma fuente de donde provenía el que salió:

```
while ( true )
    x = xMax()
    i = x.num
    send(x.data)
    x.data = receive[i]()
    insertObject(x)
```

[**2 pts.**] Si la cantidad total de datos en las k fuentes es n, ¿cuál es la cantidad total de pasos que ejecuta tu algoritmo, en notación $O( )$?

$O(n \log k)$, ya que el `for` ejecuta k operaciones `insertObject(x)`, c/u de las cuales ejecuta un número de pasos proporcional a $\log k$, lo que da un subtotal de $k \log k$); y el `while` ejecuta $n–k$ operaciones *insertObject(x)*, c/u de las cuales nuevamente ejecuta un número de pasos proporcional a $\log k$, lo que da otro subtotal de $(n–k)\log k$.

2011-2-I2-P1–InsertionSort (1/1)

Estructuras de Datos y Algoritmos – IIC2133

I2 - pauta

5 octubre 2011

1. Sea $a[1 \ldots n]$ un arreglo de n números distintos. Si $j < k$ y $a[j] > a[k]$, entonces el par (j, k) se llama una inversión de a.

a) ¿Cuál es exactamente el máximo número de inversiones que puede tener un arreglo de n números distintos? Justifica.

Este caso se da cuando el arreglo está ordenado justamente "al revés", es decir, de mayor a menor; entonces, hay una inversión por cada par de elementos: (1, 2), (1, 3), …, (1, n), (2, 3), (2, 4), …, (2, n), …, (n–1, n). Así, hay exactamente $(n–1) + (n–2) + \ldots + 1 = n(n–1)/2$ inversiones.

b) Justifica que cualquier algoritmo de ordenación que ordena intercambiando elementos adyacentes —por ejemplo, ordenación por inserción (*insertionSort*)— toma tiempo $\Omega(n^2)$ en promedio para ordenar n elementos.

Al intercambiar dos elementos adyacentes, sólo resolvemos una inversión. Por lo tanto, hay que justificar que un arreglo tiene $\Omega(n^2)$ inversiones en promedio. En realidad, hay $n(n\text{-}1)/4$ inversiones en promedio: tomemos un arreglo $a[1]$, $a[2]$, …, $a[n]$ y su inverso $a[n]$, $a[n\text{-}1]$, …, a[1]; el par (j, k) será una inversión en uno o en el otro; luego, cada dos arreglos, se dan todas las inversiones posibles, es decir, $n(n\text{-}1)/2$ inversiones.

2011-2-I2-P2–QuickSort (1/1)

2. Supón que queremos usar *Quicksort* para ordenar un arreglo con muchas claves duplicadas; por ejemplo, ordenar el archivo de alumnos de la universidad por año de nacimiento. En este caso, el desempeño de *Quicksort* puede mejorarse bastante; por ejemplo, un subarreglo que tiene sólo datos iguales (todos con la misma clave) no necesita seguir siendo procesado, pero *Quicksort* va a seguir particionándolo en subarreglos cada vez más pequeños.

Una idea es particionar el arreglo en tres partes: las claves que son menores que el pivote, las que son iguales al pivote, y las que son mayores que el pivote.

a) [**4 pts.**] Da un algoritmo, llamado *Partition3*, de complejidad $O(n)$ —es decir, equivalente al *Partition* estudiado en clase— para realizar esta nueva partición en tres (sobre un subarreglo de n datos).

La idea es que se hace una pasada por el arreglo $a[e \,..\, w]$ de izquierda a derecha que mantiene tres punteros:
- un puntero lt tal que $a[e \,..\, lt\text{-}1]$ es menor que el pivote
- un puntero gt tal que $a[gt\text{+}1 \,...\, w]$ es mayor que el pivote
- un puntero k tal que $a[lt \,...\, k\text{-}1]$ es igual al pivote
- $a[k \,...\, gt]$ aun no han sido examinados

> ***Partition3***:
> Se elige el pivote $v = a[e]$, y los punteros $lt = e$, $gt = w$
> Recorremos el arreglo con k, a partir de $k = e\text{+}1$ y hasta $k = gt$:
> - si $a[k] < v$, intercambiamos $a[lt]$ con $a[k]$; incrementamos lt y k
> - si $a[k] > v$, intercambiamos $a[k]$ con $a[gt]$; decrementamos gt
> - si $a[k] = v$, incrementamos k

b) [**2 pts.**] Escribe la nueva versión de *Quicksort*, que hace uso de *Partition3*.

```
Quicksort(a, e, w):
if ( e < w )
    —aquí va el código de Partition3
    Quicksort(a, e, lt-1)
    Quicksort(a, gt+1, w)
```

2010-2-I2-P2–Estabilidad de algoritmos de ordenación (1/1)

2. Con respecto a los siguientes algoritmos de ordenación, (**a**) ¿cuáles son estables y cómo se sabe que lo son? Para los que no lo son, (**b**) explica cómo se los puede hacer estables y a qué costo. Recuerda que un algoritmo de ordenación es ***estable*** si los datos con igual valor aparecen en el resultado en el mismo orden que tenían al comienzo.

i) Insertionsort. ii) Mergesort. iii) Heapsort. iv) Quicksort.

Respuesta:

a) [**3 pts.**]
[**1 pt.**] *Insertionsort* y *Mergesort* son estables.
[**1 pt.**] *Insertionsort* (diapositiva #19 de los apuntes) es estable porque al comparar las claves de b y $a[j\text{-}1]$, "subimos" (o avanzamos hacia $a[0]$) solo si la clave de b es estrictamente menor que la de $a[j\text{-}1]$.
[**1 pt.**] *Mergesort* (diapositiva # 26 de los apuntes) es estable porque al comparar el $a[p]$, de la primera "mitad", con el $a[q]$, de la segunda mitad, colocamos $a[p]$ en el resultado temporal (tmp) solo si su clave es estrictamente menor que la de $a[q]$.

b) [**3 pts.**]
Heapsort y *Quicksort* no son estables.
[**2 pts.**] Una forma de hacerlos estables es guardar el índice de cada elemento (la ubicación del elemento al comienzo) junto con el elemento. Así, cuando comparamos dos elementos, los comparamos según sus valores (*key*) y, si son iguales, usamos los índices para decidir.
[**1 pt.**] Por cada elemento, se necesita almacenar adicionalmente su índice. Si los índices van entre 0 y n, cada uno se puede almacenar en $\log n$ bits. Así, en total, se necesita $n \log n$ espacio adicional.

2010-2-I2-P4–QuickSort (1/1)

4. Escribe una versión no recursiva (y, por lo tanto, iterativa) de *Quicksort*, que use la versión de *partition* estudiada en clase. Supón que dispones de un stack de números enteros con las operaciones *push*, *pop* y *empty* habituales. La idea es que después de llamar a *partition*, en lugar de llamar recursivamente a *Quicksort* dos veces, coloques en el stack los índices correspondientes a cada una de las dos partes. Así, tu versión iterativa debe iterar mientras el stack no esté vacío.

Respuesta:

```
void iterQuicksort(int[] a)
{
   Stack s = new Stack();
   s.push(0); s.push(a.length-1);     [1 pt.] Iniciar el stack con los índices extremos del arreglo original.
   int e, m, w;
   while (!s.empty())
   {
      w = s.pop(); e = s.pop();       [2 pts.] Sacar del stack los índices extrremos de la parte a procesar.
      if (e < w)                      [1 pt.] Verificar si la parte tiene al menos dos elementos.
      {
         m = partition(a, e, w);
         s.push(e); s.push(m-1);      [1 pt.] Colocar en el satck los índices extremos de la parte izquierda.
         s.push(m+1); s.push(w);      [1 pt.] Colocar en el stack los índices extremos de la parte derecha.
      }
   }
}
```

2010-2-Ex-P2–Estabilidad de algoritmos de ordenación (misma pregunta que 2010-2-I2-P2) (1/1)

2) Con respecto a los siguientes algoritmos de ordenación, (**a**) ¿cuáles son estables y cómo se sabe que lo son? Para los que no lo son, (**b**) explica cómo se los puede hacer estables y a qué costo. Recuerda que un algoritmo de ordenación es ***estable*** si los datos con igual valor aparecen en el resultado en el mismo orden que tenían al comienzo.

i) Insertionsort. ii) Mergesort. iii) Heapsort. iv) Quicksort.

2 Árboles

En esta sección se encuentran los siguientes contenidos:

- Heap
- ABB
- AVL
- Árbol rojo-negro
- Árbol 2-3, 3-4

2020-2-I1-P2–ABB, Heap (1/2)

Pregunta 2

a) Escribe un algoritmo $\mathcal{O}(n)$ para convertir un ABB de n claves en un min-Heap. Demuestra que es correcto.

b) ¿Por qué encontrar el elemento de menor prioridad en un Heap de n elementos requiere revisar solo $\frac{n}{2}$ elementos?

Solución Pregunta 2a)

Recordemos que un ABB tiene un orden total de los datos, por lo que podemos obtener las claves ordenadas haciendo un recorrido *in-order*, el cual es $\mathcal{O}(n)$. En clases se discutió como hacer un recorrido *in-order* del árbol para imprimir las claves en orden, de la siguiente manera:

```
1: procedure InOrder(árbol T)
2:     if T es un árbol then
3:         InOrder(T.left)
4:         imprimir T.key
5:         InOrder(T.right)
6:     end if
7: end procedure
```

Necesitamos modificar esta función para que en lugar de imprimir las claves en orden, nos entregue una lista ligada o arreglo con las claves en orden. Lo más simple es hacerlo sobre una lista:

```
1: procedure InOrder(árbol T, lista ligada L)
2:     if T es un árbol then
3:         InOrder(T.left)
4:         agregar (T.key, T.value) al final de L
5:         InOrder(T.right)
6:     end if
7: end procedure
```

Tras llamar a InOrder(T, L), tenemos en la lista L todos los pares $(key, value)$ de T, ordenados de menor a mayor clave. Esta lista podemos trivialmente convertirla a un arreglo A en $\mathcal{O}(n)$.

Notar que A está ordenado de menor a mayor, por lo que cumple las propiedades de *min-Heap*. Esto es, para cada elemento en posición α se tiene que $A[\alpha] \leq A[2\alpha + 1]$ y $A[\alpha] \leq A[2\alpha + 2]$. Por lo tanto A es el heap que buscamos (almacenado como un arreglo)

Finalmente, se tiene que el algoritmo es correcto puesto a que utiliza dos procesos finitos de $O(n)$ pasos y entrega el output deseado, ya que el array A cumple las propiedades de *Heap*.

[1pt] Por poner los datos del árbol ordenadamente en una lista o arreglo

[1pt] Por argumentar que un arreglo ordenado cumple con la propiedad de Heap

[1pt] Por demostrar que el algoritmo propuesto es correcto

Si el algoritmo propuesto no es $\mathcal{O}(n)$ entonces el puntaje máximo a obtener es 1pt

2020-2-I1-P2–ABB, Heap (2/2)

Solución Pregunta 2b)

El elemento de menor prioridad en un heap no puede tener hijos, ya que de tenerlos, sus prioridades deberían ser aun menores.

Por lo tanto, si el elemento de menor prioridad está en una de las hojas del heap, pero no sabemos cual.

Eso significa que para encontrarlo, sería necesario revisar cada hoja.

Por lo tanto, necesitamos saber cuantas hojas tiene.

Haciendo las inserciones por nivel (para que el heap sea lo más balanceado posible), tenemos que la cantidad de hojas en función de n es:

$h(1) = 1$, donde la raíz es hoja.

$h(2) = 1$, donde la nueva hoja cuelga de una hoja con $n = 1$

$h(3) = 2$, donde la nueva hoja es ahora hermana de la hoja que apareció en $n = 2$

$h(4) = 2$, donde la nueva hoja cuelga de una hoja con $n = 3$

$h(5) = 3$, donde la nueva hoja es ahora hermana de la hoja que apareció en $n = 4$

Esto sigue así:

$h(6) = 3$

$h(7) = 4$

$h(8) = 4$

$h(9) = 5$

Por lo que $h = \lceil \frac{n}{2} \rceil$.

[**1.25pt**] Por identificar que la clave de menor prioridad es una hoja

[**1.25pt**] Por contar la cantidad de hojas en función de n

[**0.5pt**] Por la justificación

2020-2-I1-P3–ABB, ABB-balanceado (1/2)

Pregunta 3

a) Sea $\boldsymbol{T}$ un ABB de altura $\boldsymbol{h}$. Escribe un algoritmo que posicione un elemento arbitrario de $\boldsymbol{T}$ como raíz de $\boldsymbol{T}$ en $\boldsymbol{\mathcal{O}(h)}$ pasos.

b) Sean $\boldsymbol{T_1}$ y $\boldsymbol{T_2}$ dos ABB de $\boldsymbol{n}$ y $\boldsymbol{m}$ nodos respectivamente. Explica, de manera clara y precisa, cómo realizar un merge entre ambos árboles en $\boldsymbol{\mathcal{O}(n+m)}$ pasos para dejarlos como un solo ABB balanceado $\boldsymbol{T}$ con los nodos de ambos árboles.

Solución Pregunta 3a)

La solución puede o no considerar el proceso de búsqueda del nodo solicitado (de ahora en adelante, $\boldsymbol{y}$). Este paso tiene complejidad $\boldsymbol{\mathcal{O}(h)}$, por lo que sigue dentro de la cota especificada.

Tras este proceso, es necesario hacer rotaciones de tal forma que $\boldsymbol{y}$ quede en la raíz del árbol. Esto se puede realizar utilizando las rotaciones AVL vistas en clases. A continuación se propone un pseudocódigo.

```
1:  procedure VolverRaiz(nodo y)
2:      while y tiene padre do
3:          x ← y.padre
4:          if y es hijo izquierdo de su padre then
5:              rotar hacia la derecha en torno a x-y        ▷ esto modifica el padre de y
6:          else
7:              rotar hacia la izquierda en torno a x-y      ▷ esto modifica el padre de y
8:          end if
9:      end while
10: end procedure
```

Vimos en clases que cada rotación es $\boldsymbol{\mathcal{O}(1)}$ y en cada una de estas $\boldsymbol{y}$ sube un nivel. En el peor caso $\boldsymbol{y}$ es una hoja por lo tanto es necesario hacerlo subir $\boldsymbol{h}$ niveles $\boldsymbol{\implies}$ se hacen $\boldsymbol{\mathcal{O}(h)}$ rotaciones.

[1pt] Por identificar que una rotación permite hacer subir el nodo $\boldsymbol{y}$ en un nivel.

[1pt] Por escribir el algoritmo correctamente.

[1pt] Por justificar la complejidad, indicando que cada rotación es $\boldsymbol{\mathcal{O}(1)}$.

Si el algoritmo propuesto no es $\boldsymbol{\mathcal{O}(h)}$ entonces el puntaje máximo a obtener es 1pt

2020-2-I1-P3–ABB, ABB-balanceado (2/2)

Solución Pregunta 3b)

La pregunta no solicita un algoritmo, por lo que basta con describir el proceso:

1. Se itera por sobre los nodos de ambos árboles de manera ordenada, copiando los nodos a un array. Este paso consiste en un proceso recursivo que visita siempre el nodo izquierdo antes que el derecho (recorrido *in-order* o algoritmo visto en la ayudantía, ver solución pregunta 2a). Haciendo esto tenemos dos arreglos ordenados, con los elementos de $\boldsymbol{T_1}$ y $\boldsymbol{T_2}$ respectivamente.
2. Combinamos estos dos arreglos usando la subrutina ***merge*** de MergeSort en $\boldsymbol{\mathcal{O}(n+m)}$. Con esto obtenemos un arreglo ordenado $\boldsymbol{A}$ con los $\boldsymbol{n+m}$ elementos de ambos árboles.
3. Finalmente, se convierte el array ordenado $\boldsymbol{A}$ en un ABB balanceado poniendo la mediana como raiz e insertando los valores menores y mayores a esta en las ramas izquierdas y derechas respectivamente. Notar que encontrar la mediana en un array ordenado es $\boldsymbol{O(1)}$ por lo que proceso se realiza en $\boldsymbol{O(m+n)}$ pasos.

La solución consta de tres pasos $\boldsymbol{O(m+n)}$, por lo que la rutina completa es $\boldsymbol{O(m+n)}$.

[1pt] Por explicar como obtener el arreglo ordenado con todos los datos

[1pt] Por explicar como crear un árbol balanceado a partir de un arreglo ordenado

[1pt] Por justificar la complejidad del proceso

Si el proceso propuesto no es $\boldsymbol{\mathcal{O}(n+m)}$ entonces el puntaje máximo a obtener es 1pt

2020-2-I1-P4–AVL, Árbol negro-rojo (1/9)

Pregunta 4

a) Cualquier árbol AVL es también un árbol Rojo-Negro; solo hace falta pintar adecuadamente los nodos del AVL de color rojo o negro. Explica cómo decidir de qué color pintar cada nodo de un árbol AVL para que el árbol resultante sea un árbol Rojo-Negro (válido).

b) Calcula la profundidad a la que se encuentra la hoja menos profunda en un AVL de altura $\boldsymbol{h}$ lo más desbalanceado posible. ¿Cómo se relaciona con la profundidad a la que se encuentra la hoja menos profunda en un árbol Rojo-Negro de altura $\boldsymbol{h}$, también lo más desbalanceado posible? ¿Para qué valores de $\boldsymbol{h}$ estas dos profundidades son iguales?

Solución Pregunta 4a)

Para poder pintar un árbol AVL de rojo negro, primero debemos recordar las siguientes propiedades:

- En un árbol AVL, las alturas de dos sub-árboles hermanos difieren en 0 o 1.
- En un árbol Rojo-Negro, la ruta de la raíz a cada hoja debe contener la misma cantidad de nodos negros. En particular, dos sub-árboles hermanos deben tener la misma cantidad de nodos negros camino a cada hoja.

Sabiendo esto, debemos preguntarnos como relacionar la altura de los hermanos con la cantidad de nodos negros camino a sus hojas.

Para un sub-árbol de altura $\boldsymbol{h}$, denotaremos $\boldsymbol{n}$ como la cantidad de nodos negros camino a cada hoja. Veamos como se relacionan $\boldsymbol{h}$ y $\boldsymbol{n}$.

El mínimo valor de $\boldsymbol{n}$ para un $\boldsymbol{h}$ dado está dictado por su rama más larga, la cual tiene altura $\boldsymbol{h}$. Recordemos que cada nodo que no es negro debe ser rojo, y que por definición de árbol rojo-negro, en una rama no puede haber dos nodos rojos seguidos. Por lo que si el árbol tuviera menos nodos negros, obligaríamos a que la rama más larga tenga dos nodos rojos seguidos, lo que viola la propiedad de árbol rojo negro.

Si $\boldsymbol{h}$ es par, tenemos que la cantidad de nodos negros debe ser igual a la cantidad de nodos rojos.

$$\begin{aligned} \boldsymbol{h} &= \boldsymbol{2n} \\ \boldsymbol{min(n)} &= \frac{\boldsymbol{h}}{\boldsymbol{2}} \end{aligned}$$

Si $\boldsymbol{h}$ es impar, entonces tenemos el caso en que **la raíz es roja**[1] y la última hoja es roja, y luego intercalamos nodos rojos con negros. Por lo tanto:

$$\begin{aligned} \boldsymbol{h} &= \boldsymbol{2n+1} \\ \boldsymbol{2n} &= \boldsymbol{h-1} \\ \boldsymbol{min(n)} &= \frac{\boldsymbol{h}}{\boldsymbol{2}} - \frac{\boldsymbol{1}}{\boldsymbol{2}} \end{aligned}$$

[1] Recordar que esto es un sub-árbol, por lo que la raiz no necesariamente es negra

2020-2-I1-P4–AVL, Árbol negro-rojo (2/9)

Ahora, el máximo de $\boldsymbol{n}$ para un $\boldsymbol{h}$ dado está dictado por su rama más corta posible, ya que si fuera mayor no sería posible cumplir la propiedad de árbol rojo negro en caso de existir una rama así en el árbol.

En la pregunta 4b calculamos la altura de la rama más corta posible:

Para $\boldsymbol{h}$ par,

$$\boldsymbol{max(n) = \frac{h}{2}}$$

Y para $\boldsymbol{h}$ impar, sabemos que **la raíz debe ser negra**, por lo que

$$\boldsymbol{max(n) = \frac{h}{2} + \frac{1}{2}}$$

Por lo tanto, tenemos el $\boldsymbol{n}$ que debe tener un árbol rojo negro para permitir todos los casos posibles de estructura interna, para $\boldsymbol{h}$ par,

$$\boldsymbol{n = \frac{h}{2}}$$

Y para $\boldsymbol{h}$ impar,

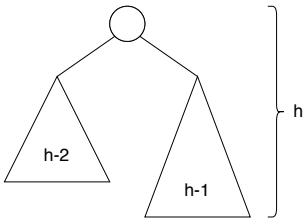
$$\boldsymbol{n = \frac{h}{2} \pm \frac{1}{2}}$$

Recordemos que un árbol AVL de altura $\boldsymbol{h}$ puede tener dos casos:

1. Ambos hijos son de igual altura, $\boldsymbol{h - 1}$
2. Ambos hijos son de distinta altura, $\boldsymbol{h - 1}$ y $\boldsymbol{h - 2}$

Ahora debemos combinar esos casos con la paridad de las alturas.

Caso 1: Distinta altura, $\boldsymbol{h - 1}$ par



Ambos hermanos deben tener igual $\boldsymbol{n}$.

2020-2-I1-P4–AVL, Árbol negro-rojo (3/9)

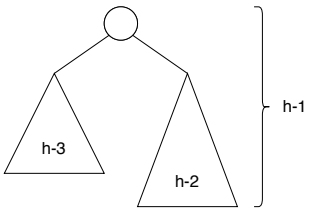
El $\boldsymbol{n}$ del hermano mayor, como $\boldsymbol{h-1}$ es par, es $\frac{\boldsymbol{h-1}}{\boldsymbol{2}}$

El $\boldsymbol{n}$ del hermano menor, como $\boldsymbol{h-2}$ es impar, es $\frac{\boldsymbol{h-2}}{\boldsymbol{2}} \pm \frac{\boldsymbol{1}}{\boldsymbol{2}}$

Ya que ambos deben ser iguales, entonces el hermano menor debe ser $\frac{\boldsymbol{h-2}}{\boldsymbol{2}} + \frac{\boldsymbol{1}}{\boldsymbol{2}}$, el cual es el caso en que la raíz de ese árbol es negra.

Por lo tanto, la raíz del hermano menor debe ser negra.

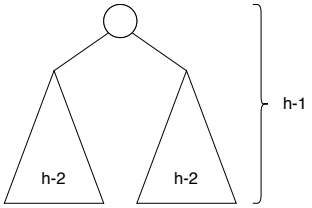
¿Qué hay del hermano mayor? Veamos sus hijos, los cuales pueden ser uno de los dos casos de un AVL. En primer lugar, cuando sus hijos son de distinta altura



Estos hermanos a su vez deben tener el mismo $\boldsymbol{n}$, el cual está dictado por el hermano de altura par $\boldsymbol{h-3}$. Este $\boldsymbol{n}$ es entonces $\frac{\boldsymbol{h-3}}{\boldsymbol{2}}$, el cual es exactamente el $\boldsymbol{n}$ de su padre, $\boldsymbol{-1}$.

Por lo tanto en este caso la raíz de este sub-árbol debe ser negra.

Cuando los hijos de este sub-arbol son de la misma altura, tenemos:



En este caso, ambos árboles son de altura impar, por lo que su $\boldsymbol{n}$ podría o no ser igual al del padre.

Cabe destacar que no existe una única manera de pintar un AVL a Rojo-Negro. En este caso, vamos a preferir el caso en que el $\boldsymbol{n}$ es lo menor posible[2], por lo que el $\boldsymbol{n}$ de estos hijos sería $\frac{\boldsymbol{n-2}}{\boldsymbol{2}} - \frac{\boldsymbol{1}}{\boldsymbol{2}}$. Este $\boldsymbol{n}$ es entonces $\frac{\boldsymbol{h-3}}{\boldsymbol{2}}$, el cual es exactamente el $\boldsymbol{n}$ de su padre, $\boldsymbol{-1}$.

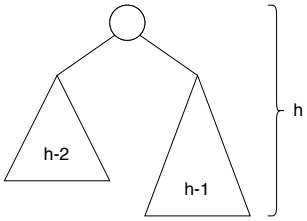
Por lo tanto en este caso la raíz de este sub-árbol debe ser negra.

Por lo tanto, la raíz del hermano mayor debe ser negra, independiente de como sean sus hijos

[2]Para así minimizar la altura del 2-4 correspondiente

2020-2-I1-P4–AVL, Árbol negro-rojo (4/9)

Caso 2: Distinta altura, $h-1$ impar



Ambos hermanos deben tener igual n.

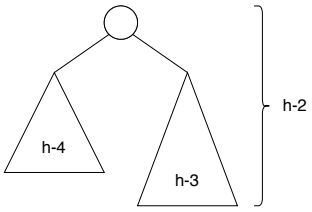
El n del hermano mayor, como $h-1$ es impar, es $\frac{h-1}{2} \pm \frac{1}{2}$

El n del hermano menor, como $h-2$ es par, es $\frac{h-2}{2}$

Ya que ambos deben ser iguales, entonces el hermano mayor debe ser $\frac{h-2}{2} - \frac{1}{2}$, el cual es el caso en que la raíz de ese árbol es roja.

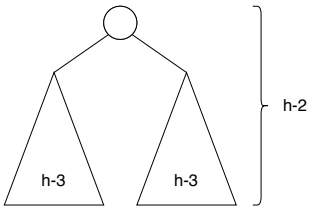
Por lo tanto, la raíz del hermano mayor debe ser roja

Para el hermano menor es necesario hacer el mismo análisis de sus hijos que hicimos para el caso 1.



El n de estos hermanos está determinado por el hermano par, $h-4$, el cual sería $\frac{h-4}{2}$, el cual es exactamente 1 menos que el n de su padre.

En este caso entonces pintaríamos la raíz del hermano menor de negro.



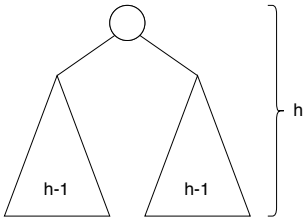
El n de estos hermanos es $\frac{h-3}{2} \pm \frac{1}{2}$, ya que $h-3$ es impar. Nuevamente, elegimos el mínimo n posible y nos quedamos con $\frac{h-3}{2} - \frac{1}{2}$. Esto es exactamente 1 menos que el n de su padre.

2020-2-I1-P4–AVL, Árbol negro-rojo (5/9)

En este caso entonces también pintaríamos la raíz del hermano menor de negro.

Por lo tanto, la raíz del hermano menor debe ser negra, independiente de como sean sus hijos

Caso 3: Misma altura, $h-1$ par



Ambos hermanos deben tener igual $\boldsymbol{n}$.

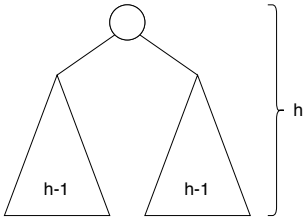
El $\boldsymbol{n}$ de ambos hermanos, como $\boldsymbol{h-1}$ es par, es $\frac{\boldsymbol{h-1}}{\boldsymbol{2}}$

Para determinar de que color pintar sus raíces, miramos a sus hijos.

Este exactamente el mismo análisis que para el hermano mayor del caso 1.

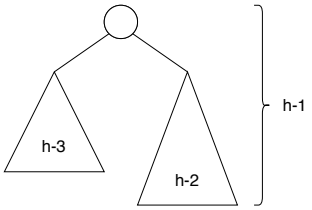
Por lo tanto, la raíz de ambos hermanos debe ser negra, independiente de como sean sus hijos

Caso 4: Misma altura, $h-1$ impar



El $\boldsymbol{n}$ de ambos hermanos, como $\boldsymbol{h-1}$ es par, es $\frac{\boldsymbol{h-1}}{\boldsymbol{2}} \pm \frac{\boldsymbol{1}}{\boldsymbol{2}}$

Veamos que pasa con sus hijos.

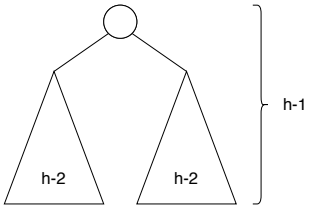


2020-2-I1-P4–AVL, Árbol negro-rojo (6/9)

El n de estos hermanos está determinado por el hermano de altura par, $h-2$, por lo que sería $\frac{h-2}{2}$

Esto podría ser menor o igual que el n de su padre. Nuevamente, vamos a preferir que todo n sea lo menor posible, por lo que ambos serían iguales.

Por lo tanto, en este caso las raíces de ambos hermanos en el primer árbol serían rojas.



Tenemos el mismo caso anterior: el n es $\frac{h-2}{2}$.

Siguiendo la misma lógica, llegamos entonces a la misma conclusión

Las raíces de ambos hermanos deben ser rojas, independiente de como sean sus hijos

[**1.5pt**] Por explicar ambos casos en que ambos hijos son de la misma altura.

[**1.5pt**] Por explicar ambos casos en que ambos hijos son de distinta altura

[**No es necesario que la explicación tenga este nivel de detalle**]

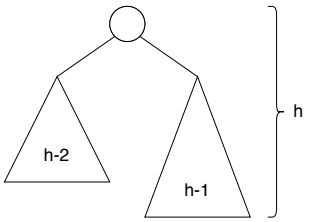
2020-2-I1-P4–AVL, Árbol negro-rojo (7/9)

Solución Pregunta 4b)

Árbol AVL

La propiedad fundamental de un AVL nos dice que, para dos árboles hermanos, sus alturas no difieren en más de 1.

Definimos en clases el AVL lo más desbalanceado posible como el árbol en el que **cada** par de árboles hermanos difiere en 1 de altura. En particular, un árbol de altura $\boldsymbol{h}$ se ve como sigue:



La hoja menos profunda se encuentra en el sub-arbol de menor altura.

Podemos expresar la profundidad $\boldsymbol{p}$ a la que se encuentra la hoja menos profunda en funcion de $\boldsymbol{h}$ como sigue:

$$\boldsymbol{p(h) = 1 + p(h-2)}$$

Ya que la hoja menos profunda está en el sub-árbol de menor altura, y este sub-arbol está un nivel más abajo que el árbol actual. Si expandimos esto nos damos cuenta que:

$$\begin{aligned} \boldsymbol{p(h)} &= \boldsymbol{1 + p(h-2)} \\ &= \boldsymbol{2 + p(h-4)} \\ &= \boldsymbol{3 + p(h-6)} \\ &\boldsymbol{\cdots} \\ &= \boldsymbol{k + p(h-2k)} \end{aligned}$$

Donde basta con ver un AVL de altura 1 y altura 2 para ver que $\boldsymbol{p(1) = 1}$ y $\boldsymbol{p(2) = 2}$.

Por lo tanto, en el caso que $\boldsymbol{h}$ sea impar, la recursión termina con $\boldsymbol{p(1)}$, y por lo tanto

$$\begin{aligned} \boldsymbol{h - 2k} &= \boldsymbol{1} \\ \boldsymbol{2k} &= \boldsymbol{h - 1} \\ \boldsymbol{k} &= \boldsymbol{\frac{h}{2} - \frac{1}{2}} \end{aligned}$$

2020-2-I1-P4–AVL, Árbol negro-rojo (8/9)

Reemplazando en $p(h)$:

$$\begin{aligned} p(h) &= k + p(h - 2k) \\ &= \frac{h}{2} - \frac{1}{2} + p(1) \\ &= \frac{h}{2} - \frac{1}{2} + 1 \\ p(h) &= \frac{h}{2} + \frac{1}{2} \end{aligned}$$

Para el caso de h par la recursión termina con $p(2)$, por lo que:

$$\begin{aligned} h - 2k &= 2 \\ 2k &= h - 2 \\ k &= \frac{h}{2} - 1 \end{aligned}$$

Reemplazando en $p(h)$:

$$\begin{aligned} p(h) &= k + p(h - 2k) \\ &= \frac{h}{2} - 1 + p(1) \\ &= \frac{h}{2} - 1 + 2 \\ p(h) &= \frac{h}{2} + 1 \end{aligned}$$

Árbol Rojo-Negro

La propiedad de balance de un árbol rojo-negro está dictada por que la cantidad de nodos negros camino a una hoja debe ser la misma para todas las hojas.

En este caso, un árbol Rojo-Negro lo más desbalanceado posible es un árbol que tiene hojas cuya ruta no tiene nodos rojos, mientras que además tiene hojas cuya ruta tiene la máxima cantidad posible de nodos rojos.

Sea n la cantidad de nodos negros para cualquier rama, y r la cantidad de nodos rojos de la rama más larga. Por definición, la hoja menos profunda está a profundidad n.

$$h = n + r$$

Sabemos que un nodo rojo no puede tener hijos rojos, por lo que $r \leq n$.

2020-2-I1-P4–AVL, Árbol negro-rojo (9/9)

El máximo valor de r es n, el cual solo puede suceder cuando h es par.

Por lo tanto,

$$h = 2n$$
$$n(h) = \frac{h}{2}$$

En el caso de h impar el máximo valor que puede tomar r es $r = n - 1$.

Por lo tanto, $h = 2n - 1$

$$h = 2n - 1$$
$$2n = h + 1$$
$$n(h) = \frac{h}{2} + \frac{1}{2}$$

Podemos ver que para h par, $p(h) > n(h)$, y que para h impar, $p(h) = n(h)$

Por lo tanto, $p(h) \geq n(h)$, es decir, la profundidad de la hoja menos profunda en un AVL es mayor o igual a la profundidad de hoja menos profunda en un árbol Rojo-Negro.

[**1.25pt**] Por el cálculo de la hoja menos profunda en el árbol AVL.

(Sólo [**1pt**] si calcula uno solo de los dos casos, solo [**0.5**] si no calcula)

[**1.25pt**] Por el cálculo de la hoja menos profunda en el árbol Rojo-Negro

(Sólo [**1pt**] si calcula uno solo de los dos casos, solo [**0.85**] si no calcula. No hay descuento por no calcular si cita la solución ayudantía.)

[**0.5pt**] Por la comparación entre ambos.

2020-1-I2-P1–ABB (1/4)

IIC2133 – Estructuras de Datos y Algoritmos
Interrogación 2

Hora inicio: 14:00 del 6 de mayo del 2020

Hora máxima de entrega: 23:59 del 7 de mayo del 2020

Rellena lo siguiente e inclúyelo al principio de tu entrega. Nos reservamos el derecho a no corregir tu prueba si no lo haces:

Yo, *Nombre Apellido* , doy fe de que todas las respuestas contenidas en esta prueba fueron elaboradas por mí, sin haber consultado sobre la prueba a ninguna persona ajena al cuerpo docente del curso.

Firma

1. La propiedad fundamental de un ABB es que las claves almacenadas en el subárbol izquierdo son todas menores que la clave almacenada en la raíz, la que a su vez es menor que cualquiera de las claves almacenadas en el subárbol derecho. Teniendo presente esta propiedad, responde:

   a. Considera que tienes un ABB vacío T sin autobalance y una lista desordenada L de n números. ¿Cómo puedes utilizar T para ordenar L? ¿Cuál sería la complejidad de este algoritmo en notación Ω? ¿Qué características tiene L cuando se da este caso? Da un ejemplo con $n = 11$.

   b. Definimos B_x como los nodos en la ruta de búsqueda de una hoja x en un árbol T. Definimos A_x como todos los nodos a la izquierda de B_x, y C_x como todos los nodos a la derecha de B_x. ¿Es posible que haya un nodo en C_x de clave menor a la clave de un nodo en A_x?
   Si la respuesta es sí, da un ejemplo. En caso contrario, demuestra que no es posible.

2020-1-I2-P1–ABB (2/4)

Pontificia Universidad Católica de Chile
Escuela de Ingeniería
Departamento de Computación
IIC2133 – Estructuras de datos y algoritmos
Primer Semestre 2020

Pauta Interrogación 2

Problema 1

a)

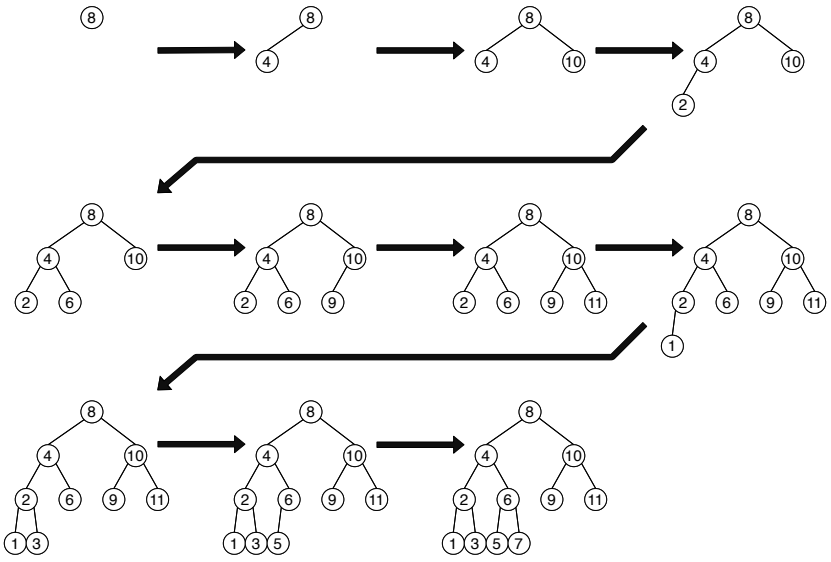
La estrategia es tomar todos los elementos de L e insertarlos en T. Luego podemos obtener los elementos de L en orden haciendo un recorrido *in-order* del árbol. [**0.1 pts**]

Este algoritmo es $\Omega(n \cdot log(n))$, ya que son n inserciones y en el **mejor caso** cada una toma $\mathcal{O}(log(n))$, mientras que el recorrido *in-order* es $\Theta(n)$. [**0.2 pts**]

Para ver cuando se da este caso, tenemos que pensar en un ABB perfectamente balanceado T'. Para que una lista L genere el árbol T', se debe cumplir que cada número aparezca en L después que todos sus ancestros en T'. [**0.5 pts**]. Si solo incluye un caso particular como el de las medianas, [**0.3 pts**]

Un ejemplo sería:

$$L = [8,\ 4,\ 10,\ 2,\ 6,\ 9,\ 11,\ 1,\ 3,\ 5,\ 7]$$



[**0.2 pts**] por mostrar como queda el árbol luego de insertar todos los elementos de L. Basta con mostrar solo el árbol final. Debe cumplir con la propiedad dicha anteriormente.

2020-1-I2-P1–ABB (3/4)

b)

Primero que todo, la respuesta correcta a la pregunta es, **no**, no puede haber un elemento en A_x mayor a un elemento en C_x. **[0.1 pts]**

Posible demostración [0.9 pts]:

Para demostrar esto, es útil expresar los grupos de nodos en términos de conjuntos:

$$B_x = \{b \mid b \text{ está en la ruta de búsqueda hacia } x\}$$

$$A_x = \{a \mid a \notin B_x \wedge (a \text{ es hijo izquierdo de un nodo en } B_x \text{ o es descendiente de otro elemento en } A_x)\}$$

$$C_x = \{c \mid c \notin B_x \wedge (c \text{ es hijo derecho de un nodo en } B_x \text{ o es descendiente de otro elemento en } C_x)\}$$

Veamos como se relacionan los elementos de A_x y C_x respecto a x.

Para $a \in A_x$, tenemos 2 casos:

- a es hijo izquierdo de un nodo en B_x: Llamemos a dicho nodo "b". Por definición de ABB, a es menor a todo el subárbol derecho de b. Pero por definición de B_x, b forma parte de la ruta hacia x, y como $a \notin B_x$, x tiene que estar en el subárbol derecho de b. Por lo tanto, $a < x$.
- a es descendiente de otro elemento en A_x: Por definición de ABB, si un nodo dado es menor a un número, todos sus descendientes también cumplen con esa desigualdad, sin importar la profundidad. Todos los a que surgen de este caso son descendientes de los a del primer caso, por lo que también se cumple que $a < x$

Es decir,

$$\forall a \in A_x,\ a < x$$

Análogamente,

$$\forall c \in C_x,\ x < c$$

Por lo tanto,

$$\forall a \in A_x,\ \forall c \in C_x,\ a < c$$

Escala en caso de hacer una demostración distinta

En caso de poner una demostración distinta, se hizo la siguiente escala para cada caso.

- 0 pts si la demostración no tiene lógica, o no hay formalidad en ella y es incorrecta.
- 0.3 pts si la demostración tiene lógica, pero tiene errores graves de formalidad o planteamientos en la lógica.
- 0.6 pts si la demostración tiene lógica, pero tiene errores pequeños en la formalidad o errores logicos en las ideas planteadas.
- 0.9 pts si la demotración es correcta, tiene la formalidad requerida y demuestra correctamente lo pedido.

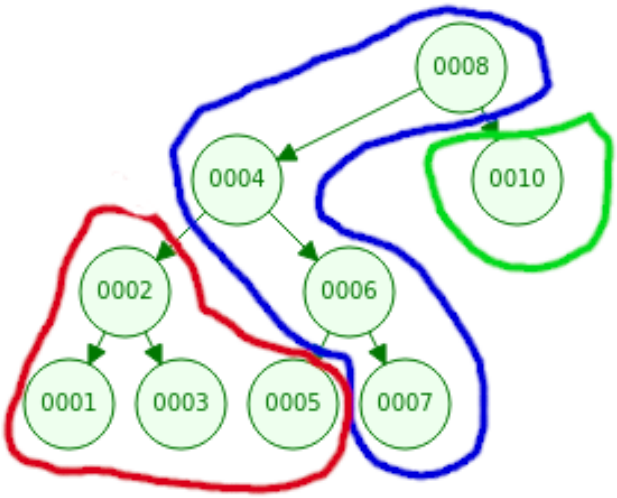
2020-1-I2-P1–ABB (4/4)

Contra Ejemplo

En varios casos se enuncio que

$$\forall a \in A_x, b \in B_x \quad a < b$$

Lo cual es incorrecto, como contraejemplo aca hay un arbol que lo prueba, en el cual x = 7. En azul Bx, rojo Ax y verde Cx



2020-1-I2-P2–AVL (1/3)

2. La propiedad fundamental de un AVL, además de ser un ABB, es que las alturas del subárbol izquierdo y del subárbol derecho difieren a lo más en 1. Teniendo presente esta propiedad, responde:

    a. Considera que tienes un AVL vacío T y una lista desordenada L de n números.
    ¿Cómo puedes utilizar T para ordenar L? ¿Cuál sería la complejidad de este algoritmo en notación Ω? ¿Qué características tiene L cuando se da este caso? Ejecuta tu algoritmo con $L = [17, 29, 53, 61, 73, 37, 43]$.

    b. Demuestra que basta con hacer una sola rotación (simple o doble) para corregir el desbalance producto de una inserción en un árbol AVL.

2020-1-I2-P2–AVL (2/3)

Problema 2

a)

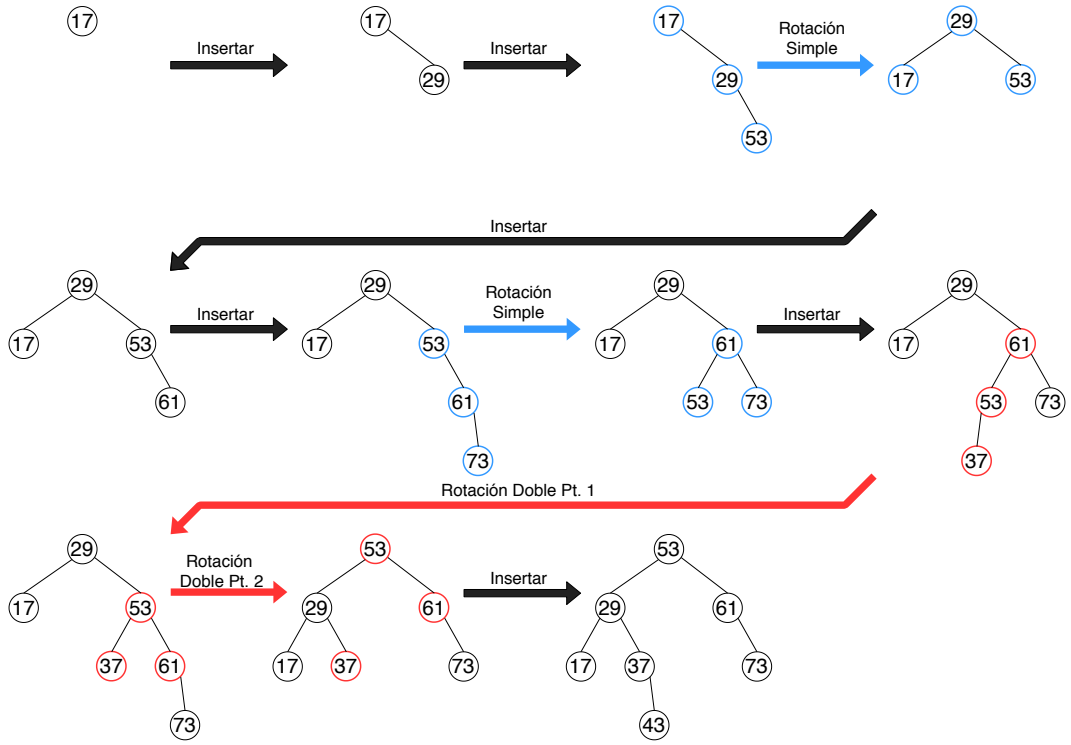
La estrategia es tomar todos los elementos de L e insertarlos en T. Luego podemos obtener los elementos de L en orden haciendo un recorrido *in-order* del árbol. **[0.1 pts]**

Este algoritmo es $\Omega(n \cdot log(n))$, ya que son n inserciones y cada una toma $\Theta(log(n))$, mientras que el recorrido *in-order* es $\Theta(n)$. **[0.2 pts]**

Este caso se da independiente de las características de L. **[0.2 pts]**

$$L = [17,\ 29,\ 53,\ 61,\ 73,\ 37,\ 43]$$

2020-1-I2-P2–AVL (3/3)



[**0.1 pts**] por las inserciones, [**0.2 pts**] por las rotaciones simples y [**0.2 pts**] por la rotación dobles.
Si el alumno copia y pega una imagen del procedimiento realizado por un sitio web, sin explicación alguna, es 0/0.5 puntos en esta parte.

b)

Por propiedad sabemos que si un árbol es AVL todos sus subárboles son AVL. Esto implica que la diferencia de altura entre el subárbol derecho y izquierdo de todo subárbol del árbol AVL es 0 o 1.

Para que una inserción produzca un desbalance, tiene que cumplirse que el se inserta en el subárbol de la altura mayor, donde ya había una diferencia de altura de 1.
Esto, por que al hacerse está inserción donde ambos subárboles tienen la misma altura, la nueva diferencia sería de 1, y por lo tanto sigue siendo AVL balanceado.

Solo hay 4 posibles casos para los que ocurre esto.
Estos son desbalance externo (clase 7, slide 9) y desbalance interno (clase 7, slide 13), que puede ocurrir para el subárbol derecho u izquierdo. Estos casos son simétricos y las rotaciones se hacen como se indican ahí.

(0.4) Explica cada caso (interno y externo) correctamente.
(0.2) Explica solo un caso (interno y externo) u ambos casos con errores menores.
(0) No explica ningún caso u los explica incorrectamente.

(0.2) Explica cada caso simétrico al ya explicado (implícita o explícitamente).
(0.1) Explica solo un caso simétrico al ya explicado (implícita o explícitamente).

2020-1-I2-P3–Árbol 2-3 (1/3)

3. Los árboles 2-3 son árboles de búsqueda en que los nodos tienen ya sea una clave y dos hijos, o bien dos claves y tres hijos; y todas las hojas del árbol (que se exceptúan de la regla anterior porque no tienen hijos) están a la misma profundidad. Teniendo presente estas propiedades, responde:

    a. ¿Cuál es la altura máxima que puede tener un árbol 2-3 con n elementos? ¿Y la mínima? ¿Cómo es la estructura del árbol cuando ocurre cada uno de estos casos?

    b. Queremos insertar una clave x en un árbol 2-3 T de altura h, que tiene n claves. ¿Qué debe cumplirse para que esta inserción aumente la altura de T? ¿Para qué valores de n está garantizado que **sí** aumentará la altura? ¿Para qué valores de n está garantizado que **no** aumentará la altura?

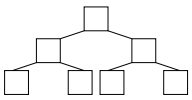
2020-1-I2-P3–Árbol 2-3 (2/3)

Problema 3

a)

La altura será mayor mientras menos elementos tenga el árbol por nivel, y viceversa. Siguiendo esta lógica la altura será mayor cuando todos los nodos sean nodos 2, será menor cuando todos sus nodos sean nodos 3.

(a) Sólo nodos 2



(b) Sólo nodos 3

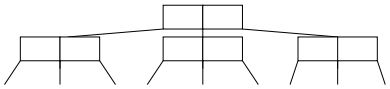


Figura 1: Comparación de estructura de árboles 2-3

Viendo la figura *a*), podemos notar que la cantidad de elementos para el k-ésimo nivel es de 2^{k-1}. Podemos notar también, que la cantidad de elementos acumulados hasta el k-ésimo nivel es $2^k - 1$. Análogamente para *b*), la cantidad de elementos para el k-ésimo nivel es $2 \cdot 3^{k-1}$, y la cantidad de elementos acumulados hasta el k-ésimo nivel es $3^k - 1$. Si despejamos la altura en cada una, obtenemos para la altura mínima:

$$h = \log_3(n+1)$$

y para la altura máxima

$$h = \log_2(n+1)$$

Cabe mencionar que estos casos son cuando el árbol esta completamente lleno. Para alcanzar un nivel k de altura, debe exceder necesariamente los $k-1$ niveles anteriores , y no superar los k niveles. Por lo tanto, la altura mínima en función de la cantidad de elementos quedaría expresada como:

$$h = \lceil \log_3(n+1) \rceil$$

En el caso de la altura máxima, como las hojas deben estar a la misma altura, al añadir un valor extra, el árbol se rebalancea, manteniendo su altura, por lo que quedaría expresada como:

$$h = \lfloor \log_2(n+1) \rfloor$$

en los casos que $n > 0$. Si $n = 0$ en realidad no existiría estructura de datos, por lo que las fórmulas pierden sentido.

Son 0.5 pt por cada fórmula:

- ◇ **0.2 por indicar que el máximo era solo con nodos 2 y el mínimo sólo con nodos 3,**
- ◇ **0.2 por llegar a la fórmula sin función techo/piso (despejando h o n),**
- ◇ **0.1 por la formula con función techo/piso, despejando h**

Si llega a una fórmula incorrecta, se realiza descuento

2020-1-I2-P3–Árbol 2-3 (3/3)

En primer lugar, llamaremos al "camino" de un árbol T a los nodos que hay que recorrer desde la raíz para llegar a un nodo hoja. De esta forma, para que una inserción provoque el aumento de altura de un árbol T se debe cumplir que, en el camino a la hoja donde se insertará la nueva clave, todos los nodos sean nodos 3 (es decir, que tengan 2 claves). Así, se hará un *split* desde la hoja donde insertamos hasta la raíz, aumentando finalmente la altura **(0.3 puntos)**.

Ahora evaluaremos dos casos en dónde siempre aumenta la altura, y donde nunca aumenta la altura para una altura h con n claves.

Garantía de que aumenta la altura: El árbol T debe tener solo nodos 3. Notemos que si pasa lo contrario (al menos uno no es un nodo 3), entonces podemos buscar el camino en donde podamos insertar de tal forma que este nodo pase a ser nodo 3, lo que haría que no aumente la altura. Teniendo esto, la cantidad de claves en función de la altura h es

$$claves = \sum_{i=1}^{h} 2 \cdot \left(3^{i-1}\right)$$

dado que en un nivel i hay 3^{i-1} nodos, con cada uno 2 claves. Resolviendo obtenemos **(0.3 puntos)**

$$claves = 3^h - 1$$

Garantía de que no aumenta la altura: El árbol T no debe tener ningún camino que tenga únicamente nodos 3. Notemos por la definición del comienzo que la condición para que aumente la altura es que agreguemos una clave en la hoja que tenga un camino con sólo nodos 3, por lo que debemos asegurarnos que no exista tal camino. Luego, para garantizar esto debemos contar la cantidad de claves para que no exista un árbol con dicho camino.
La menor cantidad de claves con la que podría aumentar la altura es tal que todos sean nodos 2 excepto un camino en donde sean nodos 3. Teniendo esto, si tomamos esa cantidad de claves - 1 no podremos tener un camino con nodos 3, y garantizaremos desde esa cantidad hacia abajo que no aumente la altura.

Un árbol con las características antes mencionadas tendría la siguiente estructura:

- nivel $1 \rightarrow 2$ claves (nodo 3)
- nivel $2 \rightarrow 4$ claves (nodo 3 + 2 nodos 2)
- nivel $3 \rightarrow 8$ claves (nodo 3 + 6 nodos 2)

  $\vdots$
- nivel $h \rightarrow 2^h$ claves (nodo 3 + $2^h - 2$ nodos 2)

Si sumamos obtenemos

$$claves = \sum_{i=1}^{h} 2^i - 1$$

siendo la suma de las claves de cada nivel - 1 por lo comentado anteriormente. Resolviendo obtenemos

$$claves = 2^{h+1} - 2$$

Luego, para garantizar que la inserción en el árbol T no tendrá efectos en la altura h, la cantidad de claves debe ser menor a $2^{h+1} - 2$. Además, para la factibilidad de un árbol 2-3, se debe cumplir que las claves sean mayores a $2^h - 1$, que es la cantidad mínima de claves que un árbol 2-3 de altura h puede tener **(0.4 puntos)**.

2020-1-Ex-P3–Árbol rojo-negro (1/4)

3. Respecto a los árboles rojo negro:

   a) Justifica que la rama más larga del árbol tiene a lo más el doble de nodos que la rama más corta. Entiéndase por rama la ruta de la raíz hasta una hoja.

   b) En el algoritmo de inserción estudiado en clases, un nodo recién insertado se pinta de rojo. Con esto, corremos el riesgo de violar la propiedad 3 de árbol rojo-negro (según las diapositivas); y cuando así ocurre, usamos rotaciones y cambios de color para restaurar esa propiedad. En cambio, si pintásemos el nodo de negro, no correríamos este riesgo.
      i) ¿Por qué no pintamos de negro un nodo recién insertado?
      ii) Si lo hiciésemos, ¿qué habría que hacer a continuación para volver a tener un árbol rojo-negro?

   c) Considera un árbol rojo-negro formado mediante la inserción de n nodos, siguiendo el algoritmo de inserción estudiado en clase.

      Justifica que si $n > 1$, entonces el árbol tiene al menos un nodo rojo.

2020-1-Ex-P3–Árbol rojo-negro (2/4)

Problema 3

Parte A

OPCION 1

La cantidad de nodos negros en la rama más corta debe ser la misma que en la más larga (por propiedad 4 de ARN vista en clases). Llamemos a esta cantidad "n" **[0,5 por incorporar propiedad y usar cantidad de nodos negros para comparar ambas ramas]**. La rama más corta posible tendrá todos los nodos negros (será de n nodos) **[0,5 mínimo de nodos de rama más corta]**.

Ahora, la rama más larga posible será de $n + r$ nodos, donde r es la cantidad máxima de nodos rojos que puede tener.

Para analizar el máximo r, consideremos la propiedad 3 vista en clases, con la que sabemos que si un nodo es rojo, sus hijos deben ser negros (hojas nulas se consideran negros). Si consideramos además que la raíz es negra, sabemos que la cantidad de nodos rojos será menor a la de negros (por cada rojo en una rama estará su hijo negro ($(n-1) >= r$), y además está la raíz) [Si nulos no se cuentan como negros, la cantidad de rojos puede ser a lo más la de negros ($(n-1) >= r$)] **[0,5 máximo de nodos de rama más larga considerando propiedad 3]**.

Es decir, la rama más larga posible tendrá $n + r$ nodos donde $r = n$ [$r = n - 1$]. Entonces tendrá $2n$ [$2n - 2$] nodos, el doble de los n nodos de la más corta posible- Entonces, podemos decir que la rama más larga de un ARN tendrá a lo más el doble de nodos que la más corta **[0,5 por enunciar conclusión correctamente, sólo si tenía bien los pasos anteriores]**

OPCION 2

Usamos análisis del árbol 2-3-4. Notamos que en él, todas las ramas tienen la misma altura h **[0.5]**. Además, como los nodos de tipo 2 corresponden a 1 nodo negro en ARN (un nivel), los de tipo 3 a 1 negro con un hijo rojo en ARN (dos niveles), y los de tipo 4 a 1 negro con dos hijos rojos en ARN (dos niveles), vemos que la rama más corta posible en el ARN tendrá solo nodos tipo 2 en el 2-3-4, y se convertirá en el ARN en una rama con altura h **[0.5 altura más corta]**. Por otro lado, la más larga posible estará constituida por nodos tipo 3 o 4, y esto se traducirá en ARN en una rama de altura $2h$ **[0.5 altura más larga]**. Así, vemos que la rama más larga posible tendrá 2 veces la altura de la más corta posible. Es decir, la más larga tendrá a lo más el doble de nodos de la más corta **[0.5]**.

Parte B

i

Porque para que un árbol sea RN, debe cumplir con la propiedad 4. Si insertamos una hoja negra a un árbol que ya es RN y queda con más de una rama, siempre romperemos la propiedad (excepto para la raíz) **[0.6 por mencionar que propiedad se rompe siempre o siempre que no sea la raíz]**. Esto se explica ya que, si antes tenía "n" nodos negros en cada rama, ahora tendrá "n+1" en una de ellas, y "n" en las demás. Así, al insertar el nodo como negro la propiedad se rompe siempre, mientras que si insertamos el nodo como rojo, romperá la propiedad 3 solo a veces. **[0.2 por hacer distinción por frecuencia]** Además, restaurar la propiedad 4 será en general más costoso, ya que incide globalmente en el árbol, mientras que la propiedad 3 es más fácil de arreglar con cambios locales **[0.2 por hacer distinción por complejidad]**

2020-1-Ex-P3–Árbol rojo-negro (3/4)

ii

Deberíamos restaurar la propiedad 4 (sin violar las otras, cuidando sobre todo la propiedad 3) **[0,2]**. Si quisieramos priorizar mantener el nodo insertado como negro, esto lo podríamos conseguir con un mecanismo que parta localmente con rotaciones y cambios de color que se complejizaría mucho, o con un mecanismo que revise todo el árbol globalmente **[0.3]**.

Si pensamos en el árbol 2-3-4 equivalente, lo que estamos haciendo es insertar un nodo como tipo 2, aún cuando el nivel padre no se haya llenado. Esto, además de no asegurar que se cumplan las propiedades, no es lo más eficiente (y el árbol dejaría de tener la propiedad de 2-3-4 de todas las ramas a la misma profundidad, y se ropmería el mecanismo de split. Por lo tanto, pensemos en los casos en que se puede mantener el nodo insertado m como negro y aún así respetar la forma equivalente de 2-3-4.

(a) Se inserta bajo un nodo n tipo 2 (En ARN, sería un nodo negro sin hijos). Acá podemos forzar el nodo a subir, y podemos elegir dejarlo como el nodo negro (y n pasa a ser rojo). En ARN, n sería un nodo negro sin hijos. Lo que se haría es una rotación simple con el insertado, y cambio de color.

(b) Se inserta bajo un nodo n tipo 3, que contiene a p y q (En ARN, sería un nodo negro p con 1 hijo rojo q). Nuevamente, forzamos el nodo a subir. Si entra por el medio ($p < m < q$ ó $p > m > q$) (m entra como hijo interno de q; hacia el lado de p), podemos dejarlo como nodo negro con p y q como hijos. En caso contrario, sería muy costoso intentar mantener a m negro, y conviene simplemente pintarlo rojo.

(c) Se inserta bajo un nodo n tipo 4, que contiene a p, q y r (En ARN, sería un nodo negro p con 2 hijos rojos, q y r). Nuevamente, forzamos a m a subir. Si entra por el medio ($r < m < q$ ó $r > m > q$) (m entra como hijo interno de r o q), podemos dejarlo como nodo negro de hijo tipo 3 generado al subir a p mediante split. En caso contrario, es similar a (b) y se pinta rojo.

[0,5 Por mencionar solo uno de los casos, o ejemplo (con condiciones declaradas), en que nodo insertado quede definitivamente negro. Alternativamente, se acepta un mecanismo global que revise el árbol completo y deje un ARN que cumpla las propiedades]

Un ejemplo de mecanismo global sería insertar como en AVL, luego contar la altura h, y colorear desde las hojas hacia arriba, nivel por medio, hasta que cada rama tenga h nodos negros.

Parte C

OPCIÓN 1

En clases se usa la equivalencia del árbol Rojo Negro (ARN) con un árbol 2-3-4 (A) para la inserción **[0.2]**. En el árbol A, los nodos de tipo 2 corresponden a 1 nodo negro en ARN, los de tipo 3 a 1 negro y 1 rojo en ARN, y los de tipo 4 a 1 negro y 2 rojos en ARN [0.3 Puede estar a lo largo de la explicación, deben estar los tres]. Entonces, solo necesitamos saber que en un árbol 2-3-4 con más de un elemento, siempre habrá al menos un nodo tipo 3 o 4. **[0.3]**

Al insertar el primer nodo, (A) queda con un único nodo tipo 2. Al insertar el 2do (n=2 es caso base con n¿1), (A) queda con un único nodo tipo 3 y la condición se cumple **[0.2 Caso base]**. En adelante al ir insertando nodos que aumenten n, la única forma en que un nodo deje de ser tipo 3 es que pase a ser tipo 4 al recibir una nueva llave (seguiría cumpliendo la condición) **[0.4]**, y la única forma en que deje de ser tipo 4 es que ocurra un split al recibir una nueva llave, pero esto siempre deja como hijos dos nodos en A, uno tipo 2 y otro tipo 3 (seguiría cumpliendo la condición) **[0.4]**.
Entonces, como con n=2 se cumple la condición y al ir insertando nodos en RN (elementos en (A)) se sigue cumpliendo, podemos decir que siempre se cumple para n¿1 y que, para n¿1, un ARN formado con el método visto en clases siempre tiene al menos un nodo rojo **[0.5 si enuncia conclusión, solo si justificación es correcta]**

2020-1-Ex-P3–Árbol rojo-negro (4/4)

OPCIÓN 2

CB: n= 2, Raíz negra con un hijo rojo. $R = 1 > 0$ **[0.2 Necesario solo si en cada caso se argumenta que se mantiene o aumenta número de rojos. Si se usa argumento de que cada caso deja algún rojo, se suman los 0,2 a 0.5 de conclusión]**

Hay cinco casos de inserción:

(a) C1: Raíz. Se da solo con n = 1, no aplica a condición [Puede omitirse]

(b) C2: Padre negro. Nodo se inserta rojo y queda rojo. Rojos aumentan [$R > 0$] **[0.2 por presentar caso y explicar bien que R se mantiene positivo]**

(c) C3: Padre rojo y tío rojo. Se mantiene número de nodos rojos. [Rojos resultantes entre abuelo, tíos e insertado = $2 > 0$] [0.3 por presentar caso y explicar bien que R se mantiene positivo]

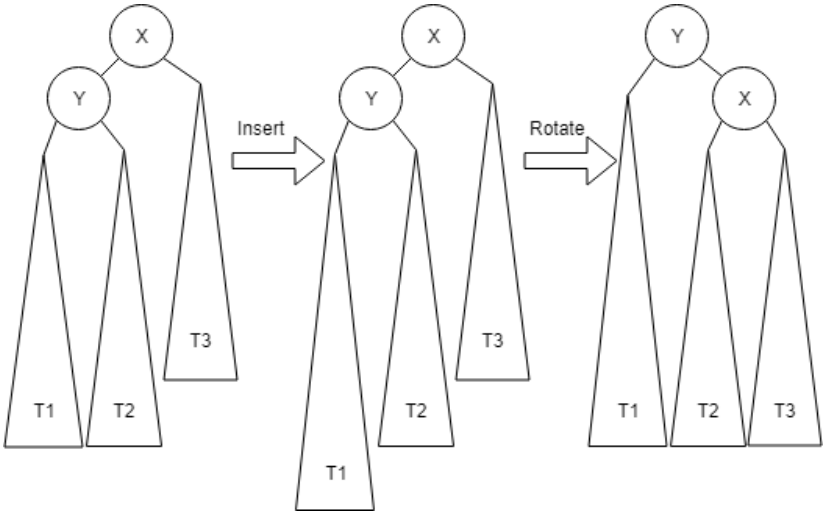
(d) C4: Padre rojo, tío negro, es hijo interno (su valor está entre su padre y su abuelo). Lleva a C5 aumentando nivel previo a inserción de rojos en uno, y como después de C5 siempre se mantiene el número de rojos (considerando el insertado), el caso cumple. [0.3 por presentar C4 y C5 y explicar bien que R se mantiene positivo. Se pueden presentar juntos].

(e) C5: Padre rojo, tío negro, es hijo externo (su valor no está entre padre y abuelo]. Rojos entre abuelo, tíos e insertado pasa de 1 a 2 [> 0].

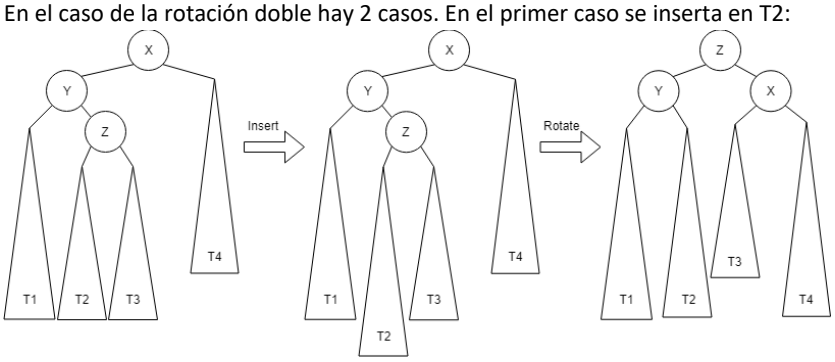
Como en n=2 hay un nodo rojo y para todos los casos posibles de inserción con $n > 1$ se mantiene o aumenta R [Como siempre al insertar un nodo con $n > 1$ se asegura $R > 0$], podemos decir que, para $n > 1$, un ARN formado con el método visto en clases siempre tiene al menos un nodo rojo [0.5 por llegar a conclusión correcta en base a todos los casos bien justificados (solo si están bien los pasos anteriores)]

2019-2-I1-P3–AVL, rotaciones (1/2)

3. Se tiene un árbol AVL T en el que se inserta una clave k que produce un desbalance. Sea x el nodo más profundo en la ruta de inserción de k tal que el subárbol que tiene como raíz a x está desbalanceado, y sean y y z los siguientes dos nodos bajando por esa ruta de inserción, como se definió en clases. Demuestra que:
   a. [3pt] Luego de hacer una rotación simple o doble en torno a x, y y z, el subárbol que tiene como raíz x vuelve a tener la misma altura que tenía antes de insertar k.
   Solución: Para demostrar esto se analizará el caso general de la rotación simple y el caso general de la rotación doble en una dirección cada una ya que el caso simétrico es equivalente.
   El caso general de la rotación simple se ve así:

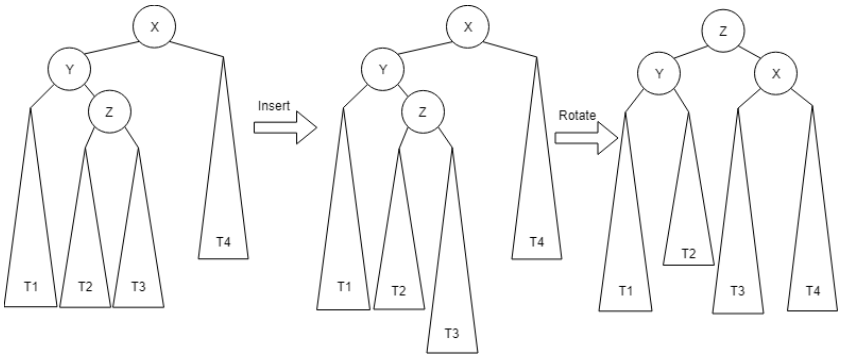


   Separamos el subárbol que rota en 5 partes: el nodo X, el nodo Y, y los subárboles T1, T2 y T3. La altura inicial antes de insertar es 2 + k donde k es la altura de T1 y T2. T3 tiene altura k – 1.
   Al insertar el árbol T1 aumenta su altura en 1 lo que produce el desbalance (si se inserta en T2 y crece entonces es una rotación doble).
   Al hacer la rotación simple el subárbol T2 queda a la misma profundidad de antes, el subárbol T1 sube 1 nivel y el subárbol T3 baja 1 nivel.
   Ya que T3 estaba 1 nivel más arriba que T2 antes de rotar, ahora llega a la misma profundidad y ya que T1 subió un nivel, ahora llega a la misma profundidad que antes de insertar. Por lo tanto, la altura del árbol sigue siendo k + 2.
   En el caso de la rotación doble hay 2 casos. En el primer caso se inserta en T2:



   Y en el segundo caso se inserta en T3:

2019-2-I1-P3–AVL, rotaciones (2/2)



El análisis es el mismo para ambos casos ya que en ambos casos suben tanto T2 como T3, T1 se queda a la misma altura y T4 baja 1 nivel. Esto hace que 3 de los 4 árboles queden a la profundidad máxima antes de insertar y uno de ellos quede más arriba. Por lo tanto, la altura del árbol se mantiene.

b. [1.5pt] Demuestra que esta rotación no genera otros desbalances en el árbol $\boldsymbol{T}$.
Solución: Para demostrar esto hay que demostrar que en el subárbol rotado no queda ningún desbalance y que no se producen desbalances fuera del subárbol rotado.
En el caso de la rotación simple y la doble, los árboles T1, T2, T3 y T4 se mantienen intactos luego de insertar y rotar. Además, ya que el nodo X es el de más abajo en estar desbalanceado, ninguno de los subárboles está desbalanceados luego de insertar. Por lo tanto, luego de rotar siguen balanceados.
Por otro lado, el subárbol que se rota mantiene su altura (demostrado en a), por lo tanto, si se produjo en desbalance en un nodo más arriba que X al insertar, este se corrige al momento de rotar.

c. [1.5pt] Demuestra que esta rotación basta para solucionar el desbalance del árbol $\boldsymbol{T}$.
Solución: Dado lo demostrado en a y en b y sabiendo que el árbol estaba balanceado antes de insertar, sabemos que todos los subárboles superiores al árbol rotado quedan balanceados y que todos los subárboles que están incluidos en la rotación también quedan balanceados. Por lo tanto, la rotación hecha fue suficiente para balancear el árbol luego de insertar.

2019-2-I1-P4–Heap (1/1)

4. Queremos extender la funcionalidad de un *Heap Binario*. Asumiendo que tienes las funciones $\boldsymbol{insert}$, $\boldsymbol{extract}$, $\boldsymbol{sift-up}$ y $\boldsymbol{sift-down}$, escribe un algoritmo para las siguientes operaciones:
   a. **[3pt]** Modificar la prioridad del elemento en la posición $\boldsymbol{i}$
   Solución: El algoritmo de cambio de prioridad es el siguiente:
   $update\ priority(H, i, value)$:
   $H[i] = value$
   $sift - up(i)$
   $sift - down(i)$
   Notar que solo sift up o sift down moverán el elemento, pero no ambos.

   b. **[3pt]** Eliminar del *Heap* el elemento en la posición $\boldsymbol{i}$
   Solución: El algoritmo de eliminación es similar al anterior pero un poco distinto:
   $pop(H, i)$:
   $value = H[i]$
   $H[i] = H[H.count]$
   $H.count - -$
   $sift - up(i)$
   $sift - down(i)$

   Tanto el algoritmo en a como el algoritmo en b funcionan en O(logn)

2019-2-I2-P1-1–Árbol rojo-negro (1/1)

Pauta I2 parte I

1) Considera un árbol rojo-negro en que el número de nodos negros en cada ruta de la raíz a una hoja es *k*;

a) [1 pt] ¿Cuál es la altura máxima posible del árbol? ¿Y la mínima? Justifica.
b) [1 pt] ¿Cuál es el máximo número de nodos que puede tener el árbol? ¿Y el mínimo? Justifica.

En este árbol hacemos una inserción:

c) [2 pts] ¿Cuál es la cantidad máxima de cambios de color que pueden ocurrir? Justifica.
d) [2 pts] ¿Cuál es la cantidad máxima de rotaciones (simples o dobles) que pueden ocurrir? Justifica.

Solución:

a) [0.5 pts] La altura máxima posible del árbol es $2k$, y esta se obtiene con un árbol que alterna nodos rojos y negros desde la raíz hasta las hojas partiendo con la raíz negra y terminando con las hojas rojas. No puede ser más alto porque no puede haber más nodos negros por enunciado y no puede haber más nodos rojos por las reglas del árbol rojo-negro.
[0.5 pts] La mínima altura es k, y es posible con un árbol binario completamente hecho de nodos negros. No rompe ninguna regla de los árboles rojo negro y no puede haber un árbol de menor altura por enunciado.
b) [0.5 pts] El máximo de nodos se obtiene con el árbol descrito en la primera parte de parte a), el cual tiene $2k$ niveles completos. Este árbol tiene $\sum_{i=0}^{2k-1} 2^i = 2^{2k} - 1$ nodos en total.
Opción 2:
[0.5 pts] El máximo de nodos se obtiene con el árbol 2-4 equivalente al árbol descrito en la primera parte de parte a), el cual tiene k niveles completos con nodos 4. Este árbol tiene $\sum_{i=0}^{k-1} 3 \cdot 4^i = 3 \cdot \sum_{i=0}^{k-1} 4^i = 3 \cdot \frac{4^k-1}{4-1} = 4^k - 1$ nodos en total.

[0.5 pts] El mínimo se obtiene con el árbol descrito en la segunda parte de a), el cual tiene k niveles completos. Este árbol tiene $\sum_{i=0}^{k-1} 2^i = 2^k - 1$.
c) [1 pt] El máximo de cambios de color que pueden ocurrir es O(k).
[2 pt] Por ejemplo, en el árbol descrito en la primera parte de a), si insertamos un nodo rojo, tendríamos que realizar un cambio de color en cada nivel hasta llegar a la raíz. No puede haber más cambios de color ya que los cambios de color se realizan solo en la ruta de inserción del nuevo dato además de los tíos de los nodos de la ruta de inserción.
d) [1 pt] El máximo de rotaciones que se hacen en la inserción es 1.
[1 pt] Una rotación corresponde a un reordenamiento de los datos en un nodo del árbol 2-4 equivalente y solo ocurre en la hoja en la que se insertó. Una vez rotado, el árbol sigue cumpliendo las propiedades de árbol rojo negro por lo que no son necesarias más rotaciones.

2019-1-C2–AVL (1/1)

Estructuras de Datos y Algoritmos - IIC2133
Control 2
3 de abril, 2019

1) [6 pts.] Escribe un algoritmo que, dado un valor *h*, calcule el número mínimo de nodos que puede tener un árbol AVL de altura *h*. Considera que un árbol que tiene solo un nodo tiene altura $h = 1$. Usa la notación pseudocódigo empleada en las diapositivas de las clases.

R: Algoritmo para mínima cantidad de nodos en AVL:

```
min_nodes(h):
    if h = 0:
        return 0
    if h = 1:
        return 1

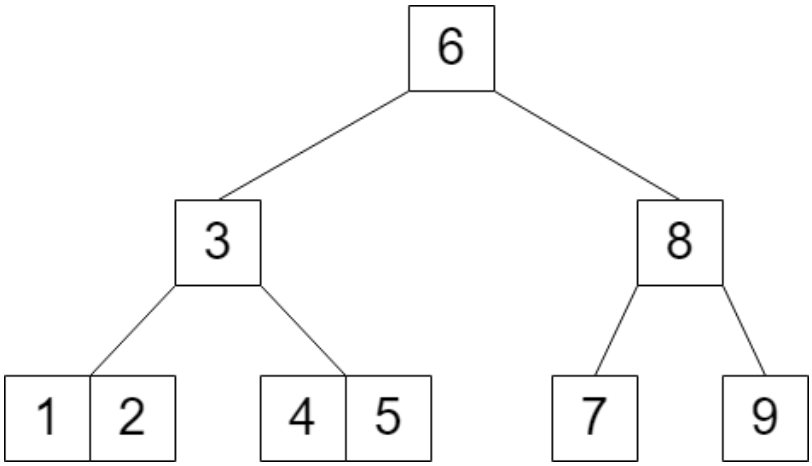
    return 1 + min_nodes(h - 1) + min_nodes(h - 2)
```

2a) [4 pts.] Encuentra un orden para insertar las claves 1, 2, 3, 4, 5, 6, 7 y 8 en un árbol 2-3 inicialmente vacío, de modo que el árbol resultante sea el que se muestra en la figura. Justifica, realizando las inserciones en el orden que encontraste.

R: Un posible orden de inserción es: 1, 3, 4, 6, 7, 2, 5, 8. Se otorgará el puntaje completo solo si el orden de inserción permite llegar al estado de la figura. Se descontará puntaje por tener pasos incorrectos. Debe existir justificación.

2b) [2 pts.] Inserta la clave 9 en el árbol de la figura y muestra el árbol 2-3 resultante.

R: Se otorga todo el puntaje si el árbol coincide con el mostrado a continuación.



2019-1-Ex-P1–Árboles rojo-negro (1/2)

PAUTA EXAMEN IIC2133 2019-1

1. Con respecto a los árboles rojo-negros:

a) **[2pt]** Justifica que es válido que un árbol rojo-negro con más de un nodo esté formado sólo por nodos negros.

Solución:

La justificación es posible hacerla directamente desde las propiedades de los **ARN**, o desde las propiedades de los **2-3 / 2-4.**

2-3 / 2-4:

Recordar que un nodo negro en un **ARN** representa un *nodo-2* en un árbol **2-3 / 2-4 [1pt]**, y un árbol **2-3 / 2-4** que sólo contiene *nodos-2* es válido. **[1pt]**

ARN:

De las propiedades de los **ARN:**

1. Un nodo puede ser rojo o negro
2. La raíz del árbol es negra
3. Los nodos rojos no pueden tener hijos rojos
4. La cantidad de nodos negros camino a cada una de las hojas debe ser la misma

[1pt]

Ninguna de estas propiedades menciona nada sobre los nodos negros con hijos negros, por lo que en principio no debería haber problema. La única propiedad que podría violarse en el caso que se pide es la 4, por lo que para que se cumpla la propiedad debe ser un árbol completo (con todos sus niveles llenos) **[1pt]**

b) **[2pt]** Justifica que un árbol rojo-negro construido sólo mediante la inserción de *n* nodos tiene al menos un nodo rojo.

Solución:

El primer paso de la inserción en un ARN es insertar el elemento como se haría en un ABB. El nodo recién insertado siempre es rojo. **[0.5pt]** Si este nodo genera conflictos estos se solucionan mediante rotaciones y cambio de color de sus ancestros, pero nunca se cambia de color el nodo recién insertado. Por lo tanto

2019-1-Ex-P1–Árboles rojo-negro (2/2)

al menos este nodo es rojo. **[1pt]** Eso sí, si el nodo queda como la raíz (*n = 1*), se debe pintar de negro. Es decir, esta propiedad solo se cumple para *n > 1*. **[0.5pt]**

c) Supongamos que todo nodo rojo en un árbol rojo-negro es "absorbido" por su padre, de modo que los hijos del nodo rojo pasan ahora a ser hijos de su abuelo (olvídate de lo que ocurre con las claves).

- **[1pt]** ¿Cuáles son los posibles números de hijos de un nodo negro del árbol después de que todos los nodos rojos han sido absorbidos?

  Solución:

  El mínimo posible es para los nodos negros que son hoja: 0 **[0.5pt]**

  El máximo posible es para los nodos negros que tienen dos hijos rojos, los cuales a su vez tienen dos hijos negros cada uno: 4 **[0.5pt]**

  [Formalidad] Por lo tanto el número de hijos de un nodo negro en este nuevo árbol puede ir entre 0 y 4.

- **[1pt]** ¿Qué puedes afirmar y ~~por qué~~ sobre las profundidades de las hojas del árbol resultante?

Solución:

[No tiene sentido preguntar por qué. Cualquier conclusión surge directamente de la definición del proceso.]

Cualquiera de estas explicaciones se considera como correcta:

- Para una hoja negra cualquiera a profundidad *P*, si existen *R* nodos rojos camino a la raíz del árbol, la profundidad de esta hoja pasa a ser *P - R,* ya que todos los nodos rojos en el camino desaparecen. **[1pt]**

- La profundidad del árbol puede verse reducida hasta en un 50% dependiendo de cómo estén distribuidos los nodos rojos. **[0.5pt]** Dar un ejemplo del mejor **[0.25pt]** y peor caso **[0.25pt]**.

2018-2-I1-P1–AVL (1/1)

Estructuras de Datos y Algoritmos – IIC2133
I1
5 de septiembre, 2018

1. Árboles AVL

a) Queremos almacenar las claves 0, 1, 2, 3, 4, 5, 6, 7, 8 y 9 en un árbol AVL inicialmente vacío. ¿Es posible insertar estas claves en el árbol en algún orden tal que ***nunca*** sea necesario ejecutar una rotación? Si tu respuesta es "sí", indica el orden de inserción y muestra al árbol resultante después de insertar cada clave. Si tu respuesta es "no", da un argumento convincente (p.ej., una demostración) de que efectivamente no es posible insertar las claves sin que haya que ejecutar al menos una rotación.

Sí es posible. La idea es hacer las inserciones de manera de mantener todo el tiempo la propiedad de balance; p.ej., procurar que el árbol se vaya llenando "por niveles". La primera clave que insertemos va a ser la raíz del árbol (ya que la idea es que no va a haber rotaciones). Por lo tanto, tiene que ser una clave k tal que el número de claves menores que k —que van a ir a parar al subárbol izquierdo— y el número de claves mayores que k —que van a ir a parar al subárbol derecho— sean parecidos. Si elegimos la clave 4, entonces hay 4 claves menores y 5 claves mayores (también podemos elegir la clave 5 y dejar 5 claves menores y 4 mayores). A continuación elegimos la raíz del subárbol izquierdo y la raíz del subárbol derecho (o viceversa). Para esto, aplicamos recursivamente la misma "regla", sobre las claves 0, 1, 2 y 3, para el subárbol izquierdo, y sobre las claves 5, 6, 7, 8 y 9, para el subárbol derecho; p.ej., insertamos 2 y luego 7. Repitiendo la estrategia, luego insertamos 1, 3, 6 y 8, y finalmente 0, 5 y 9. Así, un orden de inserción posible es 4, 2, 7, 1, 3, 6, 8, 0, 5, 9.

b) Considera la inserción de una clave x en un árbol AVL T. Definimos la *ruta de inserción* de x como la secuencia de nodos, empezando por la raíz de T, cuyas claves son comparadas con x durante la inserción. El ***procedimiento de rebalanceo*** —una vez hecha la inserción— primero sube (de vuelta, desde el nodo recién insertado) por la ruta de inserción examinando cada nodo r que está en la ruta: si r es raíz de un (sub)árbol AVL-balanceado, entonces se sigue subiendo; de lo contrario, se ejecuta la rotación que vimos en clase en torno a la arista *r—hijo izquierdo* o *r—hijo derecho*, según corresponda. Muestra los árboles AVL que se van formando al insertar las claves 3, 2, 1, 4, 5, 6, 7 y 16, en este orden, en un árbol AVL inicialmente vacío.

Inicialmente se genera el árbol con raíz 3, y se agrega como hijo izquierdo el nodo con key 2. Luego se inserta a la izquierda de 2 la clave 1 y se hace una rotación simple en la que queda 2 como raíz y sus hijos 1 y 3 a la izquierda y derecha respectivamente.
Al insertar el 4 este queda a la derecha del 3, y luego al insertar el 5 este queda a la derecha del 4 y se hace una rotación simple entre 3, 4 y 5 y queda 4 como hijo derecho de 2 y 3 y 5 como hijos de 4 a la izquierda y derecha respectivamente.
Luego se inserta el 6 a la derecha del 5 y esto crea un desbalance en el nodo raíz, por lo que se hace una rotación simple entre 2, 4 y 5 quedando como raíz el nodo 4. Del nodo cuatro cuelga a la izquierda el nodo 2, el cual tiene a su izquierda el nodo 1 y a su derecha el nodo 3. A la derecha de 4 cuelga el nodo 5, el cual tiene el nodo 6 como hijo derecho.
Al insertar el nodo 7 este queda a la derecha del nodo 6 y se hace una rotación simple entre los nodos 5, 6 y 7, quedando el nodo 6 como padre de 5 y 7.
Al insertar el nodo 16 queda como hijo derecho del nodo 7 y no hay rotación.

2018-2-I1-P2–ABB (1/1)

2. Árboles de búsqueda binarios (no necesariamente balanceados)

En clase vimos cómo se elimina una clave de un árbol de búsqueda binario (ABB). Eliminar la clave cuando el nodo que ocupa no tiene hijos o tiene sólo un hijo, *Ti* o *Td*, es fácil.

a) Es más difícil eliminar una clave cuando el nodo que ocupa tiene ambos hijos, *Ti* y *Td*; describe las acciones correspondientes. Esta forma de eliminación se llama ***eliminación por copia***. [**1 pt.**]

Se busca la clave sucesora (o predecesora) de la clave eliminada, y se la coloca, junto con su descendencia, en lugar de ésta (de la eliminada); luego, se elimina la clave sucesora, que a lo más tiene un hijo.

b) ¿Es la eliminación por copia "conmutativa" en el sentido de que eliminar *x* y luego *y* de un ABB deja el mismo árbol que eliminar *y* y luego *x*? Demuestra que lo es o da un contraejemplo. [**1 pt.**]

Contraejemplo: Supongamos que al eliminar un nodo con dos hijos, lo reemplazamos por su sucesor. Consideremos una raíz con clave 5, y dos hijos, con claves 3 y 11, respectivamente; el nodo con clave 11 a su vez tiene un hijo izquierdo con clave 7. Si eliminamos el nodo con clave 5 (dos hijos) y luego el nodo con clave 3 (hoja), dejamos un abb —raíz 7 e hijo derecho 11— distinto que si eliminamos el nodo con clave 3 (hoja) y luego el nodo con clave 5 (ahora sólo un hijo) —raíz 11 e hijo izquierdo 7.

c) Otra forma de eliminar una clave cuyo nodo tiene ambos hijos es ***eliminación por mezcla***: el nodo es ocupado por su (hijo y) subárbol izquierdo, *Ti*, mientras que su subárbol derecho, *Td*, se convierte en el subárbol derecho del nodo más a la derecha de *Ti*. Justifica que esta eliminación respeta las propiedades de ABB. [**2 pts.**]

A partir de la regla para insertar claves, que, a su vez, cumple la propiedad fundamental de ABB, sabemos que en el proceso de inserción de cualquiera de las claves que están en Ti o Td —llamemos k a una clave cualquiera en Ti o Td— pasamos por la clave —llamémosla j— que estamos eliminando, y que, por lo tanto, k podría haber ido a parar al lugar de j, posición en la que habría cumplido la propiedad de ABB con respecto al resto del árbol. Por lo tanto, poner el subárbol Ti en la posición que ocupaba el nodo con la clave j es válido.

La pregunta entonces es, ¿qué hacemos con Td? De nuevo, por la propiedad de ABB, las claves de Td son todas mayores que las claves de Ti. La única posición que corresponde a claves mayores que todas las claves de Ti, pero al mismo tiempo menores que las otras claves del árbol mayores que j es como hijo derecho de la clave más a la derecha de Ti —llamémosla m; obviamente, esta posición está "desocupada": m sólo puede ser la clave más a la derecha de Ti si no tiene hijo derecho.

d) Muestra con ejemplos que la eliminación por mezcla puede tanto aumentar como reducir la altura del árbol original. [**2 pts.**]

Para simplificar (y generalizar un poco), eliminamos la raíz del árbol. Entonces, Ti "sube" a esta posición y agregamos Td como hijo derecho del nodo más a la derecha de Ti.

La altura del árbol original, T, era H(T) = max{ H(Ti), H(Td) } + 1. La altura del nuevo árbol, T', puede ser desde H(T') = H(Ti) hasta H(T') = H(Ti) + H(Td), dependiendo de la profundidad del nodo más a la derecha de Ti. En el primer caso, H(T') es claramente menor que H(T); en el segundo, H(T') claramente puede ser mayor que H(T).

2018-2-I2-P3–ABB, rotaciones, Árbol rojo-negro (1/2)

3. Rotaciones + árboles de búsqueda balanceados

a) [*Teorema fundamental de las rotaciones*]. Muestra que cualquier árbol binario de búsqueda (no necesariamente balanceado) puede ser transformado en cualquier otro árbol binario de búsqueda con las mismas claves mediante una secuencia de rotaciones simples.

b) Determina un orden en que hay que insertar las claves 1, 3, 5, 8, 13, 18, 19 y 24 en un árbol 2-3 inicialmente vacío para que el resultado sea un árbol de altura 1, es decir, una raíz y sus hijos.

c) Considera un árbol rojo-negro formado mediante la inserción de n nodos usando el procedimiento visto en clase. Justifica que si $n > 1$, el árbol tiene al menos un nodo rojo.

d) Muestra cómo construir un árbol rojo-negro que demuestre que, en el peor caso, casi todas las rutas desde la raíz a una hoja tienen largo $2 \log N$, en que N es el número de nodos del árbol.

Respuesta:

2018-2-I2-P3–ABB, rotaciones, Árbol rojo-negro (2/2)

a) Demostrar recursivamente. Dados dos árboles A y B con las mismas claves en distinto orden (respetando las propiedades de los ABB), una posibilidad es tomar el nodo raíz de A y buscarlo en B, y luego hacer las rotaciones necesarias para llevarlo a la raíz, y luego repetir el proceso recursivamente sobre los subárboles siguientes.

Alternativamente, se puede mostrar que todo ABB se puede reordenar mediante rotaciones simples para llevarlo a una lista ligada (visto como un árbol en que todos los nodos tienen solo hijos derechos o izquierdos), la cual sería igual para todo ABB con las mismas claves, y dichas rotaciones se pueden deshacer, llegando así de cualquier árbol A a un árbol B con las mismas claves.

b) Hay varios órdenes posibles. Para que el árbol tenga altura 1 este tiene que quedar con 5 y 18 en la raíz y con 1 y 3 en el hijo izquierdo, 8 y 13 en el medio y 19 y 24 en el derecho. Un orden posible es 1-18-24-5-8-13-19, pero en todo caso es necesario mostrar la construcción del árbol y como va quedando paso a paso hasta llegar al resultado indicado.

c) Se puede demostrar inductivamente. Comenzando por el caso base, deben explicar las reglas de construcción de árboles rojo-negro (específicamente que los nodos insertados siempre son rojos), y luego explicar el paso inductivo mediante los casos posibles para la inserción de un nodo (padre negro, padre rojo con tio rojo, padre rojo con tio negro) de manera de mostrar que siempre queda al menos un nodo rojo después de que la inserción es balanceada.

d) Partiendo del caso en que un árbol rojo-negro fuese compuesto solo de nodos negros (esto es, un ABB normal balanceado), se tiene que la altura de este es O(log n) para n nodos. Luego, teniendo en cuenta que todo nodo rojo tiene hijos negros, el peor caso en términos de altura es aquel árbol en que se alternan nodos rojos y negros a cada nivel, de manera que su altura es del doble que el árbol mencionado previamente pero mantiene aún las propiedades de los árboles rojo-negros.

Alternativamente, se puede mencionar la altura de los árboles 2-4 (que siempre son transformables a un árbol rojo-negro y viceversa), siendo esta O(log n), y mostrar cómo esta aumenta al ser transformada en un árbol rojo negro, con un ejemplo que muestre un árbol 2-4 cuya altura aumente al doble al ser transformado en rojo-negro.

2018-2-Ex-P1–ABB (1/1)

Estructuras de Datos y Algoritmos – IIC2133

Examen

28 de noviembre de 2018

1. Árboles binarios de búsqueda

Las claves en un árbol binario de búsqueda están ordenadas. Por lo tanto, debería ser fácil encontrar la k-ésima clave más pequeña o determinar cuántas están en el rango $[a, b]$. Explica cómo podría hacerse:

a) ¿Qué información adicional habría que mantener en cada nodo?

En cada nodo x podemos almacenar en un campo adicional *x.size* el número total de nodos que tiene el árbol cuya raíz es x, incluyendo a x. Así, p.ej., una hoja h tiene *h.size* = 1, y si el árbol almacena 1000 claves, entonces la raíz r del árbol tiene *r.size* = 1000.

b) Describe los algoritmos necesarios para responder las consultas mencionadas más arriba.

Suponemos que cada nodo x tiene punteros *left*, *right* y p, a sus hijos izquierdo y derecho y a su padre.

```
k-esima(x, k):                  —esta operación es O(altura del árbol)
    r = x.left.size + 1
    if k = r:
        return x
    else: if k < r:
        return k-esima(x.left, k)
    else:
        return k-esima(x.right, k–r)
```

Para determinar cuántas claves están en el rango $[a, b]$, hay que saber qué posición ocupan a y b en un orden lineal de todas las claves (recorrido *inorder* del árbol T) —esto es, el ranking de a y el ranking de b— y luego restar, ranking(b, T) – ranking(a, T):

```
ranking(x, T):                  —esta operación también es O(altura del árbol)
    r = x.left.size + 1
    y = x
    while y ≠ T.root:
        if y == y.p.right:
            r = r + y.p.left.size + 1
        y = y.p
    return r
```

c) ¿Cuánto cuesta mantener la información adicional? En particular, si se hace una inserción en el árbol; y si se hace una eliminación. Justifica.

En ambos casos, el costo es O(altura del árbol). En una inserción, hay que incrementar *x.size* para cada nodo x en el camino desde la raíz al punto de inserción; el nuevo nodo (que es una hoja) tiene *size* = 1. En una eliminación, si eliminamos una hoja o un nodo con un solo hijo, hay que decrementar *x.size* para cada nodo x en el camino desde el padre del nodo eliminado hasta la raíz; si eliminamos un nodo con dos hijos, primero lo reemplazamos por su sucesor y luego eliminamos el sucesor (de su posición original).

2018-1-I1-P1–AVL (1/2)

Estructuras de Datos y Algoritmos – IIC2133

I1

3 de abril, 2018

1. Árboles AVL

a) Considera un árbol AVL que almacena las claves 1, …, 6 de la siguiente manera: 4 está en la raíz, 2 y 5 son sus hijos; 1 y 3 son hijos de 2; y 6 es hijo de 5. En este árbol, inserta las siguientes claves, en el orden dado: 7, 16, 15, 14 y 13.

En cada caso, muestra el árbol resultante justo después de la inserción, pero antes de rebalancear el árbol (si fuera necesario), e indica si es necesario o no rebalancear el árbol. Si es necesario rebalancear el árbol, entonces explica por qué es necesario, identifica la acción que hay que realizar, y muestra el árbol resultante después de realizarla.

Respuesta

Denotaremos las aristas como "nodo padre"–"nodo hijo". La inserción de 7 produce un desbalance en 5 y requiere una rotación a la izquierda en torno a la arista 5-6 [**1 pt.**]. La inserción de 16 no produce desbalances [**0.5 pts.**]; pero la de 15 produce un desbalance en 7 y exige una doble rotación: a la derecha en torno a 16-15, y a la izquierda en torno a 7-15 [**1.5 pts.**]. La inserción de 14 produce un desbalance en 6 y también exige una rotación doble: a la derecha en torno a 15-7, y a la izquierda en torno a 6-7 [**2 pts.**]. Finalmente, la inserción de 13 produce un desbalance en 4 (la raíz) y exige una rotación en torno a 4-7.

El árbol resultante tiene a 7 en la raíz, con hijos 4 y 15. Los hijos de 4 son 2 y 6, los hijos de 2 son 1 y 3, y el único hijo de 6 es 5. Los hijos de 15 son 14 y 16, y el único hijo de 14 es 13.

2018-1-I1-P1–AVL (2/2)

b) Considera la inserción de una clave x en un árbol T. Definamos la *ruta de inserción* de x como la secuencia de nodos, empezando por la raíz de T, cuyas claves son comparadas con x durante la inserción. Como vimos en clase, el algoritmo de rebalanceo primero sube (de vuelta) por la ruta de inserción examinando cada nodo que está ahí. Con respecto a esta "subida" (es decir, todavía antes de rebalancear propiamente), explica claramente:

- en qué consiste realmente "examinar" el nodo;
- qué casos pueden darse y qué hay que hacer en cada uno de ellos (para mantener el nodo actualizado); y
- cuándo se detiene.

Por último, al momento de detenerse la subida, y ya actualizado el nodo correspondiente, una posibilidad es que este nodo tenga el rol de "pivote" en el proceso de rotación que habría que hacer a continuación. Como también vimos en clase, la rotación necesaria puede ser simple o doble:

- ¿cómo sabe el algoritmo cuál rotación hay que aplicar?

Respuestas

Examinar el nodo consiste en revisar el valor del balance del nodo.

El valor del balance puede ser –1, 0 o +1. Si es 0, entonces hay que cambiarlo a –1 o +1, según corresponda, y seguir subiendo; solo en este caso se sigue subiendo.

Si, en cambio, el valor del balance del nodo examinado x es –1 o +1, entonces hay dos posibilidades, dependiendo del hijo de x desde el cual, al subir, se llegó a x: se pudo haber llegado a x por el lado que corrige el –1 o +1, y lo cambia a 0; o se pudo haber llegado por el lado que aumenta el desbalance (dejándolo temporalmente en –2 o +2):

- El primer caso es cuando llegamos a x desde su hijo derecho y el balance de x es –1, o cuando llegamos a x desde su hijo izquierdo y el balance de x es +1. En este caso, solo hay que cambiar el balance de x a 0 (y no se sigue subiendo).
- El segundo caso es cuando llegamos a x desde su hijo derecho y el balance de x es +1, o cuando llegamos a x desde su hijo izquierdo y el balance de x es –1. Lo que ocurre aquí es que x queda desbalanceado según el criterio de balance AVL; x es el pivote que vimos en clase. En este caso, hay que efectuar un rebalanceo por la vía de una rotación, ya sea simple o doble (y tampoco se sigue subiendo).

Así, la "subida" se detiene cuando encuentra un nodo con balance –1 o +1. Entonces, se procede a actualizar el balance, tal como está explicado.

La "subida" también se detiene si se llega a la raíz.

Finalmente, en el caso en que hay que realizar una rotación (el "segundo caso"), para saber cuál es la rotación que hay que realizar, hay que recordar los dos últimos nodos examinados antes de llegar a x, es decir, el hijo y y el nieto z de x: si y y z son ambos hijos izquierdos o son ambos hijos derechos, entonces la rotación necesaria es simple; en cambio, si uno es hijo izquierdo y el otro es hijo derecho, entonces la rotación necesaria es doble.

2018-1-I1-P2–Heap (1/1)

2. Heaps

Tienes dos heaps: A, de tamaño m, y B, de tamaño n. Los heaps están almacenados explícitamente como árboles binarios y no como arreglos. Si los $m+n$ elementos son todos distintos, si $m = 2^k–1$ para algún entero k, y si $m/2 < n \leq m$, explica cómo construir un heap C con los elementos de $A \cup B$.

a) ¿Cuál es la profundidad del árbol A y cuál es la profundidad del árbol B ?

Respuesta

Si consideramos la profundidad de la raíz como 0, entonces la profundidad tanto de A como de B es k–1.

b) ¿Cuál va a ser la profundidad del árbol C ? Justifica.

Respuesta

Dado que tanto A como B tienen profundidad k–1, al unirlos en un nuevo árbol, C, mediante una raíz común, este nuevo árbol necesariamente tiene profundidad k.

También se puede argumentar que, en el extremo, es decir, si $n = m$, entonces el ábol resultante de la unión tiene $2^k–1 + 2^k–1 = 2^{k+1}–2$ elementos. Como $2^{k+1}–2 = (2^{k+1}–1)–1$, se trata de un árbol binario completo en todos sus niveles menos en el último (le falta un elemento), por lo que su profundidad es k.

c) Describe un algoritmo *eficiente* para la construcción de C. Explica cuál es la complejidad de tu algoritmo.

Respuesta

Saquemos el último elemento de B, es decir, el de más a la derecha en el nivel de más abajo; llamémoslo x. Creamos un nuevo árbol binario con x como raíz y A y B como subárboles. Todo esto es O(1). Finalmente, aplicamos *heapify* sobre x. Esta operación, como vimos en clase, es $O(k) = O(\log m)$.

2018-1-I2-P1–Árbol rojo-negro, Árbol 2-3 (1/2)

Estructuras de Datos y Algoritmos – IIC2133

Pauta I2

1. Árboles de búsqueda balanceados

a) Como vimos en clase, la inserción de un nodo en un árbol rojo-negro se puede dividir en tres casos; en todos los casos, el nodo *x* recién insertado se pinta inicialmente de rojo:

1) el padre *p* de *x* es negro à estamos listos

2) el padre *p* de *x* es rojo, pero el hermano *q* de *p* es negro (si *p* no tiene hermano, lo suponemos negro) à hacemos una o dos rotaciones y le cambiamos el color a dos nodos [**ustedes tienen que saber los detalles de este caso**]

3) el padre *p* de *x* es rojo, y el hermano *q* de *p* también es rojo (es decir, el nodo recién insertado, su padre y su tío son todos rojos) à como el abuelo *r* de *x* necesariamente es negro, cambiamos los colores de *r* y de sus hijos *p* y *q*; ahora *r* es rojo (y sus hijos *p* y *q* son negros), por lo que hay que revisar el color del padre *s* de *r* :

   s es negro à estamos listos

   s es rojo à repetimos el caso 2) o el caso 3), según corresponda, pero ahora con *s* en lugar de *p*, y el hermano *t* de *s* en lugar de *q*

El caso 3) significa que potencialmente podemos llegar de vuelta hasta la raíz del árbol. Una manera de evitar tener que hacer este recorrido "de vuelta" es ir preparando el árbol a medida que vamos bajando (desde la raíz, cuando estamos buscando el punto en que hay que hacer la inserción), de modo de garantizar que *q* no será rojo —algoritmo de inserción *top-down* :

> Al ir bajando, cuando encontramos un nodo *U* que tiene dos hijos rojos, cambiamos los colores de *U* (a rojo) y de sus hijos (a negro); esto producirá un problema solo si el padre *V* de *U* también es rojo, en cuyo caso aplicamos el caso 2) anterior.

Muestra la ejecución de este algoritmo de inserción *top-down* al insertar la clave 22 en el siguiente ár-bol rojo-negro: la raíz tiene la clave 35; sus hijos, las claves 30 (rojo) y 42 (negro); los dos hijos negros de 30 tienen las claves 25 y 33; los dos hijos rojos de 42 tienen las claves 40 y 45; y los dos hijos rojos de 25, tienen las claves 20 y 27.

Respuesta (estos son, más o menos, los pasos importantes)

Al bajar desde la raíz, con clave 35, no encontramos ningún caso especial hasta que llegamos al nodo negro con clave 25, que tiene ambos hijos rojos: 20 y 27. [**0.5 pts**]

Entonces aplicamos la nueva regla: cambiamos los colores de 25 a rojo y de sus dos hijos, 20 y 27, a negros. [**0.5 pts.**]

Como el padre de 25 también es rojo (el nodo con clave 30), aplicamos el caso 2: hacemos una rotación sim-ple a la derecha de la arista 35–30 —con lo que queda 30 como raíz (roja), con hijos 25 y 35 negros— e intercambiamos colores entre padre e hijos, dejando a 30 negro, y a 25 y 35 ambos rojos. [**1.5 pt.**]

2018-1-I2-P1–Árbol rojo-negro, Árbol 2-3 (2/2)

Ahora el nodo con clave 25 tiene ambos hijos —con claves 20 y 27— negros; y como 20 es una hoja, inserta-mos allí el nuevo nodo con clave 22, que pintamos de rojo. [**0.5 pts.**]

b) Determina un orden en que hay que insertar las claves 1, 3, 5, 8, 13, 18, 19 y 24 en un árbol 2-3 inicialmente vacío para que el resultado sea un árbol de altura 1, es decir, una raíz y sus hijos.

Respuesta

Un árbol 2-3 con una raíz y sus hijos tiene a lo más 4 nodos y puede almacenar a lo más 8 claves (dos claves por nodo). Como son exactamente 8 claves las que queremos almacenar, éstas tiene que quedar almacena-das de la siguiente manera: la raíz tiene las claves 5 y 18; el hijo izquierdo, 1 y 3; el hijo del medio, 8 y 13; y el hijo derecho, 19 y 24. [**1 pt.**]

Para lograr esta configuración final hay varias posibilidades; aquí vamos a ver una. [**2 pts.**]

Podemos insertar primero las claves 1, 5 y 13, en cualquier orden, con lo cual queda 5 en la raíz, y 1 y 13 como hijos izquierdo y derecho. (En vez de 1 puede ser 3 y en vez de 13 puede ser 8. Por otra parte, en vez de empezar con 1, 5 y 13, es decir, empezar "por la izquierda", podríamos empezar por la derecha con 8-13, 18 y 19-24.) A partir de ahora, podemos insertar 3 en cualquier momento.

Ahora tenemos que conseguir que 18 quede en la raíz, junto con 5. Insertamos 18, que va a acompañar a 13, y a continuación insertamos 24, que va al mismo nodo de 13 y 18; como este nodo tiene ahora tres claves —13, 18 y 24 (lo que no puede ser)— lo separamos en dos nodos con las claves 13 y 24, respectivamente, y subimos la clave 18. Ahora podemos insertar 8 y 19, en cualquier orden.

2018-1-Ex-P3–ABB (1/1)

3. ***Treaps***

Considera un árbol binario de búsqueda (no necesariamente balanceado). Considera que los nodos de este árbol, además de tener una clave, tienen una prioridad, de modo que el árbol está ordenado según las claves de los nodos (como todo árbol binario de búsqueda), pero ahora, además, las prioridades de los nodos satisfacen la propiedad de min-heap. Es decir, si un nodo tiene clave k y prioridad q, entonces los nodos que están en su subárbol izquierdo tienen claves menores que k y los nodos que están en el subárbol derecho tienen claves mayores que k; y, además, las prioridades de los hijos de este nodo son mayores que q.

a) Describe un algoritmo eficiente para insertar un nodo con clave k y prioridad q, y analiza su complejidad.

Respuesta: Se inserta el elemento normalmente como en un árbol binario de búsqueda. Luego se hacen rotaciones subiendo el elemento hasta que su prioridad sea mayor a la de su padre **[3ptos]**.

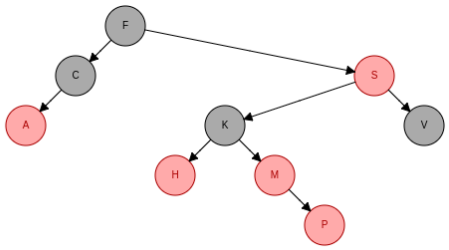
b) Describe un algoritmo eficiente para eliminar un nodo con clave k, y analiza su complejidad.

Respuesta: Si el elemento eliminado es una hoja no se hace nada especial **[0.5pts]**. Si tiene 1 hijo simplemente se remplaza por su hijo **[0.5pts]**. Si tiene 2 hijos se remplaza por el sucesor o el antecesor y luego se hacen rotaciones hasta que cumple la propiedad de heap **[2pts]**.

2017-2-I1-P5–Árbol rojo-negro (1/2)

5 Pregunta 5

El árbol de la figura es un **árbol rojo negro** al cual se le acaba de insertar la clave P.



a) **[0.5 pts]** ¿De qué color está pintado el nodo con clave P? ¿Cómo sabemos que es el color?

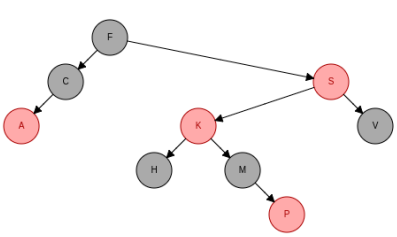
[0.5 pts] Por decir que es rojo e indicar que es por las reglas de inserción (un nodo siempre se inserta pintado de rojo).
También se da puntaje si indican que, considerando que el árbol antes de la inserción de P estaba balanceado, la raíz siempre debe ser negra y como P tiene un color distinto a la raíz, entonces debe ser rojo.

b) **[1 pts]** ¿Qué propiedad de árbol rojo-negro está siendo violada en estas condiciones?
[1 pts] Por indicar que es la propiedad que todo nodo de color rojo debe tener únicamente hijos de color rojo.l

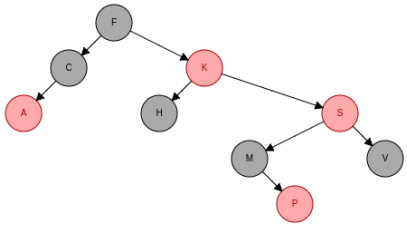
c) **[1 pts]** ¿Cuáles dos tipos de operaciones se pueden ejecutar para restaurar la condición de árbol rojo-negro?
[1 pts] Por indicar que se puede hacer una recoloración y luego una rotación doble.

d) **[3.5 pts]** Ejecuta una a una las operaciones necesarias hasta restaurar la condición del árbol rojo-negro. A continuación se muestra el resultado de aplicar la operación indicada hasta llegar al árbol balanceado:

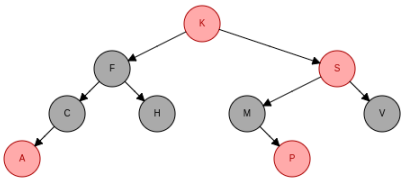
2017-2-I1-P5–Árbol rojo-negro (2/2)



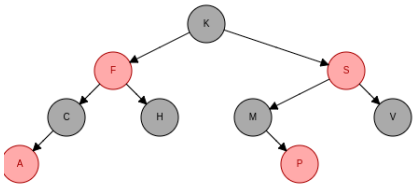
(a) Recoloración



(b) Rotación a la derecha



(c) Rotación a la Izquierda



(d) Recoloración

Figura 1: Aplicación de las operaciones necesarias para balancear el árbol

- **[0.75 pts]** Por indicar que se necesita una recoloración gracias al tío rojo de P.
- **[1 pts]** Por indicar que se necesita una rotación a la derecha ya que el tío de K es negro, con K rojo siendo el hijo derecho de un nodo rojo izquierdo.
- **[1 pts]** Por indicar que se necesita una rotación a la izquierda, ya que C es el tío negro de S y S es el hijo rojo derecho de K rojo.
- **[0.75 pts]** Por indicar que se necesita una recoloración por raíz roja.

Si hacen bien la primera recoloración, luego indican donde ocurre el nuevo conflicto y cómo solucionarlo, pero dibujan incorrectamente el árbol, entonces reciben **1.75 pts**.

2017-1-I2-P1–AVL (sin enunciado) (1/2)

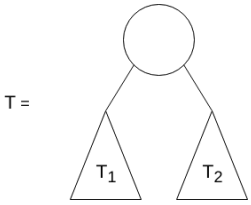
PONTIFICIA UNIVERSIDAD CATÓLICA DE CHILE
ESCUELA DE INGENIERÍA
DEPARTAMENTO DE CIENCIA DE LA COMPUTACIÓN

IIC 2133 — Estructuras de Datos y Algoritmos
Interrogación 2
Primer Semestre, 2017

Duración: 2 hrs.

1. a. (3 puntos) Ejemplo de solución correcta y su distribución de puntaje:

Por definición de AVL, las alturas de ambos subárboles no pueden diferir en más de 1 (0,25).
Sea T un AVL:



La altura h(T) está dada por

$$h\left(T\right) = 1 + max\left(h\left(T_1\right), h\left(T_2\right)\right)$$

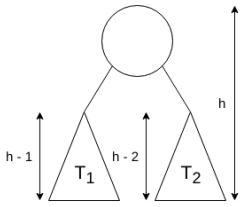
Pero por definición de AVL

$$|h\left(T_1\right) - h\left(T_2\right)| \leq 1 \quad (0{,}25)$$

En el peor caso, la igualdad es estricta.

$$|h\left(T_1\right) - h\left(T_2\right)| = 1 \quad (0{,}5)$$

Asumiendo que T_1 es más alto, $h(T_1) = h(T_2) + 1, \quad h(T) = h(T_1) + 1 = h(T_2) + 2$



Sea $N(h)$ la cantidad de nodos mínima para un árbol de altura h Podemos definirla en función de las alturas de sus subárboles:

$$N(h) = 1 + N(h-1) + N(h-2), \quad N(0) = 1 \quad (0{,}5)$$

2017-1-I2-P1–AVL (sin enunciado) (2/2)

Además se cumple que a mayor altura, más nodos. Esto significa que $N(h-1) > N(h-2)$ Luego,

$$\begin{aligned}
N(h) &> 1 + N(h-2) + N(h-1) > 2N(h-2) \quad (0,5) \\
&> 2(1 + N(h-3) + N(h-4)) \\
&> 2^2 N(h-4) \\
&> 2^2(1 + N(h-5) + N(h-6)) \\
&> 2^3 N(h-6) \\
&> 2^3(1 + N(h-7) + N(h-8)) \\
&> 2^4 N(h-8) \\
&\vdots \\
&> 2^i N(h-2i) \quad (0,5) \\
&\text{Luego, para } i = \frac{h}{2} \\
&> 2^{h/2} N(0)
\end{aligned} \tag{1}$$

Luego, como $N(0) = 1$ queda que $N(h) > 2^{h/2}$. Aplicamos logaritmo en base dos:

$$\begin{aligned}
log_2(n) &> \frac{h}{2} \\
h \in O(log(n)) &\quad (0{,}5)
\end{aligned}$$

b. (3 puntos) La demostración de esta pregunta fue vista en clases. En esta página hay un ejemplo de explicación:

http://www.mathcs.emory.edu/ cheung/Courses/323/Syllabus/Trees/AVL-insert.html

El puntaje se asignó:

(1) Explicar qué ocurre en un desbalanceo: diferencia de 2, 4 posibles configuraciones, etc.
(1) Dar a entender un caso general de desbalanceo (o cada caso), enseñando valores de alturas respectivas y detalles de cada nodo que forman parte de la rama desbalanceada.
(1) Mostrar qué ocurre al balancear (en un solo paso) y calcular los nuevos valores de altura. Explicar por qué se balancea.

2017-1-I2-P3–Árbol 2-3, Árbol 3-4 (sin enunciado) (1/2)

3. *a*) Primero deben dar un ejemplo de un árbol 2-3-4 que en el cuál al eliminar un nodo, se producen dos fusiones. Las fusiones ocurren cuando al eliminar un nodo todos sus hermanos son nodo 2, es decir, tienen solo una clave. Esto, según el algoritmo visto en clases, hace que no puedan, los hermanos del nodo eliminado, prestarle una llave al espacio dejado por el nodo eliminado. Debido a esto, se produce una fusión entre el padre del nodo eliminado y algún hermano (bajando una clave del padre para unirse con la del hermano). Luego si el padre se queda sin claves, y sus hermanos también son nodos dos, ocurre la segunda fusión.

   Mostrar el árbol con las fusiones es **1 punto**.

   Dibujar su correspondiente árbol rojo-negro (dado que se hizo bien el árbol 2-3-4) con su correspondiente eliminación de nodo es **1 punto**. Es importante que el árbol rojo-negro ïmite"de alguna forma lo que está

2017-1-I2-P3–Árbol 2-3, Árbol 3-4 (sin enunciado) (2/2)

asiendo el árbol 2-3-4 para que siga siendo su árbol asociado.

Finalmente, se debe hacer la relación de la fusión en el árbol 2-3-4 con lo que ocurre en el árbol rojo-negro. Es importante notar que una fusión corresponderá a un cambio de color en el árbol rojo-negro. Esto es porque un padre (que tenía solo una clave, es decir, que era negro) pasa a tener dos claves por haber sido unido con el hermano del nodo eliminado, por lo que se le cambia el color a rojo. (**2 puntos**).

b) En esta pueden haber varias soluciones dependiendo de cómo definieron el peor caso. Una solución era la siguiente:

- Peor caso: cuando la inserción produce mayor cantidad de rotaciones y de recoloraciones. (**1 punto**).
- Familia: Este es el caso de cuando al insertar un nodo, su padre será rojo y su tío también lo será. Y hacia arriba, los colores se van intercalando. Esto, al insertar un nodo, generará la mayor cantidad de cambios de color (hasta la raíz). (**1 punto**).

2017-1-Ex-P1-d—f—g—h—k–Árbol rojo-negro, AVL, ABB (1/1)



PONTIFICIA UNIVERSIDAD CATÓLICA DE CHILE
ESCUELA DE INGENIERÍA
DEPARTAMENTO DE CIENCIA DE LA COMPUTACIÓN

IIC 2133 — Estructuras de Datos y Algoritmos
Examen
Primer Semestre, 2017

Duración: 3 hrs.

1. Para cada una de las siguientes afirmaciones, diga si es *verdadera* o *falsa*, **siempre justificando** su respuesta.

   a) Quick Sort es un algoritmo de ordenación estable. **Respuesta:** Falso. Basta con dar un contraejemplo.

   b) Sea e la segunda arista más barata de un grafo dirigido acíclico G con más de dos nodos y aristas con costos diferentes. Entonces e pertenece al árbol de cobertura de costo mínimo para G. **Respuesta:** Verdadero. Si usamos el algoritmo de Kruskal, la segunda arista más barata es siempre agregada al MST puesto que no puede formar un ciclo en el bosque construido hasta el momento.

   c) Radix Sort es un algoritmo de ordenación que puede ordenar n números enteros y cuyo tiempo de ejecución está siempre en $\Theta(n)$. **Respuesta:** El tiempo de ejecución de Radix Sort es $d(n + k)$, cuando los datos están en $[0, k]$. Basta entonces con que k sea suficientemente grande (por ejemplo, exponencial en n), para que el algoritmo no sea $\Theta(n)$

   d) Sea A un árbol rojo-negro en donde cada rama tiene n nodos negros y n nodos rojos. Si al insertar una clave nueva en A se obtiene el árbol A', entonces cada rama de A' tiene $n + 1$ nodos negros. **Respuesta:** Verdadero. Un árbol rojo negro como A corresponde a un árbol 2-4 "completamente saturado"; es decir, uno que contiene nodos con 3 claves. Al insertar una clave nueva, el árbol 2-4 aumentará su altura en 1. Esto significa que su equivalente rojo-negro debe tener un nodo negro más por cada rama.

   e) Si se tienen dos tablas de hash, una con direccionamiento abierto y otra cerrado, las dos del mismo tamaño m y el mismo factor de carga α, ambas ocupan la misma cantidad de memoria.

   f) Sea A un árbol AVL y ℓ_1 y ℓ_2 los largos de dos ramas de A. Entonces $|\ell_1 - \ell_2| \leq 1$. **Respuesta:** Falso. Basta dar un contraejemplo

   g) La operación de inserción en un árbol AVL con n datos realiza a lo más una operación *restructure* pero toma tiempo $O(\log n)$. **Respuesta:** Verdadero, puesto que la inserción debe revisar que el balance esté correcto a lo largo de la rama donde se insertó el dato. Y esa rama tiene tamaño $O(\log n)$.

   h) Si A es un ABB y n es un nodo de A, el *sucesor* de n no tiene un hijo izquierdo. **Respuesta:** Falso. La propiedad no se cumple en general cuando n no tiene un hijo derecho.

   i) Es posible modificar la implementación de la estructura de datos para conjuntos disjuntos vista en clases, de manera que permita des-unir dos conjuntos en $O(1)$.

   j) Como diccionario, una tabla de hash es más conveniente que un árbol rojo negro en cualquier aplicación. (Sin considerar la dificultad de implementación).

   k) Sea p una secuencia de dos o más números diferentes y sea q una permutación de p distinta de p. Adicionalmente, sean A_p y A_q los árboles binarios de búsqueda que resultan de, respectivamente, insertar en orden los elementos de p y q en árboles binarios de búsqueda vacíos. Entonces A_p y A_q son distintos. **Respuesta:** Falso.

   l) Re-hashing, el procedimiento que construye una nueva tabla de hash a partir de otra existente, toma tiempo $O(n)$ en una tabla con colisiones resueltas por encadenamiento de tamaño m que contiene n datos. **Respuesta:** Falso. El tiempo depende del tamaño de la tabla y del número de datos; específicamente es $O(m + n)$.

2016-2-I1-P2–Árbol 2-3 (1/1)

2. Describe un algoritmo para eliminar una clave de un árbol 2-3; el algoritmo recibe como parámetros un puntero a la raíz del árbol y la clave que se va a eliminar. Por supuesto, tu algoritmo debe dejar como resultado un árbol 2-3. (En este problema, el algoritmo se puede describir en prosa y con dibujos.)

La idea general es primero encontrar la clave en el árbol [**a**) describe esta parte del algoritmo en términos de pasos más básicos: **1**].

Si la clave está en una hoja, entonces simplemente la eliminamos.

Si la clave está en la raíz o en un nodo interior, entonces la reemplazamos por la clave predecesora o la clave sucesora [**b**) describe esta parte del algoritmo en términos de pasos más básicos: **2**],

… y eliminamos (recursivamente) esta clave del árbol (hasta llegar a una hoja).

Si la hoja es un nodo 3, entonces estamos listos;

… si es un nodo 2, entonces hay que ser más creativos: considera que se produjo un "hoyo" en árbol y hay que hacerlo subir por el árbol hasta que pueda ser eliminado [**c**) describe esta parte del algoritmo en términos de pasos más básicos: **3**].

Respuestas:

a) Buscamos la clave en el nodo que estamos mirando; el primer nodo que miramos es obviamente la raíz del árbol. Si encontramos la clave en este nodo, entonces pasamos a **b**); de lo contrario, tenemos que descender a uno de los hijos del nodo y, recursivamente, buscar ahí. En este caso, usamos el hecho de que las claves del nodo están ordenadas para decidir a cuál hijo tenemos que descender: si la clave buscada es menor que la menor de las claves del nodo, entonces bajamos al hijo izquierdo del nodo; si la clave buscada es mayor que la mayor de las claves del nodo, entonces bajamos al hijo derecho del nodo; si la clave buscada está entre la menor y la mayor claves del nodo (en un nodo 3), entonces bajamos al hijo del medio.

b) Como es un nodo interior, entonces la clave predecesora (o la sucesora, se puede elegir cualquiera) está siempre en una hoja, más abajo en el árbol: la clave predecesora/sucesora es la más grande/pequeña de las claves almacenadas en el subárbol izquierdo/derecho (respecto de la clave que estamos eliminando). Para encontrarla (supongamos que buscamos la predecesora), bajamos a la raíz del subárbol correspondiente; si es una hoja, entonces es la clave más a la derecha en la hoja; de lo contrario, bajamos por el puntero más a la derecha del nodo y buscamos recursivamente ahí. Esta clave predecesora la ponemos en el lugar de la clave que queremos eliminar, sacándola de la hoja en la que estaba.

c) Si el "hoyo" es hijo de un nodo 2, con clave X, y tiene como hermano un nodo 2, con clave Y, entonces restructuramos este subárbol de la siguiente manera: ponemos el "hoyo" en el lugar en que está X y hacemos que X y Y formen un nodo 3 hijo del "hoyo"; con esto, el "hoyo" sube un nivel y hay que lidiar con él en este nuevo nivel.

Si el "hoyo" es hijo de un nodo 2, con clave X, y tiene como hermano un nodo 3, con claves Y y Z, entonces restructuramos así: Y queda en el lugar de X, X queda en el lugar del "hoyo", y uno de los hijos del nodo 3 original pasa a ser hijo de X. En este caso, el "hoyo" es efectivamente eliminado (no sube).

Si el "hoyo" es hijo de un nodo 3 y su hermano inmediato —a su derecha o su izqueirda— es un nodo 2 (el otro hermano puede ser un nodo 2 o un nodo 3), entonces cambiamos el padre por un nodo 2 (dejamos en el padre sólo una de las dos claves originales) y bajamos la otra clave, que pasa a formar un nodo 3 con el hermano inmediato del "hoyo". El "hoyo" es eliminado.

2016-2-I1-P3–ABB, AVL, Árbol 2-3 (1/3)

3. Considera un **árbol de búsqueda** inicialmente vacío; y considera las siguientes 9 letras como claves a ser insertadas en el árbol: A, C, E, H, L, M, P, R y S. Ejecuta la inserción, letra por letra y en el orden dado, para cada uno de los siguientes tipos de árbol:

a) [**1**] Un árbol de búsqueda binario sin propiedades de balance.

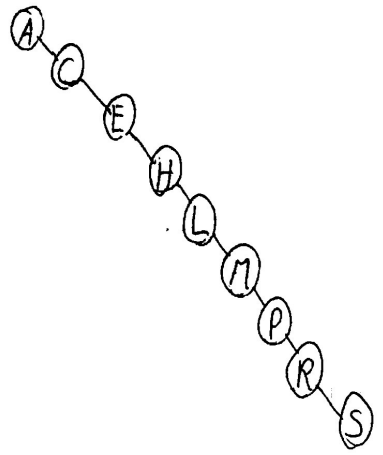
b) [**2.5**] Un árbol AVL.

c) [**2.5**] Un árbol 2-3.

Respuesta: al final del pdf.

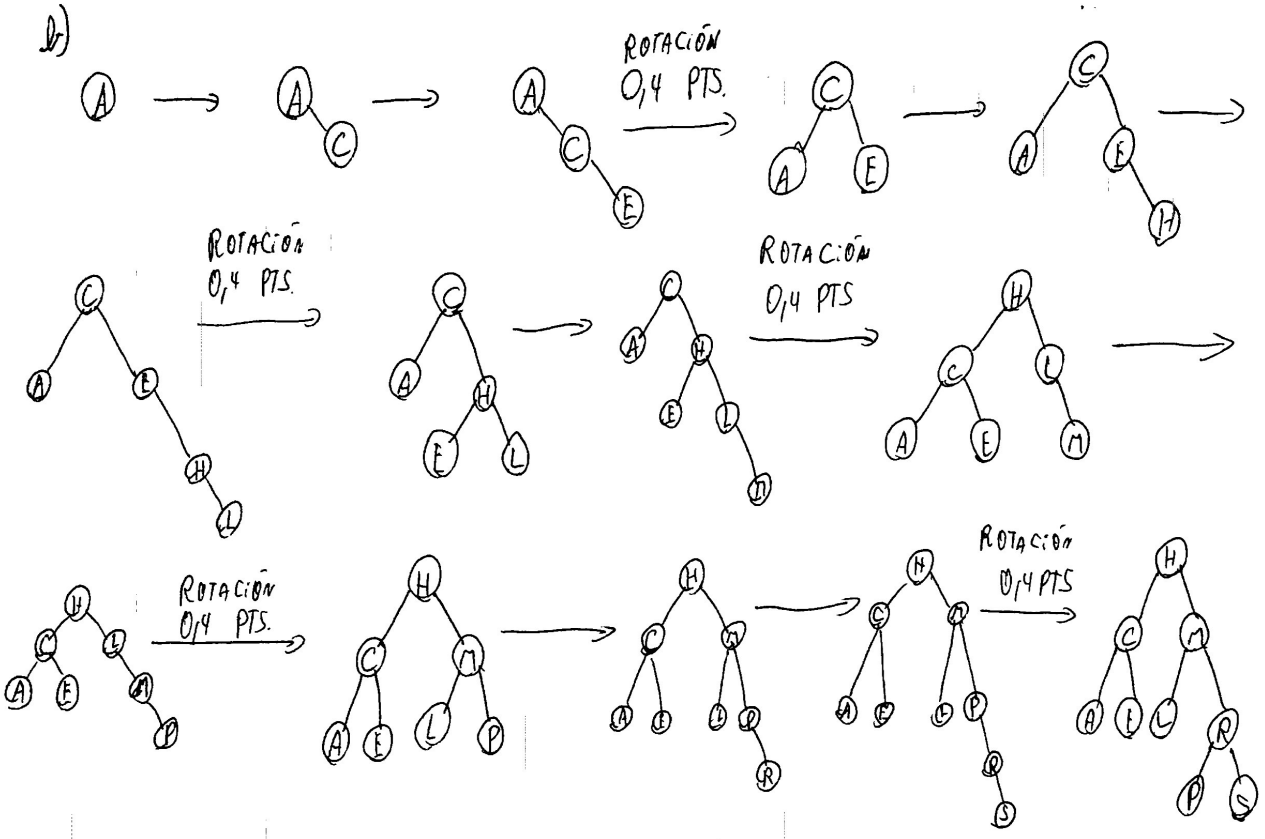
2016-2-I1-P3–ABB, AVL, Árbol 2-3 (2/3)

a)



1 PUNTO POR DEJAR EL ÁRBOL ASÍ.
(NO IMPORTA SI SE SALTAN LOS PASOS DE INSERCIÓN)

b)



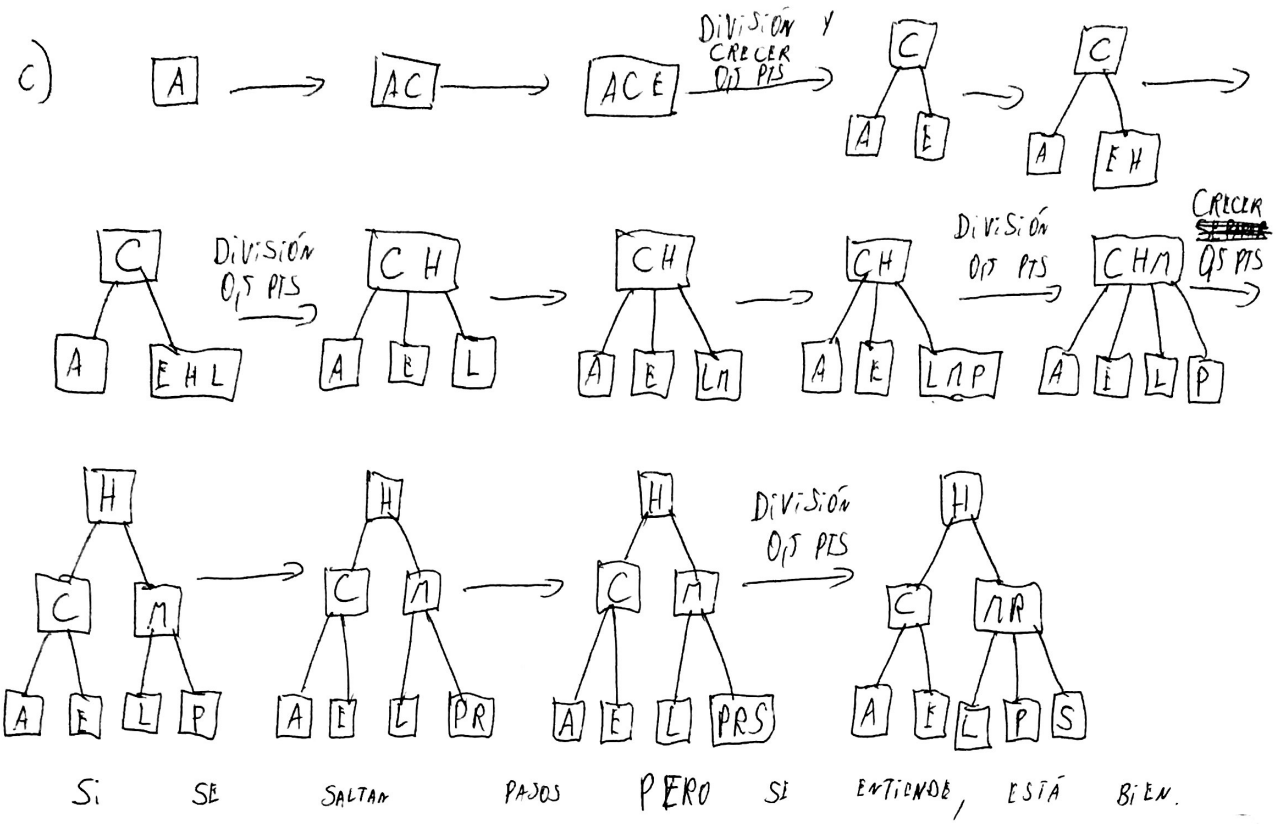
$0{,}4 \cdot 5 + 0{,}5 = 2{,}5$ PTS.

$0{,}4 \cdot 5$: ROTACIONES

$0{,}5$: HACER TODAS LAS INSERCIONAS EN LAS HOJAS CORRESPONDIENTES

SE PUEDEN SALTAR PASOS MIENTRAS QUEDE CLARO COMO LE HACEN LAS ROTACIONES INSERCIONES

2016-2-I1-P3–ABB, AVL, Árbol 2-3 (3/3)



2016-2-I1-P4-b–Min-Heap (1/1)

b) *Min-heaps* binarios:

i) ¿Cuál es el número máximo de comparaciones que es necesario realizar para encontrar la clave más grande almacenada en un *min-heap* binario? Justifica

ii) Describe un algoritmo para encontrar todas las claves menores que algún valor, X, almacenadas en un *min-heap* binario? Tu algoritmo debe correr en tiempo $O(K)$, en que K es el número de claves que cumplen la condición.

Respuestas:

i) La clave más grande sólo puede estar en una hoja del *min-heap* (mirado como árbol binario), ya que para cualquier nodo que tenga hijos, las claves de los hijos son mayores que la clave del nodo. Las claves en la hojas no tienen ningún orden particular, por lo que hay que mirarlas todas; en un *min-heap* con n claves, hay $\lceil n/2 \rceil$ hojas.

ii) Si la raíz es menor que X, entonces primero imprime la raíz, luego baja recursivamente al hijo izquierdo, y luego baja recursivamente al hijo derecho. Si el nodo que está mirando es mayor que X, entonces no hace nada con ese nodo.

2016-2-I2-P2–Árbol rojo-negro (1/1)

2. *Otro algoritmo de inserción en árboles rojo-negros*. En lugar de primero insertar el nuevo nodo, pintarlo de rojo y después hacer rotaciones y/o cambios de color (*hacia arriba*) para restaurar las propiedades del árbol, podemos ir haciendo los ajustes a medida que vamos *bajando* por el árbol hacia el punto de inserción, de modo que cuando insertamos el nuevo nodo simplemente lo pintamos de rojo y sabemos que su padre es negro. El procedimiento es el siguiente.

> Mientras bajamos por el árbol, cuando *en la ruta de inserción* vemos un nodo *X* que tiene dos hijos rojos, intercambiamos colores: pintamos *X* de rojo y sus hijos de negro. Esto producirá un problema sólo si el padre *P* de *X* es rojo. Pero en ese caso, simplemente aplicamos las rotaciones apropiadas, que se ilustran en las figs. A y B adjuntas.

a) Muestra cómo opera este nuevo algoritmo de inserción cuando insertamos un nodo con la clave **45** en el árbol de la fig. C adjunta; específicamente, muestra lo que ocurre a medida que llegas a cada nivel.

Respuesta:

Cuando se baja de la raíz 30 a su hijo derecho 70 no pasa nada, ya que 70 (el *X* del algoritmo) tiene sólo un hijo rojo, 60. Cuando se baja de 70 a 60 tampoco pasa nada, ya que 60 (el *X* del algoritmo) es rojo. Cuando se baja de 60 a 50 estamos en el caso descrito en el algoritmo: 50 (el *X* del algoritmo) es negro y sus dos hijos, 40 y 55, son rojos. [*Hasta aquí no hemos cambiado nada, sólo bajamos por el árbol*: **0.5 pts.**]

Entonces, cambiamos colores: pintamos a 50 de rojo y a sus hijos 40 y 55 de negro. Pero ahora 50 y su padre 60 son rojos; como el hermano 85 de 60 es negro, estamos en la situación descrita en la fig. A: 50 es *X*, 60 es *P*, 70 es *G*, 85 es *S*. [**1 pt.**]

Aplicando la rotación y cambios de colores sugeridos en la fig. A, queda 60 de negro como nuevo hijo derecho de 30 y con dos hijos rojos, 50 y 70, que en total tienen cuatro hijos negros, 40, 55, 65 y 85 (este último mantiene sus dos hijos rojos, 80 y 90, aunque ahora los tres están un nivel más abajo). [**1 pt.**]

Nosotros seguimos en 50, que ahora es rojo y está un nivel más arriba, así como sus hijos, 40 y 55, ahora negros. Bajamos a 40, que es una hoja negra, y por lo tanto insertamos 45 como hijo derecho de 40 y lo pintamos rojo. [**0.5 pts.**]

b) Los casos ilustrados en las figs. A y B se producen cuando el hermano *S* del padre *P* de *X* es negro. Explica claramente qué pasa con el caso en que *S* es rojo *en este nuevo algoritmo de inserción*.

Respuesta:

Este caso en realidad *no se produce*, debido a los ajustes que vamos haciendo a medida que bajamos por el árbol. Si es que encontramos un nodo *Y* con dos hijos rojos, sabemos que sus nietos tienen que ser todos negros; y como también pintamos de negro los hijos de *Y*, entonces incluso después de las posibles rotaciones no vamos a encontrar otro nodo rojo en los próximos dos niveles.

2016-1-I1-P2–ABB, Árbol 2-3 (1/5)

2. **a**) [**2**] Supongamos que la búsqueda de una clave k en un **árbol de búsqueda binario** termina en una hoja. Consideremos tres conjuntos: A, las claves a la izquierda de la ruta de búsqueda; B, las claves en la ruta de búsqueda; y C, las claves a la derecha de la ruta de búsqueda. Tomemos tres claves: $a \in A$, $b \in B$ y $c \in C$. ¿Es cierto o es falso que $a < b < c$? Justifica.

Es falso y basta mostrar un contraejemplo.

b) [**2**] Dibuja un **árbol AVL**, preferiblemente pequeño, tal que las tres próximas inserciones, sin mediar eliminaciones, produzcan desbalances que deban corregirse con rotaciones dobles. Muestra el árbol inicial y, para cada inserción, muestra el desbalance, identifica el pivote y corrige el desbalance.

Ver archivo "I1-1-2016-pauta-p2b". La idea es que el árbol inicialmente tiene que tener nodos con balances –1 o +1, balance que después de la inserción queda –2 o +2. Para que la rotación necesaria sea doble, el nodo insertado tiene que ser un hijo izquierdo, si se insertó en el subárbol derecho del que va a ser finalmente el nodo pivote, o vice versa.

c) [**2**] Determina un orden en que hay que insertar las claves 1, 3, 5, 8, 13, 18, 19 y 24 en un **árbol 2-3** inicialmente vacío para que el resultado sea un árbol de altura 1, es decir, una raíz y sus hijos.

Un árbol 2-3 con una raíz y sus hijos tiene a lo más 4 nodos y puede almacenar a lo más 8 claves (dos claves por nodo). Como son exactamente 8 claves las que queremos almacenar, éstas tiene que quedar almacenadas de la siguiente manera: la raíz tiene las claves 5 y 18; el hijo izquierdo, 1 y 3; el hijo del medio, 8 y 13; y el hijo derecho, 19 y 24. Para lograr esta configuración final hay varias posibilidades; aquí vamos a ver una.

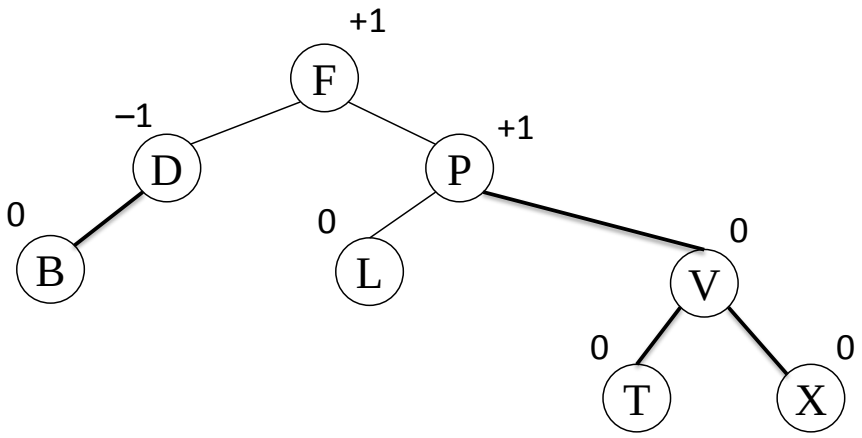
Podemos insertar primero las claves 1, 5 y 13, en cualquier orden, con lo cual queda 5 en la raíz, y 1 y 13 como hijos izquierdo y derecho. (En vez de 1 puede ser 3 y en vez de 13 puede ser 8. Por otra parte, en vez de empezar con 1, 5 y 13, es decir, empezar "por la izquierda", podríamos empezar por la derecha con 8-13, 18 y 19-24.) A partir de ahora, podemos insertar 3 en cualquier momento.

Ahora tenemos que conseguir que 18 quede en la raíz, junto con 5. Insertamos 18, que va a acompañar a 13, y a continuación insertamos 24, que va al mismo nodo de 13 y 18; como este nodo tiene ahora tres claves —13, 18 y 24 (lo que no puede ser)— lo separamos en dos nodos con las claves 13 y 24, respectivamente, y subimos la clave 18. Ahora podemos insertar 8 y 19, en cualquier orden.

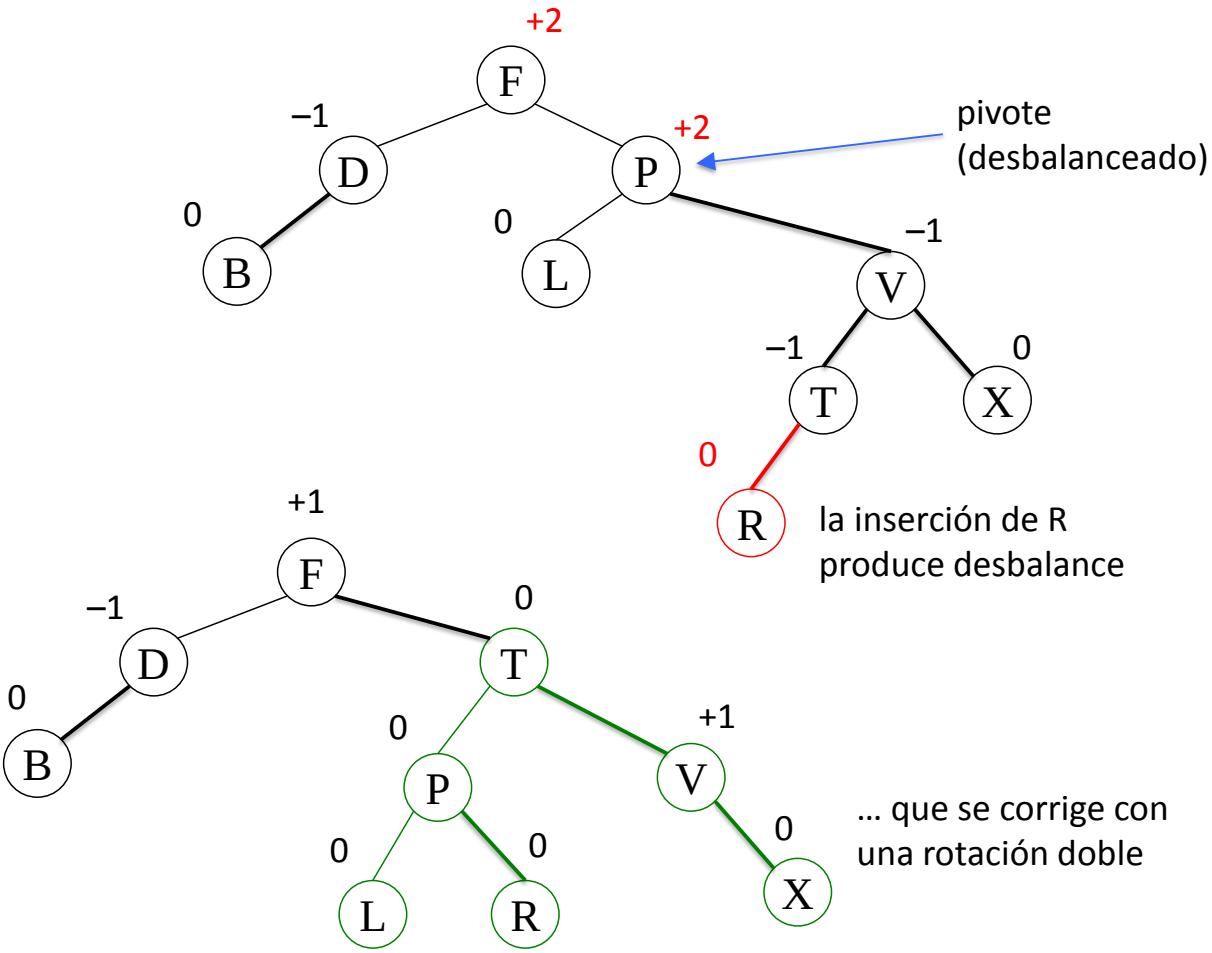
2016-1-I1-P2–ABB, Árbol 2-3 (2/5)

I1 (1-2016)
2b)

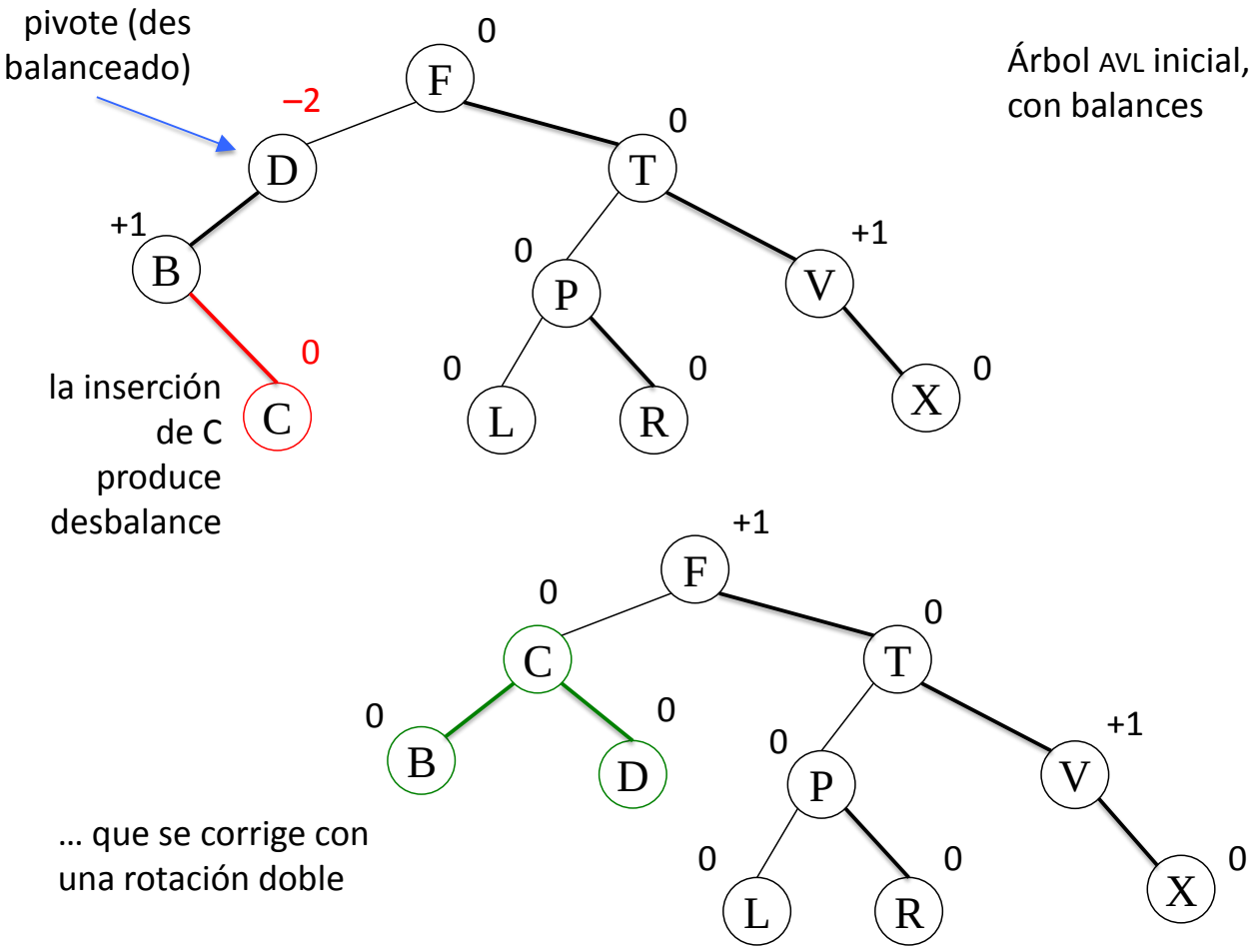
Árbol AVL inicial,
con balances



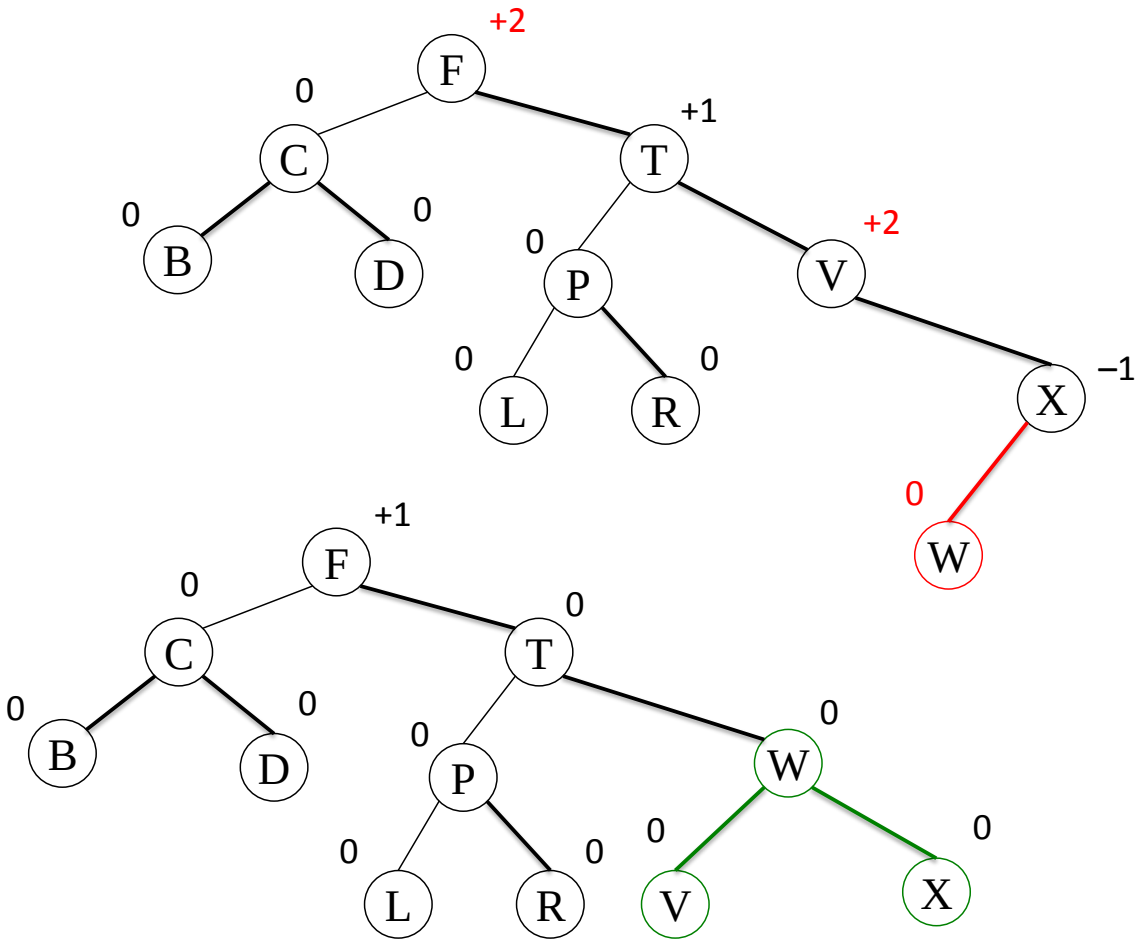
2016-1-I1-P2–ABB, Árbol 2-3 (3/5)



2016-1-I1-P2–ABB, Árbol 2-3 (4/5)



2016-1-I1-P2–ABB, Árbol 2-3 (5/5)



2016-1-I1-P3–Max-Heap (1/2)

3. **a**) [**3**] En clase estudiamos los procedimientos **up** y **down** que operan sobre ***maxHeaps***; su propósito es restaurar la propiedad de *maxHeap* cuando el valor de una clave (o prioridad) almacenada en el heap aumenta o disminuye, respectivamente. Una forma de construir un *maxHeap* a partir de las claves almacenadas en un arreglo **q** —con n claves, en las posiciones **q**[1] hasta **q**[n]— es ir insertando las claves una a una en un heap inicialmente vacío, usando el procedimiento **insert** estudiado en clase, que a su vez hace uso del procedimiento **up**; llamémosla la forma U.

Una segunda forma —llamémosla D— es aplicar el siguiente algoritmo al propio arreglo **q**:

```
k = n/2
while k ≥ 1:
   down(k)
   k = k-1
```

Compara las formas U y D de construir el *maxHeap*. En particular, **i**) calcula detalladamente el número de operaciones —básicamente, comparaciones e intercambios— que cada una realiza en el peor caso; y **ii**) demuestra que ambas construyen el mismo *maxHeap*, o bien da un ejemplo que muestre que en general construyen *maxHeaps* distintos (para un mismo arreglo **q**).

i) La forma U requiere $O(n\log n)$ operaciones; la forma D, sólo $O(n)$.

En U, insertamos la primera clave en un heap vacío, la segunda, en uno con una clave, la tercera, en uno con dos claves, …, la k-ésima, en uno con k–1 claves, …, y la n-ésima, en uno con n–1 claves. En cada inserción hay que comparar la clave que estamos insertando con la clave del padre y, si corresponde, intercambiar ambas claves; y así sucesivamente hasta llegar a la raíz, haciendo un número constante de operaciones —una comparación y a lo más un intercambio— por nivel. Es decir, el número de operaciones es

$T_U(n) = c_U(\lceil\log 1\rceil + \lceil\log 2\rceil + \ldots + \lceil\log n\rceil) = O(n\log n)$

En D, consideramos que las $n/2$ claves en la segunda mitad de `q` forman un heap c/u, de tamaño 1, por lo que inicialmente las "saltamos" y procesamos las $n/2$ claves en la primera mitad de "atrás para adelante" (por eso, `k = n/2` antes de entrar al `while`, y `k = k-1` antes de salir). El procesamiento consiste en comparar la clave en la posición k con las claves hijas, en las posiciones $2k$ y $2k$+1, y, si corresponde, intercambiarla con la mayor de las hijas, en cuyo caso repetimos estas acciones con respecto a las nuevas hijas de la clave (si las hay). Así, procesar una clave en un nivel toma un número constante de operaciones: a lo más 3 comparaciones y un intercambio. El número total de operaciones de D sale de notar que $n/4$ claves hay que procesarlas en sólo un nivel, $n/8$ claves en a lo más dos niveles, $n/16$ claves en a lo más 3 niveles, etc.:

$T_D(n) = c_D(\, n/4 + 2(n/8) + 3(n/16) + 4(n/32) + \ldots + (k-1)(n/2^k) + \ldots + (\log n - 1)\,) = O(n)$

ii) Ejemplo de que U y D construyen *maxHeaps* **distintos**: Si `q` = [1, 2, 3], entonces U construye un heap con 3 en la raíz, 1 como hijo izquierdo y 2 como hijo derecho; en cambio, D construye uno con 3 en la raíz (la raíz tiene que tener la misma clave en ambos casos), 2 como hijo izquierdo y 1 como hijo derecho.

2016-1-I1-P3–Max-Heap (2/2)

b) [**2**] El ítem más grande en un ***maxHeap*** está en la posición 1 del arreglo, y el segundo más grande está en la posición 2 o 3. Para un *maxHeap* de tamaño 31, da la lista de posiciones en donde **i) podría** estar el k-ésimo ítem más grande, y **ii) no podría** estar el k-ésimo ítem más grande; para $k = 2$, 3 y 4, suponiendo que los valores son distintos. Justifica.

i) La regla del *maxHeap* es que el ítem en la posición k es más grande que los ítemes en las posiciones $2k$ y $2k+1$; o, mirado desde otro punto de vista, dado un ítem, sólo sabemos que es más grande que sus hijos y **más pequeño que su padre**.

Esta última observación significa que los ítemes en una misma rama crecen en valor si recorremos la rama desde la hoja hacia la raíz. Por lo tanto, el segundo ítem más grande en el heap es uno que a lo más está a distancia 1 de la raíz (sólo puede haber un ítem más grande que el segundo más grande y ése tiene que estar en la raíz); es decir, las posiciones 2 o 3.

El tercer ítem más grande está a lo más a distancia 2 de la raíz; es decir, las posiciones 2, 3, 4, 5, 6 y 7. En particular, si el segundo ítem más grande está en la posición 2, entonces el tercero más grande puede estar en las posiciones 3, 4 o 5; y si el segundo está en la posición 3, entonces el tercero puede estar en las posiciones 2, 6 o 7.

Y el cuarto ítem más grande va a estar a lo más a una distancia 3 de la raíz; es decir, las posiciones 2, 3, …, 13, 14 y 15. Para acotar un poco más este conjunto, es necesario saber dónde están el segundo y el tercer ítemes más grandes, pero todas esas posiciones del arreglo son posibles para el cuarto ítem más grande.

ii) El complemento: el segundo no puede estar en la posición 4 (porque esta posición está a distancia 2 de la raíz) ni siguientes; el tercero no puede estar en la posición 8 (porque esta posición está a distancia 3 de la raíz) ni siguientes; y el cuarto no puede estar en la posición 15 ni siguientes.

2016-1-I2-P1–ABB modificado (1/1)

1. Un **árbol B⁺** (una variante de los *árboles B*) de orden m tiene las siguientes propiedades:

i) Los datos están almacenados únicamente en las hojas, ordenados por clave.

ii) Los nodos que no son hojas almacenan hasta m–1 claves (ordenadas, para guiar la búsqueda); la clave i-ésima representa la clave más pequeña en el subárbol (i+1)-ésimo.

iii) La raíz es ya sea una hoja o tiene entre dos y m hijos.

iv) Todos los nodos que no son hojas (excepto la raíz) tienen entre $\lceil m/2 \rceil$ y m hijos.

v) Todas las hojas están a la misma profundidad y tienen entre $\lceil t/2 \rceil$ y t datos, para algún t.

a) [**3**] Describe un algoritmo eficiente para insertar un nuevo dato en (una hoja de) un árbol B⁺.

Se puede emplear el mismo algoritmo "defensivo" del árbol B visto en clase: Si el nodo por el que vamos pasando, en nuestro camino de la raíz a la hoja donde se va a hacer la inserción, está lleno (es decir, tiene m–1 claves), entonces lo dividimos en dos antes de descender al hijo correspondiente. Al llegar a la hoja, si ésta tiene espacio (es decir, tiene menos que t datos), entonces simplemente insertamos el nuevo dato allí. En caso contrario, primero dividimos la hoja en dos, dejando en cada hoja nueva la mitad de los datos de la hoja original, y luego insertamos el dato en la hoja nueva correspondiente según la clave del dato. En todo este proceso, cada vez que dividimos un nodo, ya sea una hoja o no, hay que agregar un hijo más al padre del nodo; esto no es problema, porque cuando pasamos por el padre (cuando veníamos bajando desde la raíz) nos aseguramos de dejarlo con espacio: si estaba lleno, lo dividimos en dos.

También es válido el algoritmo más directo, que primero simplemente busca (a partir de la raíz) la hoja donde tiene que hacer la inserción. Una vez allí, si la hoja tiene menos que t datos, entonces simplemente inserta el dato y termina; pero si la hoja ya tiene t datos, entonces (al igual que en el algoritmo anterior) primero divide la hoja en dos, dejando en cada hoja nueva la mitad de los datos de la hoja original, y luego inserta el dato en la hoja nueva que corresponda. En este último caso, tiene que agregar un hijo adicional al padre de la hoja original. Si este nodo tiene menos que m–1 claves, entonces agregar el hijo adicional no es problema; pero si ya tiene m–1 claves, entonces hay que dividirlo en dos y agregarle recursivamente un hijo adicional a su padre. Este proceso puede llegar hasta la raíz.

En cualquiera de los dos algoritmos, si hay que dividir la raíz, entonces se la divide y los dos nodos resultantes pasan a ser hijos de una nueva raíz. Por esto está la propiedad iii) y la excepción mencionada en la propiedad iv).

Ambos algoritmos son O(log$n$), ya que hacen un número constante de pasos por cada nivel del árbol; la complejidad de dividir un nodo es independiente de n. La diferencia es que el primer algoritmo hace una sola "pasada", potencialmente más lenta, por los niveles del árbol (hacia abajo, de la raíz a una hoja), mientras que el segundo hace inicialmente una pasada rápida hacia abajo, pero podría tener que hacer una segunda pasada hacia arriba.

b) [**2**] Si tuvieras que imprimir todos los datos almacenados en la base de datos, ordenados de menor a mayor clave, ¿cuál de los dos tipos de árboles —árbol-B o árbol-B⁺— es más apropiado? Justifica.

El árbol B⁺, ya que como todos los datos están en las hojas, basta recorrer éstas secuencialmente en orden. Por supuesto, esto requiere conectar las hojas entre ellas, para que cuando se termina de recorrer una, se pueda pasar directamente a la que la sigue (en orden), sin tener que ir a buscarla a los nodos padres: es decir, cada hoja tiene un puntero a la hoja que está inmediatamente a su derecha (en la representación gráfica del árbol).

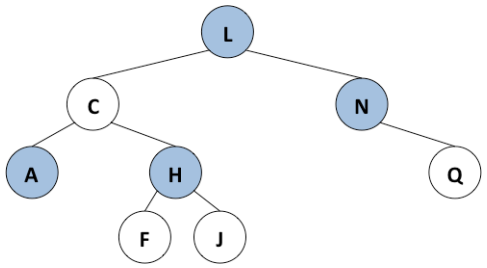
2016-1-I2-P2–Árbol rojo-negro (1/1)

2. Consideremos **árboles rojo-negros** definidos según las 4 propiedades enunciadas en clase (es decir, como una representación alternativa de árboles 2-3-4, y no simplemente de árboles 2-3).

a) [**1**] Describe un árbol rojo-negro con n claves que presenta la mayor razón posible de nodos rojos a nodos negros; justifica. ¿Cuál es esta razón?

Ya que los nodos rojos no pueden tener hijos rojos, el árbol pedido es uno en que cada nodo negro tiene dos hijos rojos, en cuyo caso la razón es 2 a 1.

b) [**3**] Considera el árbol rojo-negro que se muestra a la derecha, en que los nodos oscuros (*A*, *H*, *L*, *N*) son negros. Inserta la clave *D* y explica, muestra y ejecuta los pasos necesarios hasta volver a tener un árbol rojo-negro válido.



Al insertar *D*, queda como hijo izquierdo rojo de *F* (aquí tiene que haber un dibujo).

Como *F* también es rojo, tenemos un problema. Como *J*, el hermano de *F*, es rojo [caso descrito en la diap. 74], resolvemos el problema (en este nivel) intercambiando colores: *H* queda rojo, *F* y *J*, negros (dibujo).

Como *C*, el padre de *H*, es rojo, nuevamente tenemos un problema [caso mencionado en la diap. 76]. Como *A*, el hermano de *H*, es negro, no podemos simplemente intercambiar colores como antes; en cambio, hacemos una rotación simple a la izquierda en torno a *C*–*H*, sin cambiar colores [caso descrito en la diap. 70]: *H*, rojo, queda en lugar de *C*, también rojo, que queda como hijo izquierdo de *H*; *F*, negro, queda como hijo derecho de *C*, y *J*, negro, sigue como hijo derecho de *H* (dibujo).

Como *H* y *C*, padre e hijo, son rojos, y *N*, el hermano de *H*, es negro [caso descrito en la diap. 70], hacemos una rotación simple a la derecha en torno a *L*–*H*: *H* queda en la raíz y lo pintamos negro; *L* queda como hijo derecho de *H* y lo pintamos rojo; y *J* pasa a ser el hijo izquierdo de *L* (dibujo).

c) [**2**] En clase vimos —a propósito de los árboles AVL— que las rotaciones simples preservan la propiedad de árbol binario de búsqueda. ¿Qué pasa en el caso de los árboles rojo-negro con respecto a las propiedades adicionales, sobre el número de nodos negros y sobre nodos rojos consecutivos, en una ruta simple desde la raíz hasta una hoja? Demuestra que estas propiedades son preservadas por las rotaciones simples, o bien da ejemplos que muestren que no es así.

Las rotaciones simples no preservan ninguna de estas dos propiedades. Al poner al hijo en el lugar del padre, si el hijo es rojo (y su padre, negro), puede quedar como hijo de otro nodo rojo; p.ej., en **b**), al hacer una rotación a la derecha en torno a *H*–*F*, queda *F* como hijo derecho de *C*. En este mismo caso, el nodo negro, que estaba como raíz de un subárbol, aportando al número de nodos negros en todas las rutas hasta las hojas, ahora bajó un nivel y fue reemplazado por un nodo rojo, por lo que el número de nodos negros en algunas de esas rutas disminuyó en uno; p.ej., en **b**), al hacer una rotación a la derecha en torno a *L*–*C*, la ruta de *C* a *A* tiene un solo nodo negro, pero las otras (*C* a *F*, *C* a *J*, *C* a *Q*) tienen dos.

2015-2-I1-P1–Min-Heap (1/1)

Estructuras de Datos y Algoritmos – IIC2133

I1

31 agosto 2015

(***Algoritmo***: Secuencia ordenada y finita de pasos para llevar a cabo una tarea, en que cada paso es descrito con la precisión suficiente, p.ej., en un pseudo lenguaje de programación similar a C, para que un computador lo pueda ejecutar.)

1. Considera un conjunto de n elementos, de tipo numérico, distintos (es decir, dados dos elementos, a y b, cualesquiera que pertenecen al conjunto, se tiene que $a < b$ o $a > b$). Queremos encontrar la **mediana** del conjunto, es decir, el elemento m tal que la cantidad de elementos menores que m es igual a la cantidad de elementos mayores que m. Por supuesto, podríamos ordenar los n elementos y luego mirar el que queda en la posición $\lceil n/2 \rceil$ (suponemos que n es impar), pero posiblemente vamos a estar haciendo mucho más trabajo que el estrictamente necesario (ya que no nos interesa saber la posición relativa de ninguno de los otros n–1 elementos). Considera además que los elementos los vamos a ir leyendo uno a uno en un dispositivo móvil en que el uso eficiente de la memoria también es importante.

a) Describe una solución eficiente a este problema, a partir de las estructuras de datos estudiadas en clase: *heaps* binarios, tablas de *hash*, y/o árboles de búsqueda binarios.

La idea es mantener en todo momento un conjunto S con los $n/2$ elementos más grandes leídos hasta el momento. Una vez que los primeros $n/2$ elementos han sido leídos, cuando leemos un nuevo elemento, x, lo comparamos con el elemento más pequeño que hay en S, que denotamos por S_{min}. Si x es más grande, entonces reemplaza a S_{min} en S. S ahora tiene un nuevo elemento más pequeño, que puede ser o no x. Al finalizar el input, encontramos el elemento más pequeño en S y lo devolvemos como la respuesta.

Para que esta idea sea eficiente, usamos un min-heap para implementar S. Los primeros $n/2$ elementos los ponemos en el heap en tiempo $O(n/2)$ en total. El tiempo para procesar cada uno de los otros elementos tiene dos componentes: $O(1)$, para probar si el elemento va a parar a S, más $O(\log n/2)$, para eliminar S_{min} e insertar el nuevo elemento, si esto es necesario. Así, el tiempo total es $O(n/2 + n/2(\log n/2)) = O(n/2(\log n/2)) = O(n \log n)$.

b) Determina el número aproximado de operaciones o pasos individuales y la cantidad de celdas de memoria, ambos en función de n, que requiere tu solución en el peor caso.

2015-2-I1-P4–ABB (1/1)

4. Sobre árboles binarios de búsqueda (ABB's)

a) Describe un algoritmo de certificación para ABB's, suponiendo que sabemos que el árbol es binario. Es decir, tu algoritmo debe verificar que la propiedad de ABB se cumple. Tu algoritmo debe correr en tiempo O(número de nodos en el árbol).

P.ej., un algoritmo recursivo que, a partir de un nodo x, verifica que el hijo izquierdo de x sea abb, que el hijo derecho de x sea abb, y que la clave más grande del hijo izquierdo sea menor que la clave de x y que ésta sea menor que la clave más pequeña del hijo derecho.

b) Supongamos que la búsqueda de una clave k en un ABB termina en una hoja. Consideremos tres conjuntos: A, las claves a la izquierda de la ruta de búsqueda; B, las claves en la ruta de búsqueda; y C, las claves a la derecha de la ruta de búsqueda. Tomemos tres claves: $a \in A$, $b \in B$ y $c \in C$. ¿Es cierto o es falso que $a < b < c$? Justifica.

Falso. Para justificar, basta un ejemplo: Supongamos que la búsqueda pasa por un nodo con clave 7, baja al hijo derecho con clave 13 y baja al hijo derecho con clave 17, que es una hoja; así, las últimas tres claves en la ruta de búsqueda son 7, 13 y 17. Entonces, basta que el nodo con clave 13 tenga un hijo izquierdo, cuya clave debe ser mayor que 7 (y menor que 13), p.ej., 11; esta clave queda a la izquierda de la ruta de búsqueda, pero no es menor que cualquier clave en la ruta de búsqueda, en particular, no es menor que 7.

c) ¿Es la operación de eliminación "conmutativa" en el sentido de que elimnar x y luego y de un ABB deja el mismo árbol que eliminar y y luego x? Demuestra que efectivamente lo es o da un contra-ejemplo.

Contraejemplo: Supongamos que al eliminar un nodo con dos hijos, lo reemplazamos por su sucesor. Consideremos una raíz con clave 5, y dos hijos, con claves 3 y 11, respectivamente; el nodo con clave 11 a su vez tiene un hijo izquierdo con clave 7. Si eliminamos el nodo con clave 5 (dos hijos) y luego el nodo con clave 3 (hoja), dejamos un abb —raíz 7 e hijo derecho 11— distinto que si eliminamos el nodo con clave 3 (hoja) y luego el nodo con clave 5 (ahora sólo un hijo) —raíz 11 e hijo izquierdo 7.

2015-2-I2-P1–AVL, Árbol rojo-negro (1/1)

Estructuras de Datos y Algoritmos – IIC2133

I2 - pauta

14 octubre 2015

(***Algoritmo***: Secuencia ordenada y finita de pasos para llevar a cabo una tarea; cada paso es descrito con la precisión suficiente —p.ej., en un pseudo lenguaje de programación similar a C— para que un computador lo pueda ejecutar.)

1. **a**) ¿Es todo árbol AVL un árbol rojo-negro? Justifica.

Sí. Hay justificar que en un árbol AVL la ruta (simple) más larga de la raíz a una hoja no tiene más del doble de nodos que la ruta (simple) más corta de la raíz a una hoja. Esto es así:

- *El árbol AVL más "desbalanceado" es uno en el que todo subárbol derecho es más alto (en uno) que el correspondiente subárbol izquierdo (o vice versa); en tal caso, el número de nodos en la ruta más larga crece según 2h, y el de la ruta más corta, según h+1, en que h es la altura del árbol.*
- *Así, para cualquier ruta (simple) desde la raíz a una hoja, sea* d *la diferencia entre el número de nodos en esa ruta y el número de nodos en la ruta más corta. Si todos los nodos de la ruta más corta son negros, entonces los otros* d *nodos deben ser rojos: pintamos la hoja roja y, de ahí hacia arriba, nodo por medio hasta completar* d *nodos rojos.*

b) ¿Cuántos cambios de color y cuántas rotaciones pueden ocurrir a lo más en una inserción en un árbol rojo-negro? Justifica.

Recuerda que al insertar un nodo, x, lo insertamos como una hoja y lo pintamos rojo. Si el padre, p, de x es negro, terminamos. Si p es rojo, hay dos casos: el hermano, s, de p es negro; s es rojo. Si s es negro, hacemos algunas rotaciones y algunos cambios de color. Si s es rojo, sabemos que el padre, g, de p y s es negro; entonces, cambiamos los colores de g, p y s, y revisamos el color del padre de g.

Hacemos O(log n) cambios de color y O(1) rotaciones. Como dice arriba, esto ocurre solo cuando p *es rojo.*

Si s *es negro, hacemos exactamente una o dos rotaciones, dependiendo de si* x *es el hijo izquierdo o el hijo derecho de su padre; y hacemos exactamente dos cambios de color.*

Si s *es rojo, no hacemos rotaciones, solo cambios de color. Inicialmente, cambiamos los colores de tres nodos:* g*, que queda rojo, y sus hijos* p *y* s*, que quedan negros. Como* g *queda rojo, hay que revisar el color de su padre: si es negro, terminamos; si es rojo, repetimos estos últimos cambios de color, pero más "arriba" en el árbol.*

2015-2-Ex-P6–Árbol rojo-negro (sin respuestas) (1/1)

6. Con respecto a los árboles rojo-negros:

a) [**1 pt.**] Considere un árbol formado únicamente mediante la inserción de n nodos usando el método de inserción visto en clases. Justifica que si $n > 1$, el árbol tiene a lo menos un nodo rojo.

b) [**1 pt.**] Muestra con un ejemplo que un árbol de más de un nodo puede tener sólo nodos negros si su construcción ha incluido eliminaciones.

c) [**2 pts.**] Supón que insertamos un nodo x en un árbol rojo-negro y luego lo eliminamos inmediatamente. ¿Es el árbol resultante el mismo que el inicial? Justifica tu respuesta.

d) [**2 pts.**] Un árbol con 8 nodos tiene la siguiente configuración: la raíz, con clave Q, es negra; sus hijos tienen las claves D y T, y son rojo y negro, respectivamente; D tiene dos hijos negros: A y L; T tiene un único hijo, W; y L tiene dos hijos: J y N. Inserta en este árbol la clave G; en particular

i) [**0.5 pts.**] Dibuja el árbol original, antes de la inserción de G; deduce los colores de los nodos J, N y W.

ii) [**0.5 pts.**] Dibuja el árbol justo después de la inserción de G, pero antes de cualquier operación de restauración de las propiedades del árbol. ¿Cuál propiedad **no** se cumple?

iii) [**1 pt.**] Dibuja el árbol después de cada operación de cambio de colores y de cada operación de rotación, explicando qué problema se produce en cada caso, hasta llegar al árbol final.

2015-1-I1-P2–Heap, ABB (1/2)

2. *Colas priorizadas*.

a) Compara las implementaciones de una cola priorizada usando un **heap binario** y usando un **árbol de búsqueda binario**. Considera el caso en que la cola está inicialmente vacía y a continuación ocurren n operaciones de inserción de claves y n operaciones de extracción del elemento con la mayor clave, intercaladas arbitrariamente (por supuesto, la primera operación es una inserción y la última una extracción). ¿Cuántos pasos, en notación O( ), se ejecutan en cada una de las dos implementaciones? Considera tanto el mejor caso como el peor caso para ambas implementaciones. **Justifica**.

La intercalación consistente en alternar una inserción y una extracción —es decir, una inserción, una extracción, una inserción, una extracción, una inserción, etc.— es el mejor caso para ambas estructuras: la cola nunca tiene más de un elemento y el total de $2n$ operaciones toma $O(n)$ pasos.

El pero caso se da cuando primero se ejecutan las n inserciones y después las n extracciones. El heap tiene la gracia de estar siempre balanceado, por lo que su altura está acotada por $O(\log n)$, en cambio el árbol puede convertirse en una lista ligada, por lo que su altura está acotada por $O(n)$. De modo que en el peor caso el heap ejecuta $O(n \log n)$ pasos, mientras que el árbol puede llegar a ejecutar $O(n^2)$ pasos.

2015-1-I1-P2–Heap, ABB (2/2)

b) Tenemos k fuentes de datos. Los datos de cada fuente vienen ordenados de mayor a menor. Queremos enviar la totalidad de los datos recibidos por un único canal de salida, también ordenados de mayor a menor. El dispositivo electrónico que hace esta mezcla tiene una capacidad limitada de memoria. Describe un **algoritmo** para este dispositivo, que le permita hacer su tarea empleando un arreglo a de k casilleros, en que cada uno tiene dos campos: $a[j].data$ puede almacenar un dato, y a[j].num puede almacenar un número entero entre 1 y k. Para recibir un dato desde la fuente i, se ejecuta $receive[i]( )$, y para enviar un dato x por el canal de salida, se ejecuta $send(x)$. Si la cantidad total de datos en las k fuentes es n, ¿cuál es la cantidad total de pasos que ejecuta tu algoritmo, en notación O( )?

[**67%**] La idea es armar un (max-)heap binario de tamaño k, inicialmente con el primer dato (el mayor) de cada fuente, de modo que en la raíz quede el mayor de todos los datos:

```
for (i = 0; i < k; i = i+1)
    x.data = receive[i]()
    x.num = i
    insertObject(x)
```

A continuación, hay que ir sacando el dato que está en la raíz, enviándolo por el canal de salida, y reemplazándolo en el heap por el próximo dato recibido de la misma fuente de donde provenía el que salió:

```
for (j = 0; j < n-k; j = j+1)
    x = xMax()
    i = x.num
    send(x.data)
    x.data = receive[i]()
    if (x.data ≠ null) insertObject(x)
```

[**33%**] $O(n \log k)$, ya que el `for` ejecuta k operaciones `insertObject(x)`, c/u de las cuales ejecuta un número de pasos proporcional a log k, lo que da un subtotal de $k \log k$); y el `while` ejecuta $n–k$ operaciones *insertObject(x)*, c/u de las cuales nuevamente ejecuta un número de pasos proporcional a log k, lo que da otro subtotal de $(n–k)\log k$.

2015-1-I1-P3–AVL, Árbol rojo-negro, Árbol 2-3 (1/2)

3. *Árboles de búsqueda.*

a) La especificación de un árbol de búsqueda (no necesariamente balanceado) indica que cada nuevo elemento que se inserta debe quedar como la raíz del árbol. ¿En qué situaciones puede ser esto útil? ¿Qué tan eficientemente, en notación O( ), puede implementarse una operación `Insertar-Nodo()` que logre este objetivo? **Justifica**.

[**50%**] La raíz del árbol es el elemento que se encuentra con una comparación. Por lo tanto, me conviene que un nuevo elemento que acabo de insertar quede como raíz si a continuación lo voy a buscar varias veces; y si en el mediano plazo la probabiidad de buscar ese elemento se mantiene alta, ya que cuando insertemos otro elemento, que va a quedar como nueva raíz, la raíz anterior va a quedar como hija de la nueva raíz (o sea, se va a mantener en la parte "de arriba" del árbol por un tiempo).

[**50%**] Para lograr esto, `Insertar-Nodo()` debe hacer lo siguiente. Primero, una inserción "normal", en que el nodo insertado queda como hoja del árbol. Luego, "hacer subir" este nodo mediante rotaciones simples hasta dejarlo en la raíz del árbol: si el nodo es un hijo izquierdo, entonces se hace una rotación a la derecha, y viceversa. Recordemos que las rotaciones preservan la propiedad de árbol de búsqueda. Con cada rotación, el nodo "sube" un nivel en el árbol; por lo tanto, se necesita un número de rotaciones igual a la altura del árbol para que el nodo finalmente quede como raíz, es decir, O(*altura de árbol*).

b) Considera un **árbol AVL** inicialmente vacío, en el que queremos almacenar las claves 0, 1, 2, 3, 4, 5, 6, 7, 8 y 9. ¿Es posible insertar estas claves en el árbol en algún orden tal que **nunca** sea necesario ejecutar una rotación? Si tu respuesta es "sí", indica el orden de inserción y muestra al árbol resultante después de insertar cada clave. Si tu respuesta es "no", da un argumento convincente (p.ej., una demostración) de que efectivamente no es posible insertar las claves sin que haya que ejecutar al menos una rotación.

Sí es posible. La idea es hacer las inserciones de manera de mantener todo el tiempo la propiedad de balance; p.ej., procurar que el árbol se vaya llenando "por niveles". La primera clave que insertemos va a ser la raíz del árbol (ya que la idea es que no va a haber rotaciones). Por lo tanto, tiene que ser una clave *k* tal que el número de claves menores que *k* —que van a ir a parar al subárbol izquierdo— y el número de claves mayores que *k* —que van a ir a parar al subárbol derecho— sean muy parecidos. Si elegimos la clave 4, entonces hay 4 claves menores y 5 claves mayores (también podemos elegir la clave 5 y dejar 5 claves menores y 4 mayores). A continuación elegimos la raíz del subárbol izquierdo y la raíz del subárbol derecho (o viceversa). Para esto, aplicamos recursivamente la misma "regla", sobre las claves 0, 1, 2 y 3, para el subárbol izquierdo, y sobre las claves 5, 6, 7, 8 y 9, para el subárbol derecho; p.ej., insertamos 2 y luego 7. Repitiendo la estrategia, luego insertamos 1, 3, 6 y 8, y finalmente 0, 5 y 9. Así, un orden de inserción posible es 4, 2, 7, 1, 3, 6, 8, 0, 5, 9.

2015-1-I1-P3–AVL, Árbol rojo-negro, Árbol 2-3 (2/2)

c) Considera un **árbol rojo-negro** que corresponda a un **árbol 2-3**, tal como lo vimos en clase. Escribe el **algoritmo** que debe seguir el procedimiento de inserción en el árbol rojo-negro de manera que corresponda al procedimiento que seguiría en el árbol 2-3.

Correspondencia básica. En un árbol rojo-negro, un nodo rojo contiene la clave izquierda (menor) y los hijos izquierdo y del medio de un nodo 3 del árbol 2-3. La clave derecha y el hijo derecho del nodo 3 se convierten en la clave e hijo derecho de un nodo negro en el árbol rojo-negro; el hijo izquierdo de este nodo negro es el nodo rojo anterior. Todos los otros nodos 2 del árbol 2-3 corresponden a nodos negros en el árbol rojo-negro.

Dividimos la respuesta en tres partes.

a) Al insertar en el árbol 2-3, el caso más simple es cuando insertamos en un nodo 2 (que es una hoja), que así se convierte en un nodo 3 (y sigue siendo una hoja).

En el correspondiente árbol rojo-negro, insertamos un hijo a un nodo negro. La clave del hijo puede ser menor o mayor que la clave de su padre, es decir, puede quedar como hijo izquierdo o hijo derecho, respectivamente; además, el hijo debe ser pintado de rojo. Sin embargo, estos dos nodos deben quedar finalmente en una configuración que corresponda al nodo 3 (según la descripción de arriba). Esto será así si el hijo insertado es un hijo izquierdo. Pero si es un hijo derecho, es necesario hacer una rotación a la izquierda.

b) Un caso menos simple es cuando en el árbol 2-3 insertamos en un nodo 3, cuyo padre en un nodo 2; supongamos que el otro hijo de este nodo 2 también es un nodo 2. Es decir, en el árbol rojo-negro, insertamos en una configuración de un nodo negro con un hijo izquierdo rojo y sin hijo derecho (corresponde al nodo 3); el padre del nodo negro también es negro (corresponde al nodo 2) y tiene otro hijo negro (el otro nodo 2). Dependiendo del valor de la clave insertada, el nuevo nodo puede ir a parar como hijo izquierdo o derecho del nodo rojo, o como el hijo derecho, hasta ahora inexistente, del nodo negro: configuración inicial, justo después de la inserción.

En el árbol 2-3 el resultado es el siguiente: el nodo 2 se convierte en un nodo 3 (con dos claves y tres hijos) y el nodo 3 se convierte en tres nodos 2 (con una clave cada uno y sin hijos).

Por lo tanto, en el árbol rojo-negro la configuración final debe ser un nodo negro con dos hijos: el izquierdo rojo y el derecho negro; el hijo izquierdo rojo a su vez tiene dos hijos negros. A través de una o dos rotaciones, a uno u otro lado, y algunos cambios de colores, convertimos la configuración inicial en esta configuración final.

c) Finalmente, el caso más complicado es cuando en el árbol 2-3 insertamos en un nodo 3, cuyo padre también es un nodo 3. En el árbol rojo-negro correspondiente, la inserción es en una configuración de un nodo negro con dos hijos, izquierdo rojo y derecho negro, en que el hijo izquierdo tiene a su vez dos hijos negros, uno de los cuales, el izquierdo, tiene finalmente un hijo rojo. El nuevo nodo va a ir a parar inicialmente como hijo izquierdo o derecho de este nodo rojo, o como hijo derecho de su padre negro, y en cualquier caso pintado de rojo. En la configuración final, las hojas y sus padres son negros; pero la parte complicada se da porque alguna clave sube de nivel, potencialmente repitiendo el problema original dos niveles más arriba. Por supuesto, este problema se puede resolver aplicando recursivamente el algoritmo descrito (pero esto es más fácil decirlo que especificarlo precisamente).

2015-1-Ex-P1–AVL (1/1)

Estructuras de Datos y Algoritmos – IIC2133

Examen

30 junio 2015

1. Considera un árbol AVL en el que acabamos de insertar una nueva clave; sea *Q* (un puntero a) el nodo en que quedó esta nueva clave, y sea *P* (un puntero a) el padre de *Q*. Si cada nodo tiene un campo *balance* (o simplemente *bal*), además de punteros a su padre e hijos izquierdo y derecho (*p*, *izq*, *der*), …

… **escribe un algoritmo** —**lo más parecido a código posible**— **eficiente** que actualice el campo *balance* de cada nodo que lo necesite, y determine el nodo "pivote", o raíz del subárbol que debe ser rebalanceado mediante una rotación, si es que lo hay. ¿Qué complejidad tiene tu algoritmo? Justifica.

[**2/3**] Algoritmo:

```
Q = nodo recién insertado
P = Q.p
if Q es hijo izquierdo de P
    P.balance --
else
    P.balance ++
while P no es la raíz y P.balance ≠ 2 y P.balance ≠ –2
    Q = P
    P = P.p
    if Q.balance = 0
        return
    if Q es hijo izquierdo de P
        P.balance --
    else
        P.balance ++
if P.balance = 2 o P.balance = –2
    P es el nodo pivote
```

[**1/3**] Complejidad = O(log*n*), ya que hace un número constante de operaciones en el nodo que está mirando y luego "sube" al padre del nodo, en donde repite las operaciones; a lo más llega a la raíz, que está a altura log*n*, en que *n* es el número de nodos en el árbol, ya que los árboles AVL son balanceados. **¡ Ojo: Se asigna puntaje sólo si está bien justificado** !

2014-2-I1-P3–Heap modificado (1/1)

3. Un *heap flojo* es uno en donde $A[Parent(Parent(i))] \geq A[i]$ cuando $i \geq 4$, y tal que $A[Parent(i)] \geq A[i]$, cuando $1 \leq i < 4$. Así, se cumple que todo heap también es un heap flojo, pero no al revés.

   a) Dé un pseudocódigo para la inserción de un elemento en un heap flojo y argumente que es correcta y más eficiente en la práctica que la misma operación en heaps tradicionales.

   Respuesta: Una posible solución es la siguiente.

```
1:  function INSERT(A, n)
2:      A.heap-size ← A.heap-size + 1
3:      A[heap-size] ← n
4:      SIFTUP(heap-size)

5:  function SIFTUP(A, i)
6:      if i = 1 then return                          ▷ Llegamos a la raíz
7:      next ← Parent(Parent(i))
8:      if next < 1 then                              ▷ Este nodo no tiene abuelo
9:          next ← Parent(i)
10:     if A[next] > A[i] then
11:         swap(A[next], A[i])
12:         SIFTUP(next)
```

   El pseudocódigo anterior preservará la propiedad del *heap flojo* de forma similar que un heap tradicional. Es más eficiente ya que realizará $\lceil \lfloor \lg n \rfloor / 2 \rceil$ llamadas recursivas en lugar de $\lfloor \lg n \rfloor$.

   b) Explique en detalle (no es necesario un pseudocódigo) cómo se debe implementar la operación *SiftDown*. ¿Es esta operación más eficiente que su homónima en heaps? Justifique.

   Respuesta: La implementación de *SiftDown* seguirá la misma idea que la implementación en (a). Sin embargo, hay que considerar que cuando hacemos *SiftDown* en la raíz, tenemos que considerar tanto a los hijos como los nietos. Es decir, deberemos comparar la raíz con sus 2 hijos y 4 nietos y intercambiarla con el máximo de todos. Luego, continuamos recursivamente comparando el nodo que cambiamos con sus 4 nietos e intercambiándolos hasta que no sea necesario o alcancemos el final del heap.

   Dado lo anterior, no es claro si esta versión de *SiftDown* será más eficiente que la tradicional, ya que a pesar de que realizaremos aproximadamente la mitad de las llamadas recursivas, en cada una de ellas debemos realizar cuatro comparaciones en lugar de dos.

2014-2-I2-P2–AVL (1/1)

2. *a*) **(1/4)** ¿De qué tamaño es la rama más corta que puede tener un árbol AVL de altura h? Justifique.
Solución: Vista en clases, $\lfloor \frac{h}{2} \rfloor$

b) **(3/4)** Considere la afirmación:

> Dado un entero m correspondiente al tamaño de la tabla de hash, es posible demostrar que $h_{a,b}(k) = ((ak + b) \mod p) \mod m$, donde $a, b \in \{0, \ldots, p-1\}$ y p es un primo "grande" define una familia de funciones de hash universales.

¿Cómo debe estar definida la propiedad de ser "grande" arriba para que la afirmación sea verdadera? Muestre cómo es que cuando se viola esa propiedad la familia de funciones deja de ser universal.

Solución: Sea el universo $U = \{0, ..., u-1\}$, sea p un número primo tal que $p \leq u$, y definamos $h_{a,b}(x) = ((ax + b) \mod p) \mod m$ donde a, b son elegidos aleatoreamente módulo p tal que $a \neq 0$, primero debemos ver para que $H = \{h_{a,b}\}$ es una familia de funciones universales notemos que $h(x) = h(y)$ solamente cuando $ax + b \equiv_{mod(p)} ay + b + im$, para algun entero i entre 0 y $\frac{p}{m}$. Si $x \neq y$, su diferencia es distinta de 0 y por tanto existe una función inversa para obtener a tal que $a \equiv_{mod(p)} im(x-y)^{-1}$. Luego, hay $p-1$ posibles elecciones para a ya que $a \neq 0$ y, variando i en el rango permitido, existen $\lfloor \frac{p}{m} \rfloor$ valores posibles, luego la probabilidad de colisión es:

$$\frac{\lfloor \frac{p}{m} \rfloor}{p-1}$$

Luego tenemos que para m constante

$$\lim_{p \to \infty} \frac{\lfloor \frac{p}{m} \rfloor}{p-1} = \frac{1}{m}$$

, es decir limita la probabilidad de colisión al tamaño de la tabla.

2014-2-I2-P3–Árbol 2-3, Árbol 2-4 (1/2)

3. Los árboles $(2,3)$ son árboles binarios de búsqueda *multiway* que son como los árboles $(2,4)$, pero no permiten la existencia de 4-nodos.

   a) Demuestre que si K es un conjunto de objetos, entonces es posible construir un árbol $(2,3)$ conteniendo exactamente los objetos de K que esté *perfectamente balanceado*, es decir, en donde cada rama tiene el mismo largo.

   Solución: Procederemos por inducción.

   - Caso Base $K = 1$, si el árbol tiene un solo elemento, entonces tiene un sólo nodo, luego todas sus hojas tendran la misma altura por lo que la propiedad se cumple.
   - Hipoótesis de induccion, supongamos que para todo $0 \leq K \leq n$, se cumple que el árbol siempre se encuentra balanceado.
   - Tesis de inducción: Por demostrar la propiedad para $n+1$ elementos, consideremos un árbol $(2,3)$ de n elementos, el cual se encuentra balanceado por hipótesis de inducción. Ahora si insertamos un elemento obtenemos 2 casos:
     - Si al agregar un elemento no ocurre overflow entonces no hay problemas ya que no se crearon nodos nuevos y por ende el arbol sigue balanceado.
     - Si al agregar un elemento ocurre un overflow se toma el elemento central del nodo y se sube al padre, si en el padre no ocurre overflow, terminamos y el arbol sigue balanceado, si ocurre overflow se sube el nodo central al padre de este nodo, se juntan los dos hijos centrales del primer nodo y se separa los extremos de este nodo en dos, este procedimiento

2014-2-I2-P3–Árbol 2-3, Árbol 2-4 (2/2)

se repite hasta que no ocurra overflow, cuando terminamos nos damos cuenta que no creamos ninguna hoja extra por lo que el árbol resultante debe estar balanceado. Luego como la hipótesis implica la tesis la propiedad debe ser cierta.

b) Todo árbol $(2,3)$ es un árbol $(2,4)$. Este hecho por si solo, sin embargo, no prueba que posible usar el algoritmo de eliminación de los árboles $(2,4)$ directamente sobre los árboles $(2,3)$. Argumente convincentemente por qué sí es posible usarlo.

Solución: Supongamos que no es posible utilizar el algoritmo de eliminación de los árboles $(2,4)$ en los árboles $(2,3)$, sabemos que el algoritmo de los árboles $(2,4)$ es correcto, luego debe poder eliminar cualquier caso de un árbol $(2,4)$. Por otra parte, notemos que un árbol $(2,4)$ también contiene casos de eliminación de los arboles $(2,3)$, pero nuestro algoritmo no funcióna para la eliminación de los casos que corresponden a árboles $(2,3)$, pero esto contradice el hecho de que funciona para todos los casos de los árboles $(2,4)$ por tanto nuestra suposición de que el algoritmo no funciona para los árboles $(2,3)$ es incorrecta.

2014-2-I2-P4–AVL, Árbol rojo-negro (1/1)

4. *a*) Escriba un pseudocódigo detallado de la operación *restructure*, que es la base de las operaciones de balanceo de los árboles AVL y Rojo-Negro. Suponga que la operación toma como argumento un árbol T y tres nodos en T: u, v y w tales que v es hijo de u y w es hijo de v.

   Respuesta: lo importante de este algoritmo es que logre identificar los casos que pueden darse y actualize correctamente las referencias en el árbol.

   Existen muchos algoritmos correctos, a continuación se plantea uno.

```
1: function RESTRUCTURE(T, u, v, w)
2:     if left[u] = v then
3:         if right[v] = w then
4:             LEFT-ROTATE(v)
5:         RIGHT-ROTATE(u)
6:     else                                  ▷ casos simétricos
7:         if left[v] = w then
8:             RIGHT-ROTATE(v)
9:         LEFT-ROTATE(u)
```

   b) Dé un algoritmo que haga $O(altura[T])$ llamados a *reestructure* para implementar la operación $\mathsf{split}(T, k)$, que retorna un par $\langle T_{\leq k}, T_{\geq k} \rangle$, donde

   - T es un ABB y k es una clave.
   - $T_{\leq k}$ es un ABB que contiene elementos en T cuya clave es menor o igual a k,
   - $T_{\geq k}$ es un ABB que contiene elementos en T cuya clave es mayor o igual a k,
   - Todo elemento de T está en $T_{\leq k}$ o en $T_{\geq k}$.

   Respuesta: en esta pregunta no era necesario escribir un pseudocódigo detallado, bastaba con describir el algoritmo de forma precisa. Un algoritmo que funciona es el siguiente:

   1) Insertamos un nodo u con clave k en el árbol.
   2) Llamamos *restructure* sobre u, su padre y su abuelo.
   3) Repetimos (2) hasta que u sea la raíz o hijo de la raíz.
   4) Si u es hijo de la raíz, aplicamos la rotación adecuada para dejar a u como raíz.
   5) Los árboles buscados corresponderán a $left[u]$ y $right[u]$.

   Este algoritmo hará $O(altura[T])$ llamados a *restructure* por las siguientes razones. El paso (1) hará $O(altura[T])$ llamados a *restructure*. Además, cada vez que hacemos (2), u sube por lo menos un nivel y, por lo tanto, (2) se ejecutará menos de $altura[T]$ veces. Luego, el algoritmo cumple la cota mencionada.

2014-1-I1-P1–MinHeap (1/1)

Estructuras de Datos y Algoritmos – IIC2133

I1

14 abril 2014

(***Algoritmo***: Secuencia ordenada y finita de pasos para llevar a cabo una tarea, en que cada paso es descrito con la precisión suficiente para que un computador lo pueda ejecutar.)

1. *Colas de prioridades.*

a) ¿Cuál es el número máximo de comparaciones que es necesario realizar para encontrar la clave más grande almacenada en un *min-heap* binario? Justifica

Como se trata de un *min-heap*, la clave más grande tiene que estar en un nodo sin hijos (en un nodo con hijos, la clave del nodo debe ser menor que las claves de sus hijos), es decir, en una hoja; como puede ser cualquier hoja (no hay ninguna propiedad de orden entre las claves de los nodos en un mismo nivel), el número de comparaciones es $n/2$ (estrictamente, $\lceil n/2 \rceil$), si el *min-heap* tiene n nodos.

b) Describe un algoritmo para encontrar todas las claves menores que algún valor, X, almacenadas en un *min-heap* binario? Tu algoritmo debe correr en tiempo $O(K)$, en que K es el número de claves que cumplen la condición.

Como se trata de un *min-heap*, primero hay que mirar la raíz; si la clave de la raíz es menor que X (encontramos una), entonces hay que mirar, recursivamente, cada uno de sus hijos. Cada "mirada" de un nodo es simplemente una comparación.

c) Si el 99% de las operaciones sobre una cola de prioridades son *insert*( ) y sólo el 1% son *xMin*( ), ¿cuál implementación de la cola sería más apropiada: *min-heap* binario, arreglo no ordenado, o arreglo ordenado? Justifica.

El análisis debe tener el siguiente "sabor".
En un *min-heap*, *insert*( ) toma $\log n$ pasos y *xMin*( ) también toma $\log n$ pasos; en promedio, el número de pasos cada 100 operaciones es entonces $100 \log n$.
En un arreglo no ordenado, *insert*( ) toma un paso, pero *xMin*( ) toma n pasos; en promedio, el número de pasos cada 100 operaciones es entonces $99 + n$.
En un arreglo ordenado, *insert*( ) toma en promedio $n/2$ pasos, y *xMin*( ) toma un paso; en promedio, el número de pasos cada 100 operaciones es entonces $44.5n + 1$.
Claramente, el candidato está entre $100 \log n$ y $99 + n$. Para $n \geq 1024$ ($= 2^{10}$), preferimos el *min-heap*; pero para $n \leq 512$ ($= 2^9$), preferimos el arreglo no ordenado. El punto de quiebre está entre $n = 878$ y $n = 879$.

2014-1-I1-P3–ABB (1/1)

3. *Árboles de búsqueda binarios (ABB).*

a) Supongamos que tenemos una estimación por adelantado de cuán a menudo nuestro programa va a tener acceso a las claves de los datos (es decir, conocemos la frecuencia de acceso a las claves). Si queremos emplear un ABB, inicialmente vacío, para almacenar los datos (según sus claves), ¿en qué orden deberíamos insertar las claves en el árbol? Justifica.

La idea es encontrar lo más rápidamente posible cada clave, cuando la busquemos. Como el número de comparaciones para encontrar una clave depende directamente de su profundidad en el árbol, las claves que están cerca de la raíz se encuentran haciendo pocas comparaciones; y, si el árbol está más o menos balanceado, las claves que están cerca de las hojas se encuentran haciendo varias comparaciones. Luego, nos conviene que las claves que se van a buscar más frecuentemente estén más cerca de la raíz, y que las claves que se van a buscar con menos frecuencia estén más cerca de las hojas. En un árbol que se construye sólo con inserciones (y sin rotaciones), el primer nodo que se inserta se convierte en la raíz del árbol; por lo tanto, en nuestro caso, éste debería ser el de más alta frecuencia esperada de búsqueda.

b) Supongamos que cada vez que nuestro programa tiene acceso a una clave, k, necesita saber cuántas claves en el árbol son menores que k; esto se conoce como el **rango** de k. ¿Qué información adicional sería necesario almacenar en el árbol? ¿Cómo se determinaría el rango de k y cuál sería la complejidad de esta operación? ¿Cuánto costaría mantener actualizada la información adicional del árbol cuando se produce una inserción o una eliminación de una clave?

Como los ABB son ordenados, el rango de k es el número de claves almacenadas "a la izquierda" de la clave k. Así, si k está justamente en la raíz del árbol, entonces su rango es igual al número de nodos en el subárbol izquierdo. En cambio, si k es menor que la clave de la raíz (y, por lo tanto, está en el subárbol izquierdo), entonces su rango (en todo el árbol) es igual a su rango dentro del subárbol izquierdo, ya que las claves en el subárbol derecho son todas mayores que k. Finalmente, si k es mayor que la clave de la raíz (y, por lo tanto, está en el subárbol derecho), entonces su rango es igual al número de claves en el subárbol izquierdo (ya que todas ellas son menores que k) más uno (la clave de la raíz también es menor que k) y más el rango de k dentro del subárbol derecho (el número de claves del subárbol derecho que son menores que k). Por supuesto, el rango de k en el subárbol izquierdo y el rango de k en el subárbol derecho se determina recursivamente de la misma forma.

Por lo tanto, la información adicional que conviene almacenar en el árbol para poder aplicar el algoritmo anterior es, para cada nodo t, almacenar en un campo *size* el número de nodos que hay en el subárbol cuya raíz es t. Esta información se mantiene actualizada fácilmente. Durante una inserción, sumamos uno al campo *size* de cada nodo en la ruta de inserción, y para el nodo insertado hacemos *size* = 1. Luego de hacer una eliminación, restamos uno al campo *size* de cada nodo en la ruta desde el padre del nodo eliminado hasta la raíz. En ambos casos, esto cuesta O(altura del árbol).

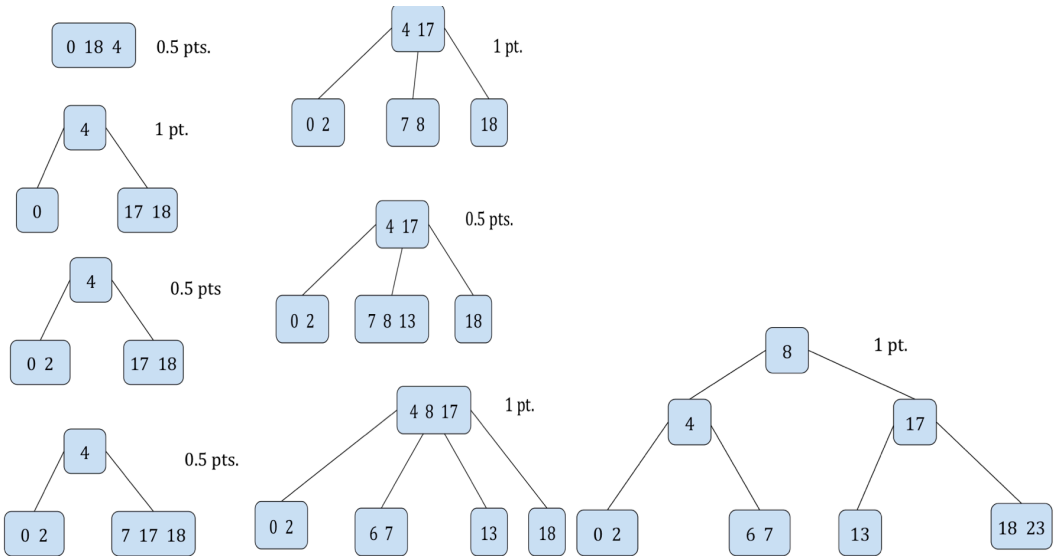
2014-1-I1-P4–AVL (1/1)

4. *Árboles de búsqueda balanceados.*

a) Describe un algoritmo de certificación para árboles AVL, suponiendo que sabemos que el árbol es un ABB. Es decir, tu algoritmo debe verificar que la propiedad de balance del árbol se cumple y que la información de balance mantenida en cada nodo está correcta. Tu algoritmo debe correr en tiempo O(número de nodos en el árbol).

El algoritmo asigna una "altura" a cada nodo: una hoja tiene altura 0; la altura de un nodo que no es una hoja es uno más que la altura de su subárbol más alto. El algoritmo comienza en la raíz del árbol.
Si el nodo es una hoja, entonces su balance debe ser 0. Si el balance no es 0, entonces el algoritmo responde "falso" y termina; de lo contrario, el algoritmo devuelve la altura del nodo: 0.
Si el nodo no es una hoja, entonces el algoritmo se ejecuta recursivamente primero en el subárbol izquierdo y luego el subárbol derecho del nodo. Si estas ejecuciones recursivas vuelven con los valores h_{izq} y h_{der}, respectivamente, entonces el algoritmo verifica la propiedad de árbol AVL (h_{izq} y h_{der} no pueden diferir en más de uno) y el balance del nodo (debe ser 0, –1 o +1, dependiendo de la relación entre h_{izq} y h_{der}). Si cualquiera de estas verificaciones no se cumple, entonces el algoritmo responde "falso" y termina; de lo contrario, el algoritmo devuelve la altura del nodo: uno más que el mayor entre h_{izq} y h_{der} y, si el nodo es la raíz del árbol, el algoritmo responde "verdadero".

b) Dibuja los árboles-B resultantes al ir insertando las claves **4**, **17**, **2**, **7**, **8**, **13**, **6** y **23** en un árbol-B de grado mínimo $t = 2$ que inicialmente contiene un solo nodo (la raíz) con las claves **0** y **18**.



2014-1-Ex-P2–Árbol rojo-negro (1/1)

2) Un árbol rojo-negro con 8 nodos tiene la siguiente configuración: la raíz, con clave Q, es negra; sus hijos tienen las claves D y T, y son rojo y negro, respectivamente; D tiene dos hijos negros: A y L; T tiene un único hijo, W; y L tiene dos hijos: J y N. Inserta en este árbol la clave G; en particular,

a) [**1 pt.**] Dibuja el árbol original, antes de la inserción de G; deduce los colores de los nodos J, N y W.

J, N y W son rojos. Si alguno fuera negro, entonces se violaría la propiedad de la altura negra del árbol: la ruta Q–D–A, en que A es una hoja, tiene sólo dos nodos negros, y cualquiera de las rutas desde la raíz a J, N o W ya tiene dos nodos negros.

b) [**2 pts.**] Dibuja el árbol justo después de la inserción de G, pero antes de cualquier operación de restauración de las propiedades del árbol. ¿Cuál propiedad no se cumple?

G se inserta en la forma habitual en un árbol de búsqueda, queda como hijo izquierdo de J, y se pinta rojo; pero J es rojo, por lo que ahora hay un nodo y su hijo ambos rojos.

c) [**3 pts.**] Dibuja el árbol después de cada operación de cambio de colores y de cada operación de rotación, explicando qué problema se produce en cada caso, hasta llegar al árbol final.

Primero, como el hermano N de J también es rojo, cambiamos colores: J y N quedan negros, y su padre, L, rojo. Pero el padre, D, de L también es rojo; es decir, "subimos" el problema de dos nodos rojos consecutivos.

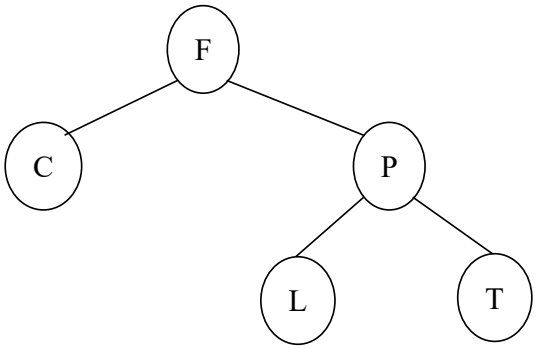
Pero ahora el hermano, T, de D es negro, por lo que no podemos aplicar el cambio de colores anterior. Como L es hijo derecho de D, rotamos D–L a la izquierda, sin cambiar colores, dejando L como hijo izquiedo de la raíz, y D como hijo izquierdo de L.

De nuevo hay un nodo y su hijo rojos, L y D, y el hermano, T, de L es negro. Pero ahora D es hijo izquierdo de L; por lo que rotamos Q–L a la derecha y pintamos Q de rojo.

2013-2-I1-P4–AVL (1/5)

4. *Árboles AVL.*

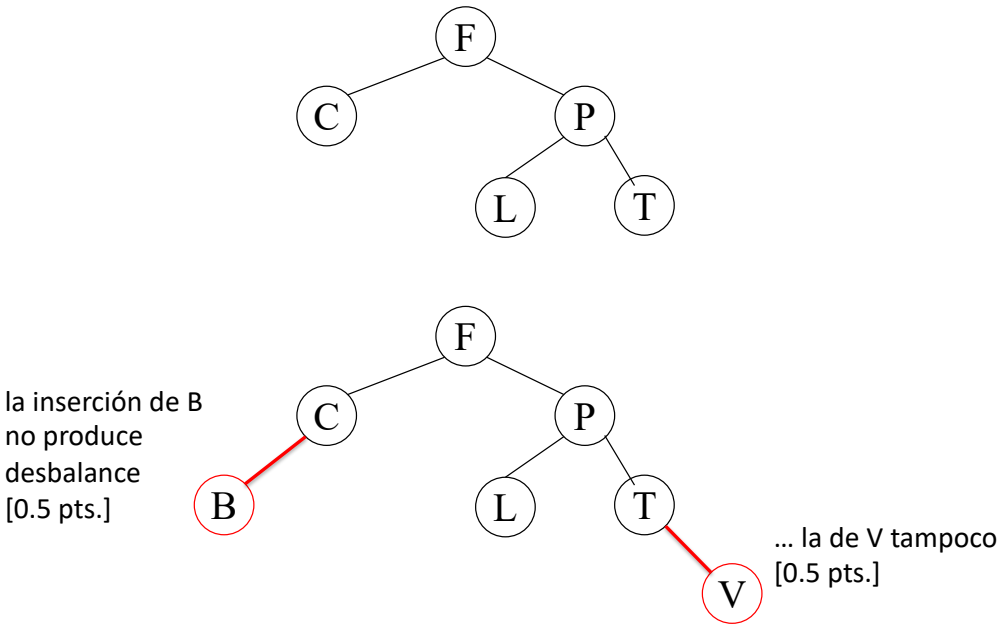
a) [**4 pts.**] A partir del árbol AVL de la figura, muestra cómo va quedando el árbol después de insertar las claves B, V, X y R, una a continuación de la otra. Específicamente, después de cada inserción explica si se produjo o no un desbalance, y, en caso afirmativo, identifica el pivote, explica cómo se resuelve el desbalance, y muestra el árbol rebalanceado.



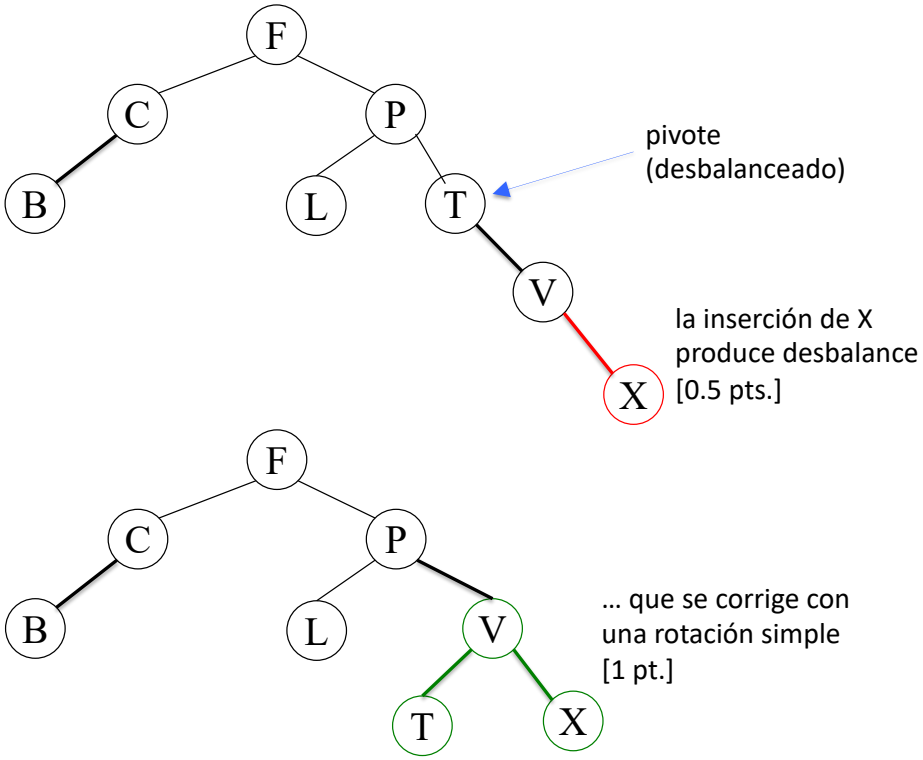
b) [**2 pts.**] Dibuja esquemáticamente (sin las claves) el árbol AVL más pequeño (con el menor número de nodos) de altura 6 (es decir, en que la rama más larga desde la raíz hasta una hoja tiene 7 nodos).

2013-2-I1-P4–AVL (2/5)

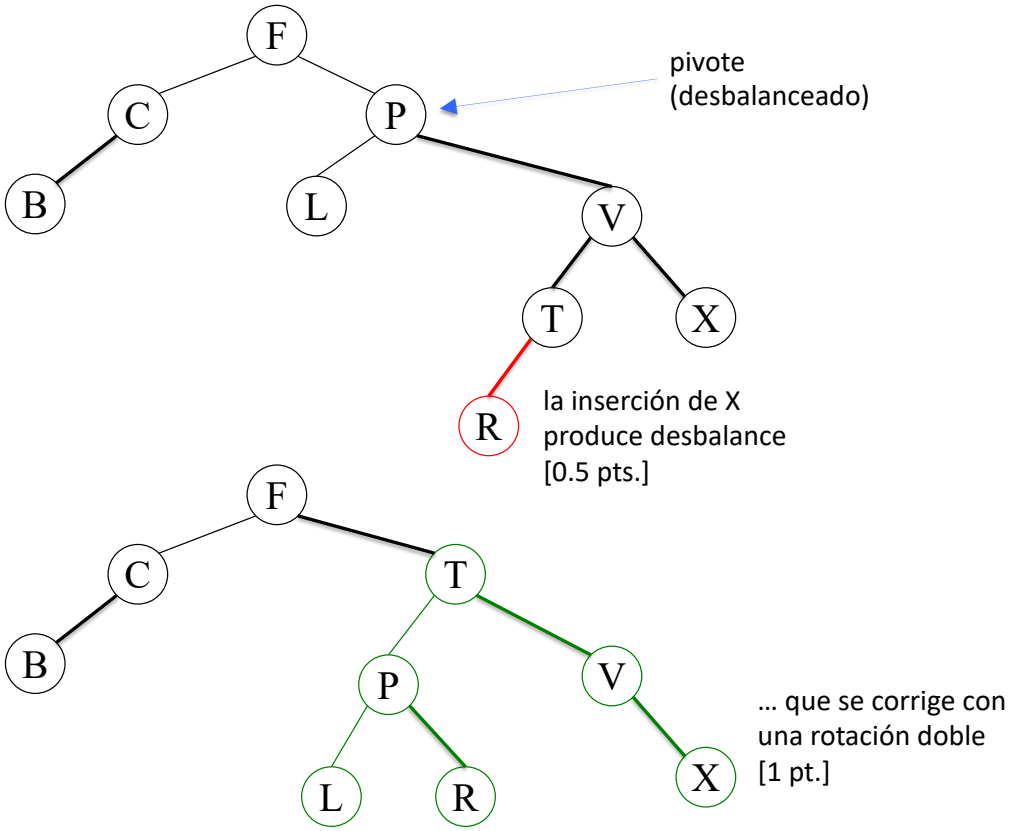
4a)



2013-2-I1-P4–AVL (3/5)

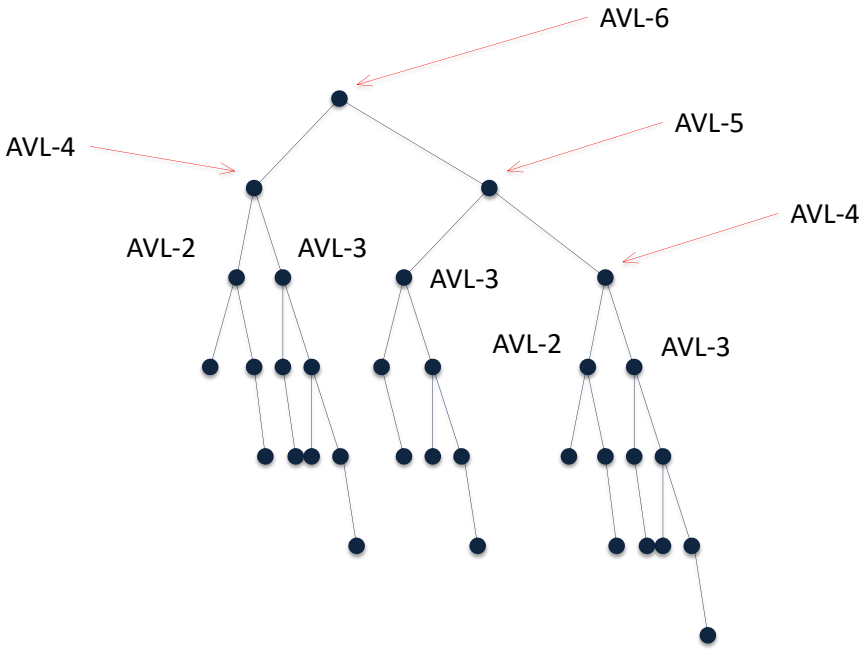


2013-2-I1-P4–AVL (4/5)



2013-2-I1-P4–AVL (5/5)

4b) El AVL más pequeño de altura 6 (AVL-6) está compuesto de un AVL más pequeño de altura 5 (AVL-5) y un AVL más pequeño de altura 4 (AVL_4); recursivamente, el AVL-5 está compuesto por un AVL-4 y un AVL-3; y el AVL-4, por un AVL-3 y un AVL-2.



2013-2-I2-P1–Propuesta de Árbol de Búsqueda (1/1)

Estructuras de Datos y Algoritmos – IIC2133

I2

4 octubre 2013

1. *Árboles B⁺ —una variación de los árboles B.* Un **árbol B⁺** de orden m tiene las siguientes propiedades:

i) Los datos están almacenados en las hojas.

ii) Los nodos que no son hojas almacenan hasta m–1 claves (para guiar la búsqueda); la clave i-ésima representa la clave más pequeña en el subárbol (i+1)-ésimo.

iii) La raíz es ya sea una hoja o tiene entre dos y m hijos.

iv) Todos los nodos que no son hojas (excepto la raíz) tienen entre $\lceil m/2 \rceil$ y m hijos.

v) Todas las hojas están a la misma profundidad y tienen entre $\lceil t/2 \rceil$ y t datos, para algún t.

a) Describe un algoritmo para insertar un nuevo dato en (una hoja de) un árbol B⁺.

b) Describe un algoritmo para eliminar un dato de (una hoja de) un árbol B⁺.

Tus descripciones pueden ser en prosa, pero deben ser claras y precisas; numera pasos, identifica casos.

Respuestas

Básicamente, se pueden emplear los mismos algoritmos "defensivos" del árbol B vistos en clase, aunque simplificando la eliminación, porque ahora sólo eliminamos datos que están en las hojas.

También son aceptables los algoritmos más simples, que primero insertan o eliminan un dato en una hoja, y luego restauran las propiedades del árbol. En la inserción, si es que la hoja queda con más de t datos, hay que dividirla en dos, por lo que hay que agregar una nueva clave al nodo padre; si este nodo ya tiene m–1 claves, entonces también hay que dividirlo en dos, pasando una nueva clave a su padre; y así sucesivamente hasta, posiblemente, llegar a la raíz; si es necesario dividir la raíz, entonces hay que crear una nueva raíz, cuyos hijos serán los dos nodos resultantes de la división de la raíz original.

En la eliminación, si la hoja queda con menos de t/2 datos, entonces se le puede traspasar un dato de un hermano "inmediato"; excepto si ambos hermanos tienen sólo t/2 datos cada uno, en cuyo caso hay que fusionar la hoja con alguno de sus hermanos, haciendo que su padre quede con una clave menos. Si este nodo tiene sólo m/2 hijos, entonces se aplica la misma estrategia; y así sucesivamente hasta, posiblemente, llegar a la raíz; si la raíz llega a quedar con un solo hijo, entonces simplemente la eliminamos y convertimos ese hijo en la nueva raíz.

2013-2-I2-P2–Árbol rojo-negro (1/1)

2. *Otro algoritmo de inserción en árboles rojo-negros.* En lugar de primero insertar el nuevo nodo (pintándolo de rojo) y después hacer rotaciones y/o cambios de color (*hacia arriba*) para restaurar las propiedades del árbol, podemos ir haciendo los ajustes a medida que vamos *bajando* por el árbol hacia el punto de inserción, de modo que cuando insertamos el nuevo nodo simplemente lo pintamos de rojo y sabemos que su padre es negro. El procedimiento es el siguiente.

Mientras bajamos por el árbol, cuando vemos un nodo X que tiene dos hijos rojos, intercambiamos colores: pintamos X de rojo y sus hijos de negro. Esto producirá un problema sólo si el padre P de X es rojo. Pero en este caso, simplemente aplicamos las rotaciones apropiadas, que se ilustran en las diapositivas 35 y 36 adjuntas.

a) Muestra cómo opera este nuevo algoritmo de inserción cuando insertamos un nodo con la clave K en el árbol de la diapositiva 37 adjunta; específicamente, muestra lo que ocurre a medida que llegas a cada nivel.

Respuesta

Cuando se baja de la raíz "H" a su hijo derecho "T" no pasa nada, ya que "T" (el X del algoritmo) tiene sólo un hijo rojo, "P". Cuando se baja de "T" a "P" tampoco pasa nada, ya que "P" (el X del algoritmo) es rojo. Cuando se baja de "P" a "L" estamos en el caso descrito en el algoritmo: "L" (el X del algoritmo) es negro y sus dos hijos, "J" y "N", son rojos. [*Hasta aquí no hemos cambiado nada, sólo bajamos por el árbol*: **0.5 pts.**]

Entonces, cambiamos colores: pintamos a "L" de rojo y a sus hijos "J" y "N" de negro. Pero ahora "L" y su padre "P" son rojos; como el hermano "W" de "P" es negro, estamos en la situación descrita en la diap. 35: "L" es X, "P" es P, "T" es G, "W" es S. [**1 pt.**]

Aplicando la rotación y cambios de colores sugeridos en la diap. 35, queda "P" de negro como nuevo hijo derecho de "H" y con dos hijos rojos, "L" y "T", que en total tienen cuatro hijos negros, "J", "N", "R" y "W" (este último mantiene sus dos hijos rojos, "V" y "Z", aunque ahora los tres están un nivel más abajo). [**1 pt.**]

Nosotros seguimos en "L", que ahora es rojo y está un nivel más arriba, así como sus hijos, "J" y "N", ahora negros. Bajamos a "J", que es una hoja negra, y por lo tanto insertamos "K" como hijo derecho de "J" y lo pintamos rojo. [**0.5 pts.**]

b) Los casos ilustrados en las diapositivas 35 y 36 se producen cuando el hermano S del padre P de X es negro. Explica claramente qué pasa con el caso en que S es rojo.

Respuesta

Este caso en realidad ***no se produce***, debido a los ajustes que vamos haciendo a medida que bajamos por el árbol. Si es que encontramos un nodo Y con dos hijos rojos, sabemos que sus nietos tienen que ser todos negros; y como también pintamos de negro los hijos de Y, entonces incluso después de las posibles rotaciones no vamos a encontrar otro nodo rojo en los próximos dos niveles.

2013-2-Ex-P2–ABB (1/1)

2) En clase vimos cómo se elimina una clave de un árbol binario de búsqueda (ABB, no necesariamente balanceado).

a) [**1 pt.**] Eliminar la clave cuando el nodo que ocupa no tiene hijos o tiene sólo un hijo, *Ti* o *Td*, es fácil: describe las acciones correspondientes.

Si el nodo no tiene hijos, entonces simplemente se elimina; si el nodo tiene un hijo, entonces ese hijo se coloca en el lugar del nodo.

b) [**1 pt.**] Es más difícil eliminar una clave cuando el nodo que ocupa tiene ambos hijos, *Ti* y *Td*: describe las acciones correspondientes. Esta forma de eliminación se llama *eliminación por copia*.

Se busca la clave sucesora (o predecesora) de la clave eliminada, y se la coloca, junto con su descendencia, en lugar de ésta (de la eliminada); luego, se elimina "recursivamente" la clave sucesora, para lo cual se aplica a), ya que a lo más tiene un hijo.

Otra forma de eliminar una clave cuyo nodo tiene ambos hijos es *eliminación por mezcla*: el nodo es ocupado por su (hijo y) subárbol izquierdo, *Ti*, mientras que su subárbol derecho, *Td*, se convierte en el subárbol derecho del nodo más a la derecha de *Ti*.

c) [**2 pts.**] Justifica que esta eliminación respeta las propiedades de ABB.

A partir de la regla para insertar claves, que, a su vez, obedece a la propiedad fundamental de ABB, sabemos que en el proceso de inserción de cualquiera de las claves que están en Ti o Td —llamemos k a una clave cualquiera en Ti o Td— pasamos por la clave —llamémosla j— que estamos eliminando, y que, por lo tanto, k podría haber ido a parar al lugar de j, posición en la que habría cumplido la propiedad de ABB con respecto al resto del árbol. Por lo tanto, poner el subárbol Ti en la posición que ocupaba el nodo con la clave j es perfectamente válido.

La pregunta entonces es, ¿qué hacemos con Td? De nuevo, por la propiedad de ABB, las claves de Td son todas mayores que las claves de Ti. La única posición que corresponde a claves mayores que todas las claves de Ti, pero al mismo tiempo menores que las otras claves del árbol mayores que j es como hijo derecho de la clave más a la derecha de Ti —llamémosla m; obviamente, esta posición está "desocupada": m sólo puede ser la clave más a la derecha de Ti si no tiene hijo derecho.

d) [**2 pts.**] Muestra con ejemplos que esta eliminación puede tanto aumentar como reducir la altura del árbol original.

Para simplificar (y generalizar un poco), eliminamos la raíz del árbol; Ti "sube" a esta posición y agregamos Td como hijo derecho del nodo más a la derecha de Ti. La altura del árbol original, T, era H(T) = max{ H(Ti), H(Td) } + 1. La altura del nuevo árbol, T', puede ser desde H(T') = H(Ti) hasta H(T') = H(Ti) + H(Td), dependiendo de la profundidad del nodo más a la derecha de Ti. En el primer caso, H(T') es claramente menor que H(T); en el segundo, H(T') claramente puede ser mayor que H(T).

2013-2-Ex-A–AVL, Árbol rojo-negro (1/1)

Examen (preguntas adicionales)

Estructuras de Datos y Algoritmos – IIC2133

2 diciembre 2013

Reemplazo de I2

A) Con respecto a los árboles rojo-negro:

a) ¿Cuántos cambios de color y cuántas rotaciones pueden ocurrir a lo más en una inserción? Justifica.

Al insertar un nodo, x, lo insertamos como una hoja y lo pintamos de rojo. Si su padre, p, es negro, terminamos. Si p es rojo, tenemos dos casos: el hermano, s, de p es negro; s es rojo. Si s es negro, realizamos algunas rotaciones y algunos cambios de color. Si s es rojo, sabemos que el padre, g, de p y s es negro; entonces, cambiamos los colores de g, p y s, y revisamos el color del padre de g.

Hacemos O(log n) cambios de color y O(1) rotaciones. Como dice arriba, esto ocurre solo cuando p es rojo.

Si s es negro, hacemos exactamente una o dos rotaciones, dependiendo de si x es el hijo izquierdo o el hijo derecho de su padre; y hacemos exactamente dos cambios de color.

Si s es rojo, no hacemos rotaciones, solo cambios de color. Inicialmente, cambiamos los colores de tres nodos: g, que queda rojo, y sus hijos p y s, que quedan negros. Como g queda rojo, hay que revisar el color de su padre: si es negro, terminamos; si es rojo, repetimos estos últimos cambios de color, pero más "arriba" en el árbol.

b) Justifica que los nodos de cualquier árbol AVL T pueden ser pintados "rojo" y "negro" de manera que T se convierte en un árbol rojo-negro.

Hay justificar que en un árbol AVL la ruta (simple) más larga de la raíz a una hoja no tiene más del doble de nodos que la ruta (simple) más corta de la raíz a una hoja.

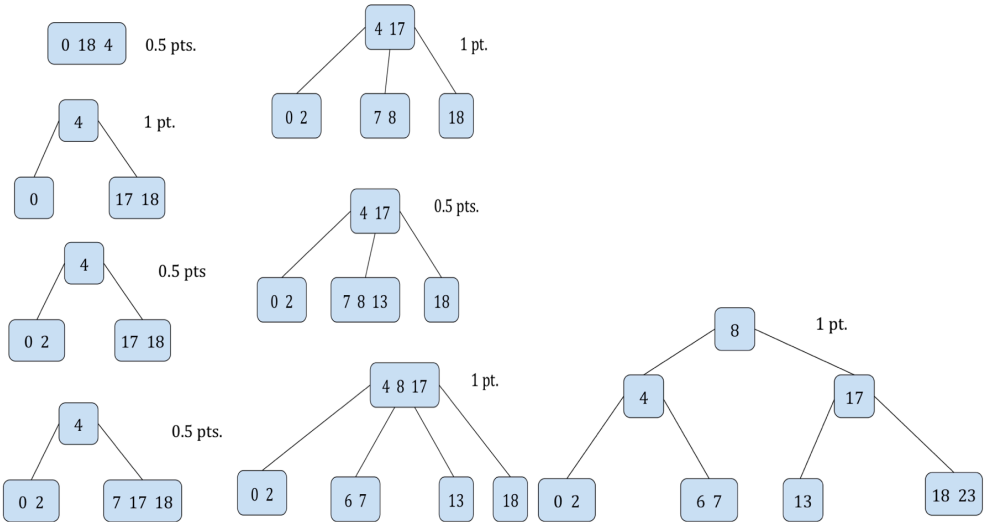
Esto es así: el árbol AVL más "desbalanceado" es uno en el que todo subárbol derecho es más alto (en uno) que el correspondiente subárbol izquierdo (o vice versa). En tal caso, el número de nodos en la ruta más larga crece según 2h, y el de la ruta más corta, según h+1, en que h es la altura del árbol.

Así, para cualquier ruta (simple) desde la raíz a una hoja, sea d la diferencia entre el número de nodos en esa ruta y el número de nodos en la ruta más corta. Si todos los nodos de la ruta más corta son negros, entonces los otros d nodos deben ser rojos: pintamos la hoja roja y, de ahí hacia arriba, nodo por medio hasta completar d nodos rojos.

2013-1-I2-P2–Árbol 3-4 (1/1)

2. Dibuja los árboles-B resultantes al ir insertando las claves 4, 17, 2, 7, 8, 13, 6 y 23 en un árbol-B de grado mínimo $t = 2$ que inicialmente contiene un solo nodo con las claves 0 y 18.

Respuesta. El puntaje por cada nueva configuración correcta es el siguiente:



2013-1-I1-P3–Heap (1/1)

3. Considera la siguiente implementación de un cola priorizada. Los elementos que están en la cola ocupan posiciones consecutivas en un arreglo *a*, a partir de la casilla *a*[0] —el arreglo se mantiene compacto. Cuando ingresamos un nuevo elemento a la cola, simplemente lo ponemos a continuación del elemento que ocupa la última casilla. Cuando sacamos un elemento de la cola, buscamos secuencialmente a partir de *a*[0] el elemento con mayor prioridad y lo sacamos; para mantener el arreglo compacto, ponemos el elemento que está en la última casilla en el lugar del que sacamos.

a) **[2 puntos]** Compara esta implementación con la que estudiamos en clase, basada en un heap binario. En parti-cular, ¿cuál es la complejidad para las operaciones *xMax( )*, *insertObject( )* e *incrementKey( )* en cada una de estas implementaciones? ¿Bajo qué condiciones conviene usar la implementación descrita en el párrafo anterior?

Supongamos que la cola tiene *n* elementos. En el caso del heap binario, las tres operaciones son O(log *n*), como vimos en clase.

En cambio, en la implementación de más arriba, *insertObject()* e *incrementKey()* son O(1), pero *xMax()* es O(*n*). Recordemos que *incrementKey()* recibe como parámetro el objeto al que se le va a incrementar su prioridad, además del valor del incremento; por lo tanto, no es necesario buscar este objeto en la cola.

Conviene usar esta nueva implementación si la gran mayoría de las operaciones son *insertObject()* e *incrementKey()*, y sólo se hace uno que otro *xMax()*.

b) **[2 puntos]** ¿Cambian tus respuestas si en lugar del arreglo *a* usamos una lista doblemente ligada? ¿Por qué?

No cambian. En particular, *insertObject()* ahora puede insertar al comienzo de la lista doblemente ligada (en lugar de a continuación del último elemento), pero sigue siendo O(1); y *xMax()* no necesita "rellenar" el espacio dejado por el elemento sacado de la cola (las listas doblemente ligadas se mantienen naturalmente compactas), pero sigue siendo O(*n*).

c) **[2 puntos]** Otra operación sobre colas priorizadas es *join( )*, que consiste en unir dos colas priorizadas, compues-tas por elementos del mismo tipo, en una nueva cola priorizada; las prioridades de cada elemento se mantienen, pero la posición relativa de la prioridad de un elemento en la nueva cola puede cambiar. ¿Cuál es la complejidad de *join( )* en cada una de las tres implementaciones mencionadas? Explica.

En el caso de las listas doblemente ligadas, *join()* toma tiempo O(1), ya que basta conectar el último elemento de una de las colas al primero de la otra, cambiando un número fijo de punteros.

En el caso del arreglo descrito en el enunciado, *join()* toma tiempo O(*n*), en que *n* es el número de elementos de la cola más corta, ya que hay que pasar cada uno de estos elementos a la otra cola: hay que ejecutar *n insertObject's*.

En el caso del heap binario, *join()* toma tiempo O(*n*), en que *n* es el número total de elementos an ambas colas: primero, hay que traspasar los elementos de la cola más corta a la otra cola, en tiempo proporcional al número de elementos de la cola más corta; luego, como vimos en clase, armar un heap binario a partir de un arreglo (desordenado) de *n* elementos es O(*n*). Si la cola más corta es mucho más corta, se puede hacer *insertObject()* de cada uno de sus elementos en la otra cola, en tiempo total O(*n'* log *n*), en que *n'* es el número de elementos de la cola más corta.

Tiempo: 105 minutos

2013-1-I2-P3–ABB (1/1)

3) Explica cómo se podría resolver el problema de selección si los datos están almacenados en un árbol binario de búsqueda (ABB), y, por lo tanto, están ordenados. Así, si no hacemos nada especial con el árbol, cuando nos pidan el k-ésimo dato más pequeño (éste es el *problema de selección*), podríamos hacer un recorrido del árbol en preorden y detenernos cuando hayamos impreso los k datos más pequeños. Sin embargo, este procedimiento tiene complejidad O(n), tanto en promedio, como en el peor caso.

Por lo tanto, la idea es agregar información adicional al árbol, que, aprovechando el hecho de que es un ABB, permita determinar más eficientemente el k-ésimo dato más pequeño. Específicamente:

a) ¿Qué información adicional es necesario almacenar en el árbol?

Respuesta:

Si nos basamos en la estrategia que usa *randomSelect*, podemos almacenar en un campo *size* de cada nodo r del árbol el número total de nodos que hay en el subárbol cuya raíz es r. Este número es igual a la suma del número de nodos que hay en el subárbol izquierdo de r ($r.left.size$) más el número de nodos que hay en el subárbol derecho de r ($r.right.size$) más 1 (el propio r).

b) ¿Cómo se determina el k-ésimo dato más pequeño en este nuevo escenario?

Respuesta:

Si nos basamos en la estrategia que usa *randomSelect*, comparamos k con $r.size$: si son iguales, entonces la respuesta es $r.key$; si $k < r.size$, entonces buscamos el k-ésimo elemento más pequeño recursivamente en el subárbol izquierdo de r; y si $k > r.size$, entonces buscamos el $(k - (r.left.size + 1))$-ésimo elemento más pequeño recursivamente en el subárbol derecho de r.

c) ¿Cuál es la complejidad (de tiempo) de la operación anterior?

Respuesta:

O(h), en que h es la altura del árbol: hace un número constante de comparaciones en un nodo, y de ahí pasa al hijo izquierdo o derecho de ese nodo, si es que pasa, en donde repite el procedimiento. Es decir, en el peor caso hace un número constante de comparaciones por nivel del árbol.

d) ¿Cuánto cuesta mantener actualizada la información adicional del árbol cuando se produce una inserción o una eliminación de un dato?

Respuesta:

Básicamente, sumar (o restar) 1 al campo *size* de cada nodo en la ruta desde la raíz del árbol hasta el nodo insertado (o eliminado).

2013-1-I2-P4–AVL (1/1)

4) Con respecto a los árboles AVL:

a) Muestra la secuencia de árboles AVL's que se forman al insertar las claves 3, 2, 1, 4, 5, 6, 7, 16, 15 y 14, en este orden, en un árbol AVL inicialmente vacío.

Las inserciones de 3 y 2 son triviales (no tienen puntaje). La inserción de 1 exige una rotación simple a la derecha. La inserción de 4 nuevamente es trivial. Las inserciones de 5, 6 y 7 exigen rotaciones simples a la izquierda. La inserción de 16 es trivial. Finalmente, las inserciones de 15 y 14 exigen rotaciones dobles: en ambos casos, una rotación simple a la derecha seguida de una rotación simple a la izquierda. El árbol AVL resultante tiene como raíz a 4, cuyos hijos son 2 y 7. Los hijos de 2 son 1 y 3; los de 7 son 6 y 15. El único hijo de 6 es 5; los hijos de 15 son 14 y 16.

b) Se ha producido la inserción de un nuevo nodo x en un árbol AVL. Escribe el método

Node determinarPivote(*Node x*)

que a partir de x sube por el árbol, actualizando los balances de cada nodo encontrado en el camino, hasta llegar al primer nodo que quedó desbalanceado como consecuencia de la inserción de x; el método retorna este nodo (para permitir la ejecución de la rotación que corresponda).

Resouesta: Pendiente.

Tiempo: 120 minutos

2013-1-Ex-A–Heap, HeapSort (1/1)

Estructuras de Datos y Algoritmos – IIC2133

Examen

25 junio 2013

A) En clase estudiamos los métodos *buildHeap*(*b*), que construye un **max-heap binario** a partir de un arreglo *b*, e *insertObject*(*x*), que inserta un nuevo elemento *x* en un max-heap. Además, en lugar de llamar a *buildHeap*(*b*), podríamos construir un max-heap llamando repetidamente a *insertObject*( ), así:

```
buildHeap'(b):
    heapSize = 1
    for i = 2 to b.length
        insertObject(b[i])
```

1) ¿Cuál es la complejidad de *buildHeap*( ), *insertObject*( ), y *buildHeap'*( ), en función del número $n = b.length$ de elementos procesados? Justifica.

buildHeap( ) hace O(n) operaciones. Este método en la práctica ejecuta **for** $j = n/2$ **downto** 1 *heapify*(j), es decir, hace *heapify*( ) sobre $n/2$ elementos. Cada *heapify*( ) hace un número constante de operaciones más una llamada recursiva. La mitad de estos *heapify*( ), es decir, $n/4$, sólo trabaja en un nivel; en la práctica, no desciende recursivamente. La mitad de los $n/4$ restantes, es decir, $n/8$, descienden recursivamente sólo un nivel; y así sucesivamente.

insertObject(( ) hace O(log$n$) operaciones. Este método coloca el nuevo objeto al final de un heap ya formado, y luego tiene que ubicarlo correctamente según su clave. Para esto, repetidamente, hace un número constante de operaciones en el nivel del árbol en el que está actualmente, y luego sube un nivel; así, hasta llegar a la raíz del max-heap. Hay O(log$n$) niveles en el árbol.

buildHeap'( ) hace O($n$log$n$) operaciones, ya que ejecuta n veces *insertObject*( ).

2) Con respecto a *buildHeap*( ) y *buildHeap'*( ), ¿construyen el mismo max-heap cuando son ejecutados sobre el mismo arreglo *b* de entrada? Demuestra que es así, o da un contraejemplo.

En general, no construyen el mismo heap. P.ej., si el arreglo original tiene los elementos [1, 2, 3], uno construye el heap [3, 1, 2]; y el otro, el heap [3, 2, 1].

3) [*Independiente de las anteriores*] ¿Cuál es el mejor caso para *hespSort*( ) y qué tan bueno es? Explica.

Cuando todos los elementos son iguales, *heapify*( ) no desciende recursivamente desde la raíz, y por lo tanto *heapSort*( ) corre en tiempo O(n).

2013-1-Ex-B–Árbol rojo-negro, ABB, Árbol 2-4 (1/3)

B) ¿Cuánto cuesta y cómo hay que hacerlo para comprobar si una estructura de datos dada es un **árbol** (de búsqueda binario) **rojo-negro**? En particular,

4) ¿Cuál es la propiedad de árbol de búsqueda binario?
Cualquier clave en el subárbol izquierdo es menor que la clave en la raíz, que a su vez es menor que cualquier clave en el subárbol derecho

5) Escribe un método *esABB*( ), que reciba un nodo como parámetro y devuelva *true* si el nodo es la raíz de un árbol de búsqueda binario; *false*, en otro caso.

Lo más simple puede ser escribir un método recursivo:

```
esABB(x):
    if ( null(x) ) return true
    if ( hoja(x) ) return true
    if ( esABB(x.derecho) && esABB(x.izquierdo) )
        if ( null(x.derecho) ) return x.key < x.izquierdo.key
        if ( null(x.izquierdo) ) return x.derecho.key < x.key
        return x.derecho.key < x.key && x.key < x.izquierdo.key
    else return false
```

6) ¿Qué propiedades adicionales debe cumplir un árbol rojo-negro?

i) Todo nodo es ya sea rojo o negro.
ii) Si un nodo es rojo, entonces sus hijos son negros.
iii) Para cada nodo, todas las rutas desde el nodo a las hojas descendientes contienen el mismo número de nodos negros.

2013-1-Ex-B–Árbol rojo-negro, ABB, Árbol 2-4 (2/3)

7) Escribe un método *esRojo-Negro*( ), que reciba un nodo como parámetro y devuelva *true* si el nodo es la raíz de un árbol de búsqueda binario rojo-negro; *false*, en otro caso.

Tal vez, lo más simple es escribir dos métodos recursivos, uno para verificar ii) y el otro para verificar iii); cualquiera de ellos puede, además, verificar i):

```
esRojo-Negro(x):
    if ( esABB(x) ) return cumple-ii(x) && cumple-iii(x)!=-1
    else return false

cumple-ii(x): —también verifica i)
    if ( null(x) ) return true
    if ( hoja(x) ) return es rojo o negro
    if ( es negro ) return cumple-ii(x.derecho) && cumple-ii(x.izquierdo)
    if ( es rojo )
        if ( null(x.izquierdo) ) return cumple-ii(x.derecho) && x.derecho es negro
        if ( null(x.derecho) ) return cumple-ii(x.izquierdo) && x.izquierdo es negro
        return cumple-ii(x.derecho) && x.derecho es negro && cumple-ii(x.izquierdo) && x.izquierdo es negro
    else return false

cumple-iii(x)
    if( null(x) ) return 0
    if( es rojo && hoja(x) ) return 0
    if( es negro && hoja(x) ) return 1
    p = cumple-iii(x.izquierdo)
    q = cumple-iii(x.derecho)
    if (p!=q) return -1
    if(p== -1 || q==-1) return -1
    if( es rojo ) return p
    if( es negro) return p+1
```

2013-1-Ex-B–Árbol rojo-negro, ABB, Árbol 2-4 (3/3)

8) [*Independiente de las anteriores*] Un caso particular de árboles-B son los árboles 2-4, en que un nodo puede tener entre 1 y 3 claves y entre 2 y 4 hijos (árbol-B con $t = 2$). Supongamos que hacemos lo siguiente: Los nodos con 2 claves los separamos en dos nodos, con una clave cada uno, en que el nodo con la clave mayor queda como padre del nodo con la clave menor, y distribuimos los tres hijos del nodo original entre estos dos nodos, de modo que ambos queden con dos hijos cada uno. Los nodos con 3 claves los separamos en tres nodos, con una clave cada uno, en que el nodo con la clave central queda como padre de los otros dos, y distribuimos los cuatro hijos del nodo original entre los dos nuevos nodos hijos, de modo que los tres nuevos nodos quedan con dos hijos cada uno. Justifica que el nuevo árbol es un árbol rojo-negro.

Aquí hay que partir por la propiedad de los árboles-B, de que todas las hojas están a la misma profundidad (desde la raíz); es decir, todas las rutas desde la raíz hasta una hoja tienen el mismo número de nodos. No es casualidad que esta propiedad se "parezca" mucho a la propiedad iii), en 6). Por lo tanto, estos nodos, o lo que quede de ellos después de las separaciones sugeridas en el enunciado, serán los nodos negros del nuevo árbol: el mismo número de ellos en cada ruta desde la raíz hasta una hoja.

En el caso de los nodos con dos o tres claves, que en el nuevo árbol separamos en un padre y uno o dos hijos, el nodo que queda como padre mantiene el color negro; los nuevos nodos hijos de este padre negro (uno o dos, que compartían con él el nodo original) serán los nodos rojos del árbol rojo-negro. Es fácil ver que los hijos de estos nodos rojos son negros, tal como lo exige la propiedad ii), en 6).

2012-1-I2-P2–Funciones sobre árbol (1/2)

IIC2133 - Estructuras de datos y algoritmos I-2012 – Interrogación 2

Pregunta 2 – Árboles (20 pts)

Considera la siguiente representación de árbol (estracto):

```
class Tree {
    public:
        double value();
        List<Tree>* children();
    private:
        double nodeValue;
        List<Tree>* childrenList;
}
```

Construye la clase TreeStatistics, que tiene los siguientes dos métodos:

```
double average(const Tree& tree);
double total(const Tree& tree);
```

los cuales, respectivamente, entregan el promedio de los valores del árbol y el total de los valores del árbol.

```
class TreeStatistics {
public:
    /** 10 pts */
    double average(const Tree& tree) {
        return total(tree) / count(tree);
    }
    /** 10 pts */
    double total(const Tree& tree) {
        double subtotal = tree.value();
        for (Iterator<Tree> it = tree.children().iterator(); it.hasNext(); ) {
            subtotal += total( it.next() );
        }
        return subtotal;
    }
```

2012-1-I2-P2–Funciones sobre árbol (2/2)


```
private:
        int count(const Tree& tree) {
                int counted = 1;
                for (Iterator<Tree> it = tree.children().iterator(); it.hasNext(); ) {
                        counted += count( it.next() );
                }
                return counted;
        }
}
```

2012-1-I2-P3–Árbol rojo-negro, AVL, ABB (1/2)

Pregunta 3 – Árboles de búsqueda binaria (20 pts)

a) (9 pts) El algoritmo de inserción de un árbol rojo negro podría dividirse en tres etapas.

Describe muy brevemente estas etapas (en qué consiste y cual es su objetivo o función).

1 .- Descender hasta el lugar de inserción, aplicando cambio de color cada vez que hay dos hermanos rojos, y arreglando con rotación simple o roble. Su objetivo es llegar al lugar donde se insertará el nodo, evitando que el padre del nodo sea rojo y tenga un hermano rojo. (3 pts)

2.- Insertar un nodo rojo. (3 pts)

3.- Si el padre es rojo, balancear con rotación simple o doble dependiendo del caso. (3 pts)

b) (5 pts) Tanto un árbol AVL como un árbol rojo negro son árboles ABB auto balanceados, cuyas operaciones (insertar / eliminar) no son triviales de implementar. Entonces, ¿por qué construir o usar una estructura tan complicada cuando existe otra mucho más simple (un árbol ABB “normal”) que cumple la misma función? (Responde brevemente)

Porque es más eficiente: al mantenerse balanceado, sus operaciones son O(log(n)), mientras que un ABB normal puede desbalancearse, haciendo que sus operaciones sean O(n) en el peor caso.

c) (6 pts) Considera el siguiente caso hipotético: “alguien” desarrolló una librería en C++ con una interfaz de árboles ABB (con operaciones para insertar, eliminar, consultar existencia y obtener mínimo y máximo) y 3 implementaciones: con un árbol ABB simple, un árbol AVL y un árbol rojo-negro. Considera también la siguiente función, que recibe un valor y una cota inferior y superior y verifica si una base de datos contiene el valor y si todos los valores de la base de datos están entre las cotas entregadas:

```
boolean containsAndInRange(int value, int lowerBound,
                int upperBound) {
     DataSource* ds = InMemoryDatabase::datasource();
     ABB* abb = ____________________;
     return abb->find(value)
```

2012-1-I2-P3–Árbol rojo-negro, AVL, ABB (2/2)


```
                    && lowerBound <= abb->min()
                    && abb->max() <= upperBound;
    }
```

Teniendo esto en cuenta, puedes completar la función con una de estas opciones:

i. ds->getDataInSimpleABBTree()

ii. ds->getDataInAVLTree()

iii. ds->getDataAsRedBlackTree()

Elige una opción, y explica (brevemente) por qué la elegiste por sobre las otras 2.

Dado que lo único que se hará con el árbol es buscar (find) y obtener los valores máximo y mínimo, nos basta con un árbol que esté balanceado. Por esta razón, tanto un AVL como un rojo negro son igualmente válidos como solución. Dado que da igual entre estos 2, es preferible en primer lugar el que sea “nativo” del datasource, y en caso que ambos tengan que ser construidos por el datasource, el Rojo-Negro es preferible, pues es más eficiente en inserción.

2012-1-I2-P4–AVL, Árbol rojo-negro (1/2)

Pregunta 4 – Uso de estructuras (20 pts)

En esta pregunta, considera que tienes las siguientes estructuras a tu disposición: ArrayList, LinkedList, DoublyLinkedList, Queue, Stack, Heap, HashMap, ABBTree, AVLTree, RedBlackTree y BTree.

Un programa lee enteros desde dos archivos, cada uno con un número variable entre 1 y 5 millones de datos. El programa debe escribir en un tercer archivo la lista de todos los números, en orden ascendente, que estén en alguno de los dos archivos. Así, por ejemplo, si un archivo contiene 67, 11, y 42 y el otro contiene 11 y 17, el programa deberá escribir un archivo con 11, 17, 42 y 67.

Elige la o las estructura(s) que mejor se ajusta(n) al problema. Para cada estructura, responde brevemente:

a) ¿Que rol desempeña?

b) ¿Por qué esa estructura y no otra?

c) Si esa estructura no estuviera disponible, ¿con cual la reemplazarías? ¿por qué?

Si crees que ayudaría a entender tu respuesta, escribe en seudo código cómo harías este programa usando las estructuras que elegiste.

Algoritmo:

Sea n = número total de datos.

Acá es importante identificar que, dado que los datos están en archivos, estaremos obligados a leerlos secuencialmente.

2012-1-I2-P4–AVL, Árbol rojo-negro (2/2)

IIC2133 - Estructuras de datos y algoritmos I-2012 – Interrogación 2

Si usamos un ABB balanceado, podemos hacer que las operaciones sean O(log(n)) cada una, y al leer secuencialmente un archivo y luego el otro, tendremos ~ n*log(n) operaciones. Si luego recorremos el árbol inorden (lo cual es O(n)), tendremos los datos ordenados.

Estructura(s): Árbol Rojo-Negro.

a) Rol: Los datos cada archivo se leen secuencialmente y se ingresan al árbol (primero un archivo, luego el segundo).

b) Porque permite hacer la carga de los datos en O(n*log(n)) operaciones, y obtener los datos ordenados en O(n). En comparativa con otras estructuras, un AVL es del mismo orden, pero sabemos que un AVL requiere más operaciones en inserción que un rojo negro (en el peor caso, aproximadamente el doble). Un heap también es del mismo orden de inserción, pero obtener los datos ordenados desde el heap requiere O(n*log(n)) operaciones.

Una tabla hash que usa como clave y valor los datos también suena atractivo (sus operaciones son ~O(1), manteniendo el factor de carga bajo 0.75). Sin embargo, esto tiene 2 inconvenientes:

i: si no inicializamos la tabla con el máximo n posible / 0.75, la tabla tendrá que hacer resize, y la operación de resize de una tabla hash es similar a la de un ArrayList : O(n). Así que o desperdiciamos mucho espacio (con una tabla de 13,3 millones de entradas, cuando podrían haber sólo 2 millones de datos), o resentimos el performance de la tabla hash.

Ii: la tabla hash no está ordenada, por lo que estaríamos obligados a ordenar los datos luego. Y sabemos que ordenar, en el mejor caso, es O(n*log(n)) (usando heapsort). Este argumento es fatal para todas las estructuras no ordenadas.

c) Con un AVL, por que inserción y obtener los datos ordenados sigue siendo O(n*log(n)) y O(n) respectivamente, mejor que cualquier otra estructura.

2011-2-I1-P3–ABB balanceado (1/1)

3. Un árbol binario de búsqueda (ABB) se dice **balanceado** si la diferencia en altura de ambos subárboles de cualquier nodo es cero o uno (p.ej., un árbol AVL). Un ABB se dice **perfectamente balanceado** si es balanceado y todas sus hojas están en un mismo nivel o, a lo más, en dos niveles (p.ej., un *heap* binario). Da un algoritmo para convertir cualquier ABB en un ABB perfectamente balanceado.

Para esto, considera un ABB totalmente desbalanceado, convertido en una lista ligada hacia la derecha, con 2^k–1 nodos, para algún entero k. Partiendo desde la raíz, baja por la lista y haz una rotación simple a la izquierda cada dos nodos (la primera rotación es del hijo derecho de la raíz con respecto a la raíz; la segunda rotación involucra a los dos nodos siguientes en la lista; y así sucesivamente).

¿Qué ha pasado con los desbalances cuando llegas abajo? ¿Qué tendrías que hacer a continuación para convertir finalmente el árbol en uno perfectamente balanceado y completo? ¿Cómo manejarías el caso de un árbol que no puede ser completo, es decir, que su número de nodos no puede escribirse como 2^k–1?

La sugerencia que parte en el segundo párrafo es como la 'segunda' pare del algoritmo pedido. La primera parte consiste en convertir el árbol original en una lista ligada hacia la derecha; esto lo vimos en la ayudantía de 25/8:

`while (` haya un nodo x con un hijo izquierdo y `)`
hacer una rotación simple a la derecha, de y con respecto a x

Como cada rotación simple a la derecha reduce en uno el número de hijos izquierdos en el árbol, a lo más n–1 rotaciones son necesarias para convertir todo el árbol en una lista ligada hacia la derecha.

Una vez hecho esto, se aplica el algoritmo sugerido. Las rotaciones sugeridas, a lo largo de la lista y 'nodo por medio', van reduciendo paulatinamente el desbalance: las primera rotación reduce el desbalance de la raíz en dos; las siguientes, en uno cada una. Así, al llegar abajo, el desbalance de la raíz se ha reducido de n–1 a n–3–$\lfloor n/2 - 1 \rfloor$. También se han reducido los desbalances de los subárboles. Lo que hay que hacer a continuación es repetir el proceso, desde la raíz, por la rama más a la derecha del árbol; el número rotaciones a realizar en esta segunda pasada va a ser la mitad de las de la primera pasada, y el árbol va a quedar aún un poco mejor balanceado. Y así sucesivamente, hasta formar un árbol completo (ya que n es de la forma 2^k–1) perfectamente balanceado.

Si el árbol original no es completo, hay que tratar el número de nodos 'sobrantes' como casos especiales, p.ej., haciendo ese número de rotaciones a la izquierda justo antes de aplicar el algoritmo sugerido.

2011-2-I1-P4–Árbol rojo-negro (1/1)

4. Uno de los ejercicios propuestos en el texto de Cormen et al. afirma que si un árbol rojo-negro de más de un nodo ha sido construido sólo a través de operaciones de inserción (sin eliminaciones), entonces el árbol tendrá al menos un nodo rojo. Muestra con un ejemplo que un árbol rojo-negro de más de un nodo puede tener sólo nodos negros si su construcción ha incluido eliminaciones.

Supongamos un árbol inicialmente vacío, al que insertamos las claves 3, 2 y 4, en este orden: 3 queda en la raíz, y es negro, y 2 y 4 son sus hijos izquierdo y derecho, respectivamente, ambos rojos. Si ahora insertamos la clave 1, esta va a parar como hijo izquierdo de 2, también rojo. Al ser rojos 2 y su hijo 1, hay que hacer algo; en este caso, como el hermano de 2, 4, también es rojo, intercambiamos colores por niveles: pintamos negros 2 y 4 y pintamos rojo 3. Pero como 3 es la raíz, lo volvemos a pintar negro. En este momento, sólo 1 es rojo.

Si ahora eliminamos 1, simplemente eliminamos una hoja roja, por lo que no es necesario hacer ninguna otra operación en el árbol, que así queda sólo con tres nodos negros.

2011-2-I3-P4–Nuevo árbol de ordenación (1/1)

4. Cambiemos levemente la definición de un árbol-B. Definamos el *orden* del árbol como el número máximo de hijos que puede tener un nodo. Entonces, un árbol-B de orden m tiene las siguientes propiedades: la raíz tiene al menos dos hijos, excepto si es una hoja; cada nodo interno (que no sea la raíz) tiene k–1 claves y k referencias a hijos, en que $\lceil m/2 \rceil \leq k \leq m$; cada hoja tiene k–1 claves, en que $\lceil m/2 \rceil \leq k \leq m$. P.ej., en un árbol-B de orden $m = 5$, cada nodo interno tiene entre 2 y 4 claves y entre 3 y 5 hijos, y cada hoja tiene entre 2 y 4 claves.

Supón que comenzamos con un árbol-B de orden 5 vacío.

a) Muestra el árbol después de insertar las claves 8, 14, 2, 15.

b) Muestra el árbol después de insertar las claves 3, 1, 16, 6, 5.

c) Muestra el árbol después de insertar las claves 27, 37, 18, 25, 7, 13, 20.

2011-2-Ex-P2–ABBB (1/1)

2. Supón que tienes n votos para presidente del centro de alumnos, en que cada voto es el número de alumno (un entero) del candidato.

a) Sin saber quiénes son los candidatos ni cuántos candidatos hay, da un algoritmo de tiempo $O(n\log n)$ para determinar al ganador, usando a lo más $O(n)$ memoria extra.

Ver **b**).

b) Si ahora sabes que hay $k < n$ candidatos, da un algoritmo de tiempo $O(n\log k)$ para determinar al ganador, usando a lo más $O(k)$ memoria extra.

A medida que vamos revisando los votos, los vamos insertando en un ABBB: cada inserción en un ABBB de (a lo más) k elementos toma tiempo O(logk). Si el número de alumno del voto ya está en el ABBB, entonces no lo insertamos (nuevamente), sino que incrementamos un contador.

2011-2-Ex-P3–AVL, Árbol rojo-negro (1/1)

3. Justifica que los nodos de cualquier árbol AVL T pueden ser pintados "rojo" y "negro" de manera que T se convierte en un árbol rojo-negro.

Hay justificar que en un árbol AVL la ruta (simple) más larga de la raíz a una hoja no tiene más del doble de nodos que la ruta (simple) más corta de la raíz a una hoja.

Esto es así: el árbol AVL más "desbalanceado" es uno en el que todo subárbol derecho es más alto (en uno) que el correspondiente subárbol izquierdo (o vice versa). En tal caso, el número de nodos en la ruta más larga crece según $2h$, y el de la ruta más corta, según h+1, en que h es la altura del árbol

Así, para cualquier ruta (simple) desde la raíz a una hoja, sea d la diferencia entre el número de nodos en esa ruta y el número de nodos en la ruta más corta. Si todos los nodos de la ruta más corta son negros, entonces los otros d nodos deben ser rojos: pintamos la hoja roja y, de ahí hacia arriba, nodo por medio hasta completar d nodos rojos.

2010-2-I1-P2–ABB (1/1)

2. Demuestra que cualquier ABB arbitrario de n nodos puede ser transformado en cualquier otro ABB arbitrario de n nodos usando $O(n)$ rotaciones.

a) Primero, prueba que con a lo más n–1 rotaciones a la derecha puedes convertir cualquier ABB en uno que es simplemente una lista de n nodos ligada por los punteros a los hijos derechos.

*Llamemos **lista derechista** a la lista de nodos ligada por los punteros a los hijos derechos; queremos convertir el árbol en una lista derechista de n nodos. Inicialmente, la lista derechista tiene al menos un nodo: la raíz del árbol.*

Cada rotación a la derecha respecto de un nodo que está en la lista derechista, y tiene un hijo izquierdo, agrega ese hijo izquierdo a la lista derechista. La lista comienza con al menos un nodo, que es la raíz. Por lo tanto, con a lo más n–1 rotaciones a la derecha, todos los nodos han pasado a la lista.

Ahora observa que si conocieras la secuencia de rotaciones a la derecha que transforman un ABB arbitrario T en una lista ligada T', entonces podrías ejecutar esta secuencia en orden inverso, cambiando cada rotación a la derecha por su rotación inversa a la izquierda, para transformar T' de vuelta en T.

b) Explica cómo transformar un ABB T_1 en otro T_2, pasando por la lista ligada única T' de nodos de T_1 (que es la misma que la de los nodos de T_2). ¿Cuántas rotaciones a lo más es necesario hacer para esto?

*T' es la lista derechista. Como vimos en **a**), podemos convertir T1 en T' con a lo más n–1 rotaciones a la derecha. Como los nodos de T2 son los mismos que los de T1, entonces también podríamos convertir T2 en T' con a los más n–1 rotaciones a la derecha. Si a partir de T' aplicamos estas últimas n–1 rotaciones en orden inverso, y cada vez rotamos a la izquierda en vez de la derecha, obtenemos T2.*

2010-2-I1-P3–Árbol rojo-negro (1/1)

3. ¿Cuántos cambios de color y cuántas rotaciones pueden ocurrir a lo más en una inserción en un árbol rojo-negro? Justifica tus respuestas.

Recuerda que al insertar un nodo, x, lo insertamos como una hoja y lo pintamos de rojo. Si el padre, p, de x es negro, terminamos. Si p es rojo, tenemos dos casos: el hermano, s, de p es negro; s es rojo. Si s es ne-gro, realizamos algunas rotaciones y algunos cambios de color. Si s es rojo, sabemos que el padre, g, de p y s es negro; entonces, cambiamos los colores de g, p y s, y revisamos el color del padre de g.

Hacemos O(log n) cambios de color y O(1) rotaciones. Como dice arriba, esto ocurre solo cuando p *es rojo.*

Si s *es negro, hacemos exactamente una o dos rotaciones, dependiendo de si* x *es el hijo izquierdo o el hijo derecho de su padre; y hacemos exactamente dos cambios de color.*

Si s *es rojo, no hacemos rotaciones, solo cambios de color. Inicialmente, cambiamos los colores de tres nodos:* g*, que queda rojo, y sus hijos* p *y* s*, que quedan negros. Como* g *queda rojo, hay que revisar el color de su padre: si es negro, terminamos; si es rojo, repetimos estos últimos cambios de color, pero más "arriba" en el árbol.*

2010-2-I1-P4–Max-Heap (1/1)

4. Un arreglo a tiene inicialmente el siguiente contenido:

$$a = [\,0\ \ 18\ \ 14\ \ 17\ \ 19\ \ 8\ \ 13\ \ 6\ \ 4\ \ 23\ \ 0\ \ 12\ \ 15\,].$$

Transfórmalo en un *max-heap* binario usando el siguiente algoritmo:

```
for (k = 5; k >= 0; k = k-1) heapify(k)
```

Muestra el resultado, como árbol binario, después de cada iteración.

2010-2-Ex-P4–Árbol rojo-negro (1/1)

4) Con respecto a los árboles rojo-negro:

a) Supón que "absorbemos" todo nodo rojo dentro de su padre negro, de modo que los hijos del nodo rojo son ahora hijos del padre negro. [**2** pts.] ¿Cuáles son los posibles grados de un nodo negro después de que todos sus hijos rojos son absorbidos? [**1** pt.] ¿Qué pasa con la profundidad de las hojas del árbol resultante?

b) Supón que insertamos un nodo x en un árbol rojo-negro y luego lo eliminamos inmediatamente. ¿Es el árbol resultante el mismo que el inicial? Justifica tu respuesta.

a) [**2** pts.] El grado grado de un nodo negro después de absorber sus hijos rojos es

2, si sus hijos eran negros
3, si un hijo era negro y el otro rojo
4, si ambos hijos eran rojos

[**1** pt.] Todas las hojas del árbol resultante tienen la misma profundidad.

b) No (en general). Basta con mostrar un contraejemplo en que cambie la forma o los colores.
P.ej., un árbol de raíz 3 (negra), y con hijo izquierdo 2 (rojo): si insertamos el nodo 1 (rojo), como hijo izquierdo de 2, y hacemos una rotación simple a la derecha (para evitar un nodo rojo con padre también rojo), queda el árbol con raíz 2 (negra), e hijos derecho 1 e izquierdo 3 (ambos rojos). Si ahora eliminamos el nodo 1, que acabamos de insertar, queda el árbol con raíz 2 (negra), y con hijo derecho 3 (rojo), distinto del árbol inicial.

3 Hashing

En esta sección se encuentran los siguientes contenidos:

- Tablas de Hash
- Funciones de Hash

2020-2-I2-P4–Hash con encadenamiento (1/1)

Pregunta 4

Considera la siguiente implementación de una tabla de hash con encadenamiento: la tabla es un arreglo $\boldsymbol{T}$ en que cada casillero $\boldsymbol{T[i]}$ tiene tres campos: dos punteros, y un espacio para guardar lo que queremos guardar en la tabla. En todo momento, todos los casilleros vacíos de la tabla se mantienen en una única lista $\boldsymbol{L}$ doblemente ligada de casilleros disponibles, usando los campos punteros de los casilleros para indicar el anterior y el siguiente. Para insertar un elemento con clave $\boldsymbol{k}$ en la tabla, hacemos lo siguiente: miramos el casillero $\boldsymbol{T[h(k)]}$; si el casillero está disponible, entonces lo extraemos en $\boldsymbol{O(1)}$ de la lista $\boldsymbol{L}$ y lo usamos para guardar el elemento; en caso contrario, sacamos el primer casillero disponible de la lista $\boldsymbol{L}$, guardamos allí el elemento, y colocamos este casillero a la cola de una lista simplemente ligada que parte en $\boldsymbol{T[h(k)]}$. Compara este esquema con el esquema de encadenamiento visto en clases. En particular, compara la complejidad esperada de las operaciones de inserción, búsqueda y eliminación.

Solución Pregunta 4)

Por cada una de las operaciones (inserción, búsqueda y eliminación):

- 1 punto por el cálculo correcto de la complejidad (dependiendo de si es caso promedio o peor caso según mencionado por estudiante): O(n) en el peor caso y O(1) en el caso promedio. 0.5 puntos si no se justifica o se justifica parcialmente.

  Elementos que se pueden usar para el cálculo del caso promedio:

  - Al asumir que h(k) es uniforme, se puede hacer un argumento a que en el caso promedio las listas ligadas con elementos de cada uno de los nodos tendrán un número acotado de elementos, con esto se puede argumentar que es O(1).
  - Alternativamente se puede hacer el símil con el esquema encadenamiento visto en clases, y argumentar que la diferencia de operaciones en cada uno de los pasos tiene la misma complejidad asintótica. Con esto, se puede acudir a los resultados de complejidad vistos en clase.

  En el peor caso, también se puede acudir al símil con el esquema de encadenamiento visto en clases, o también explicar cuál es el peor caso, donde todos los elementos resultan tener el mismo h(k), por lo que para insertar un elemento sería O(n), para acceder o eliminar a un elemento el peor caso es cuando el elemento está al final de la lista ligada, en el mismo caso anterior, por lo que también sería O(n) porque habría que recorrer esa lista ligada en su completitud.

- 1 punto por la comparación con encadenamiento. Independiente del caso, el estudiante debe llegar a que la complejidad es igual a la del esquema de encadenamiento visto en clases. En la comparación, el estudiante debe aludir al símil en las operaciones realizadas por cada uno de los esquemas, llegando a la conclusión que tienen la misma complejidad.

Puntaje total de la pregunta: 6 puntos.

2019-2-I2-P1-2–Tabla de Hash (1/1)

2) La pizzería intergaláctica M87 sirve pizzas de todos los incontables sabores existentes en todos los multiversos, y necesita ayuda para hacer más eficiente la atención de los millones de pedidos que recibe por segundo. Los pedidos funcionan de la siguiente manera:

- Una persona solicita una pizza del sabor que haya escogido y da su nombre. Su pedido se agrega al sistema.
- Cuando la pizza está lista, se llama por el altavoz a la persona que haya pedido ese sabor hace más tiempo, y, una vez entregada la pizza, se borra ese pedido del sistema.

Explica cómo usar tablas de hash para llevar a cabo este proceso eficientemente. ¿Qué esquema de resolución de colisiones debería usarse y por qué? ¿Qué es lo que se guarda en la tabla?

Solución:

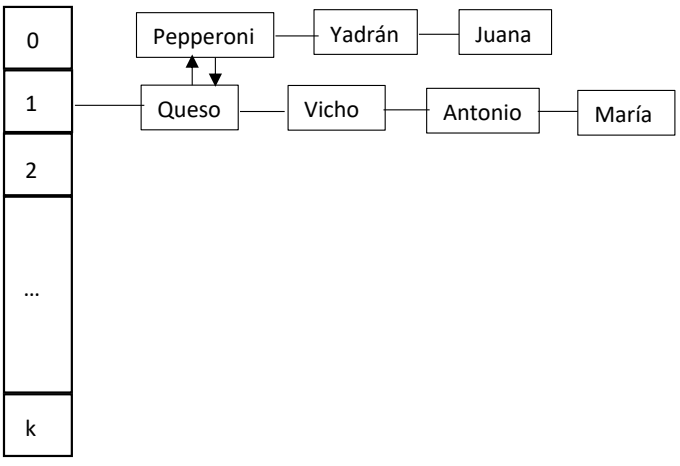
Para eso usamos una **tabla de hash** que nos permita guardar múltiples *values* para un mismo *key*. En este caso *key* corresponde al tipo de pizza y un *value* corresponde al nombre de la persona que lo pidió. **[1.5pts]**

Los *values* de un mismo *key* se deben guardar en una Cola (FIFO), de manera que agregar un nuevo *value* o extraer el siguiente sea $O(1)$ y se atienda en orden de llegada. **[1pt].** (Si deciden guardarlo en un Heap que ordena por orden de llegada, las operaciones son $O(log\ n)$, por lo que sólo obtienen **[0.5pts]**)

La eliminación del pedido sale automática con el heap o la cola ya que obtener el siguiente elemento lo extrae de la estructura. **[1pt]**

Pero como el dominio de las *key* es infinito, debemos ir despejando las celdas de la tabla cuando una *key* se queda sin *values*, ya que no tenemos memoria infinita. **[0.5pts]** Para esto es necesario usar **encadenamiento** ya que permite eliminar *key*s de la tabla sin perjudicar el rendimiento de esta. De esta manera, en caso de que varios sabores de pizza sean hasheados a la misma celda de la tabla (colisión de sabores), las colas FIFO de los nombres de las personas que pidieron esas pizzas deberán ser encadenadas en una lista doblemente ligada que las contenga **[2pts].**

Ej: tabla de hash con k celdas, con colisión de sabores Pepperoni y Queso en k = 1.



2019-1-C3–Tabla de Hash, Heap (1/2)

Estructuras de Datos y Algoritmos – iic2133

Control 3

17 de abril, 2019

Nombre: _____

1) Una tienda quiere premiar a sus k clientes más rentables. Para ello cuenta con la lista de las n compras de los últimos años, en que cada compra es una tupla de la forma $(ID\ Cliente, Monto)$. La rentabilidad de un cliente es simplemente la suma de los montos de todas sus compras.

Explica cómo usar *tablas de hash* y *heaps* para resolver este problema en tiempo esperado —o promedio— $O(n + n \log k)$.

Solución: El problema se separa en dos partes. Se asignará puntaje por separado para cada una.

- **Calcular la rentabilidad de cada cliente (2pts)**

Queremos obtener el set de tuplas $(ID\ Cliente, Rentabilidad)$, donde la rentabilidad para el $ID\ Cliente = i$ es la suma de todos los montos de las tuplas de la forma $(i, Monto)$.

Para esto creamos una tabla de hash T donde se almacenan tuplas $(ID\ Cliente, Monto)$. Cada vez que se inserte un $ID\ Cliente$:

- Si ya está en la tabla, se suma el monto recién insertado al monto guardado.
- Si no, se guarda en la tabla con el monto recién insertado como monto inicial.

Al hacer esto con todas las n tuplas hemos efectivamente encontrado la rentabilidad para cada cliente. **[1pt]**

La inserción en esta tabla tiene tiempo esperado $O(1)$ como se ha visto en clases. Como se realizan n inserciones, esta parte tiene tiempo esperado $O(n)$. **[1pt]**

- **Buscar los *k* clientes más rentables (4pts)**

Posible solución:

Sea m el total de clientes distintos. Creamos un *min-heap* de tamaño k que contendrá los k clientes más rentables que hemos visto hasta el momento. De este modo, la raíz del heap es el cliente **menos rentable** de los k **más rentables** encontrados hasta el momento. **[1pt]**

Iteramos sobre las tuplas en T, insertando en el heap hasta llenarlo, usando la rentabilidad como prioridad. Luego de haber llenado el heap, seguimos iterando sobre T. Para cada elemento que

2019-1-C3–Tabla de Hash, Heap (2/2)

veamos que sea mayor a la raíz, lo intercambiamos con esta. Luego hacemos *sift-down* para restaurar la propiedad del heap. Esto mantiene la propiedad descrita en el párrafo anterior. **[2pts]** (También es posible extraer la cabeza del heap e insertar el nuevo elemento normalmente)

Cada inserción en el heap toma $O(log\ k)$. En el peor caso insertamos los m elementos en el heap, por lo que esta parte es $O(m\ log\ k)$. Pero como en el peor caso $m = n$, esta parte es $O(n\ log\ k)$. **[1pt]**

Para cada sección, no se asignará puntaje si no explican correctamente el método propuesto o este no resuelve correctamente el problema. Para la segunda sección, se asignará como máximo 1pt si no alcanzan la complejidad solicitada.

2) La pizzería intergaláctica M87 sirve pizzas de todos los incontables sabores en existencia en todos los multiversos, y necesita ayuda para hacer más eficientes la atención de los pedidos, ya que recibe millones por segundo. Los pedidos funcionan de la siguiente manera:

- Una persona solicita una pizza del sabor que haya escogido y da su nombre. Su pedido se agrega al sistema.
- Cuando una pizza está lista, se llama por el altavoz a la persona que haya pedido ese sabor hace más tiempo, y, una vez entregada la pizza, se borra ese pedido del sistema.

Explica cómo usar *tablas de hash* para llevar a cabo este proceso eficientemente. ¿Qué esquema de resolución de colisiones debería usarse y por qué? ¿Qué es lo que se guarda en la tabla?

Solución: Necesitamos resolver las siguientes operaciones:

- Registrar pedido (*pizza*, *nombre*). Para una misma *pizza* los nombres deben guardarse por orden de llegada.
- Buscar el siguiente *nombre* para una *pizza* dada.
- Eliminar el pedido del sistema.

Para eso usamos una **tabla de hash** que nos permita guardar múltiples *values* para un mismo *key*. En este caso *key* corresponde al tipo de pizza y un *value* corresponde al nombre de la persona que lo pidió. **[1.5pts]**

Los *values* de un mismo *key* se deben guardar en una Cola (FIFO), de manera que agregar un nuevo *value* o extraer el siguiente sea $O(1)$ y se atienda en orden de llegada. **[1pt].** (Si deciden guardarlo en un Heap que ordena por orden de llegada, las operaciones son $O(log\ n)$, por lo que sólo obtienen **[0.5pts]**)

La eliminación del pedido sale automática con el heap o la cola ya que obtener el siguiente elemento lo extrae de la estructura. **[1pt]**

2019-1-Ex-P3–Identificación de Hash, Dijkstra (1/3)

3. Se tiene un *stream* de largo indefinido, que termina con el dato *d**. Cada dato *d* = (*u*, *v*) está formado por los nombres de dos ciudades, *u* y *v*, y representa la existencia de un vuelo directo de la ciudad *u* a la ciudad *v*. La idea es interpretar cada dato como una arista direccional que se agrega a un grafo inicialmente vacío.

2019-1-Ex-P3–Identificación de Hash, Dijkstra (2/3)

a) [**2 pts.**] Explica cómo guardas el grafo en memoria; es decir, describe las estructuras de datos necesarias (tales como tablas de hash, listas de adyacencia, matriz de adyacencia, etc.), en lo posible ayudado por un dibujo. Ten presente que los datos originales son nombres de ciudades y no números. Define el **struct nodo** que usarías en C y qué cosas guarda.

Solución:

En primer lugar, el largo del stream es indefinido, y además no conocemos la lista de todas las ciudades. Eso significa que si o si es necesario el uso de un diccionario (tabla de hash) para guardar los datos, ya que no podemos definir de antemano una forma de guardarlos. **[0.75pt]** Por otro lado, es necesario usar el modelo de listas de adyacencia usando listas ligadas, ya que podemos ir agregándole vecinos a un nodo en O(1) sin costo adicional de memoria (a diferencia de la matriz de adyacencia que ocupa mucha más memoria de la necesaria y además no conocemos el tamaño de que debería tener o un arreglo de largo fijo para los vecinos) **[0.75pt]**

En esta tabla se guardan los pares <key, value> donde key son los nombres de las ciudades y value es un contenedor con toda la información de ese nodo.

Este contenedor tiene la siguiente forma:

```
struct nodo
{
   id; // El nombre de la ciudad
   vecinos; // la lista de adyacencia (nodos vecinos) de ese nodo
}
```

La lista de adyacencia tiene punteros directamente a los nodos vecinos

[0.5pt]

[Los tipos de C no son relevantes mientras quede claro qué se está guardando]

Considera ahora que cada dato en realidad es un trío de la forma *d = (u, v, w)* en que *w* es la duración del vuelo, en minutos (es decir, un número entero no negativo), entre *u* y *v*.

b) [**1 pt**] ¿Cómo cambia tu modelo del grafo en a)?

Solución:

2019-1-Ex-P3–Identificación de Hash, Dijkstra (3/3)

La única diferencia es que ahora hay que guardar el peso *w* de las aristas del grafo en algún lado. **[0.5pt]** Ese lugar puede ser en la lista de vecinos del struct nodo, donde ahora en lugar de guardar punteros a los vecinos se guarden tuplas `(nodo*, peso)`. **[0.5pt]**

Finalmente, queremos aprovechar el hecho de que tenemos los datos representados como un grafo direccional con costos para buscar las rutas más cortas (rápidas) desde la ciudad *a* "a" todas las otras ciudades.

c) **[1pt]** ¿Qué algoritmo usarías para lograr esto? ¿Por qué?

Solución:

Dijkstra. **[0.5pt]** Porque el algoritmo de Dijkstra partiendo desde un nodo *"a"* genera un **árbol de rutas más cortas** desde ese nodo a todos los otros nodos, que es precisamente lo que queremos. **[0.5pt]**

d) **[1pt]** ¿Qué estructuras de datos adicionales necesitas para ejecutar eficientemente el algoritmo de c)? ¿Por qué?

Solución:

Dijkstra es un algoritmo codicioso que **en cada paso explora el siguiente nodo al que es más barato llegar desde "a" dado los nodos que se han explorado**. **[0.5pt]** Para esto necesita una **cola de prioridades** (heap) **[0.5pt]**

e) **[1pt]** ¿Es necesario modificar tu modelación en b) para usar este algoritmo? ¿Por qué?

Solución:

Sí. Cuando el algoritmo de Dijkstra explora un nuevo nodo lo hace junto con una referencia de "desde donde" se exploró ese nodo (también conocido como "el padre del nodo"), creando así el árbol. Para esto es necesario agregar un puntero al nodo padre en el struct. **[1pt]** *El algoritmo también debe poder identificar si un nodo ha sido o no explorado, aunque esto es posible hacerlo revisando si el nodo padre es nulo. Si se menciona un atributo de este estilo sin mencionar el nodo padre solo se da* **[0.25pt]**

2018-2-I2-P2–Función de Hash, Tabla de Hash (1/2)

2a) Hashing

Queremos multiplicar dos números X y Y, y para verificar si el resultado $Z = X \times Y$ es correcto, aplicamos una función de hash h a los tres números y vemos si

$$h(\, h(X) \times h(Y)\,) = h(Z).$$

Si los números difieren, entonces cometimos un error, pero si son iguales, entonces el resultado es (muy probablemente) correcto. h se define como calcular repetidamente la suma de los dígitos, hasta que quede un solo dígito, en cuyo caso 9 cuenta como 0.

P.ej., si $X = 123456$ y $Y = 98765432$, entonces $Z = 12193185172992$, y $h(X) = h(21) = 3$, $h(Y) = h(44) = 8$, $h(Z) = h(60) = 6$, y efectivamente, $h(3 \times 8) = h(24) = 6 = h(Z)$. Notemos que el dígito 9 no influye en el resultado calculado de h; p.ej., $h(49) = h(13) = 4$.

a) Da una expresión matemática para $h(X)$ [**1.5 pts.**]; y **b**) muestra que este método de verificación del resultado de la multiplicación es correcto [**1.5 pts.**].

Respuesta:

a) **X mod 9** (También se aceptan respuestas en donde el alumno calcula la suma de los dígitos con alguna expresión y después puso módulo 9, o expresiones recursivas, funciones compuesta y otras donde se mencione explícitamente que h(9) = 0 o se maneje bien ese caso)

b)

Sabemos que:

$$X = 9 \times Q_1 + R_1$$
$$Y = 9 \times Q_2 + R_2$$
$$X \times Y = Z$$

Y queremos demostrar que:

$$h(h(X) \times h(Y)) = h(Z)$$
$$h(h(9 \times Q_1 + R_1) \times h(9 \times Q_2 + R_2) = h(X \times Y)$$
$$h(h(9 \times Q_1 + R_1) \times h(9 \times Q_2 + R_2) = h((9 \times Q_1 + R_1) \times (9 \times Q_2 + R_2))$$

En la expresión de la izquierda podemos eliminar todos los términos múltiplos de 9 ya que estamos trabajando en mod 9, obteniendo

$$h(R_1 \times R_2) = h((9 \times Q_1 + R_1) \times (9 \times Q_2 + R_2))$$
$$h(R_1 \times R_2) = h(81 \times Q_1 \times Q_2 + 9 \times Q_1 \times R_2 + 9 \times Q_2 \times R_1 + R_1 \times R_2)$$

Nuevamente, eliminamos los términos múltiplos de 9 de la expresión de la derecha ya que estamos trabajando en mod 9, obteniendo

$$h(R_1 \times R_2) = h(9 \times (9 \times Q_1 \times Q_2 + Q_1 \times R_2 + Q_2) + R_1 \times R_2)$$
$$h(R_1 \times R_2) = h(R_1 \times R_2)$$

Notas: Se aceptaran demostraciones menos formales, incluso con palabras, siempre que mencionen que los cocientes se eliminan ya que se está trabajando en mod 9 y quedan los restos. No se aceptarán respuestas que solo mencionan la propiedad sin demostrarla. Si el alumno no uso **X mod 9** en la parte a, se deberá evaluar caso a caso.

2b) Tablas de *hash*

2018-2-I2-P2–Función de Hash, Tabla de Hash (2/2)

Para cada uno de los siguientes problemas, responde si es posible resolver el problema eficientemente mediante *hashing*. En caso afirmativo, explica claramente cómo; de lo contrario, sugiere otra forma de resolverlo según lo estudiado en el curso.

a) Considera un sistema de respaldo en que toda la información digital de una empresa tiene que ser respaldada, es decir, copiada, cada cierto tiempo. Una propiedad de estos sistemas es que solo una pequeña fracción de toda la información cambia entre un respaldo y el siguiente. Por lo tanto, en cada respaldo, solo es necesario copiar la información que efectivamente ha cambiado. El desafío es, por supuesto, encontrar lo más que se pueda de la información que no ha cambiado. [**1.5 pts.**]

Si es posible resolver mediante hashing. Un ejemplo de implementación eficiente mediante hashing es el uso de una función de hash que inicialmente se haya usado para respaldar toda la información. Estos elementos habrían llegado a algún espacio de la tabla. Al momento de querer respaldar nuevamente la información que cambió, uno puede tomar cada archivo de la información digital y utilizar el hash para identificar si este se modificó en la tabla (por ejemplo, hashear el nombre del archivo con su path y la fecha de modificación, o también hashear el archivo completo) y en caso de que coincidan, no se respalda. En caso de que sean diferentes, se puede generar el nuevo hash a partir de un hash incremental y los pocos cambios generados para luego insertar nuevamente en la tabla (liberando la anterior o complementando con el resto de la información nueva a respaldar).

b) Dada una lista L de números, queremos encontrar el elemento de L que sea el más cercano a un número dado x. [**0.5 pts.**]

No es resolvible por hashing eficientemente. Dado que es una lista sin un orden específico, se debe considerar alguna alternativa que entrega un orden para luego hacer por ejemplo una búsqueda binaria sobre un arreglo o una búsqueda sobre un árbol binario balanceado.

c) Queremos encontrar un string S de largo m en un texto T de largo n. [**1 pt.**]

Se vio en clases. Se puede resolver eficientemente mediante hashing. Usando una función de hash incremental, se toman los primeros m caracteres del texto de largo n para calcular el hash. De esta manera se compara con el hash del string S a buscar. En caso de que no sean iguales los hash, debes eliminar el primer elemento del string y agregar el siguiente del texto (O(1)) para luego volver a realizar el proceso. En caso de que existan colisiones, es mejor revisar caracter a caracter en caso de que sean iguales los hash.

3. Rotaciones + árboles de búsqueda balanceados

a) [*Teorema fundamental de las rotaciones*]. Muestra que cualquier árbol binario de búsqueda (no necesariamente balanceado) puede ser transformado en cualquier otro árbol binario de búsqueda con las mismas claves mediante una secuencia de rotaciones simples.

b) Determina un orden en que hay que insertar las claves 1, 3, 5, 8, 13, 18, 19 y 24 en un árbol 2-3 inicialmente vacío para que el resultado sea un árbol de altura 1, es decir, una raíz y sus hijos.

c) Considera un árbol rojo-negro formado mediante la inserción de n nodos usando el procedimiento visto en clase. Justifica que si $n > 1$, el árbol tiene al menos un nodo rojo.

d) Muestra cómo construir un árbol rojo-negro que demuestre que, en el peor caso, casi todas las rutas desde la raíz a una hoja tienen largo $2 \log N$, en que N es el número de nodos del árbol.

Respuesta:

2018-1-I2-P2–Tabla de Hash (1/2)

2. Tablas de *hash*

Explica cómo manejar una tabla de hash si los elementos se almacenan dentro de la tabla (recuerda que puede haber colisiones), y además mantenemos una lista ligada de todos los casilleros vacíos. Su-ponemos que cada casillero de la tabla puede almacenar un *flag* (un bit 0 o 1) y, ya sea, un elemento más un puntero, o bien dos punteros. El objetivo es que todas las operaciones de diccionario (inser-ción, búsqueda y eliminación), así como las operaciones sobre la lista ligada, puedan ser ejecutadas en tiempo esperado O(1). ¿Es necesario que la lista ligada sea doblemente ligada?

Respuesta

Usamos el *flag* para indicar si el casillero tiene un puntero y un elemento (vale 0, cuando forma parte de una lista de elementos insertados que tienen el mismo valor de hash, similar a hashing con encadena-miento), o dos punteros (vale 1, cuando forma parte de la lista **doblemente ligada** de casilleros vacíos). Los punteros son simplemente índices de casilleros en la tabla. Sea L la lista de casilleros vacíos, y sean *prev* y *next* los dos punteros de cada casillero vacío. [**0.5 pts.**]

2018-1-I2-P2–Tabla de Hash (2/2)

Sea x el dato que nos interesa y sea k su valor de hash.

Inserción. Si el casillero k está vacío (su *flag* vale 1, por lo que forma parte de L), entonces lo sacamos de L —en tiempo O(1), ya que L es doblemente ligada— y almacenamos ahí x, p.ej., en el campo del puntero *prev*. Además, ponemos el *flag* del casillero en 0 y el puntero *next* en *null*. El casillero k pasa así a ser el primer elemento (y por ahora el único) de una lista de elementos que tienen valor de hash k. [**1 pt.**]

Si, por el contrario, el casillero k está ocupado (su *flag* vale 0), entonces hay dos posibilidades:

- Es el primer elemento de una lista L_k de elementos insertados que tienen valor de hash k ; entonces saca-mos un casillero (p.ej., el primero) de L, almacenamos x en este casillero (en el campo *prev*), y agregamos el casillero a la lista L_k (p.ej., como segundo elemento). Todas estas operaciones toman tiempo O(1). [**0.5 pts.**]

- Es un elemento de una lista de elementos insertados que tienen valor de hash $k' \neq k$; entonces sacamos el casillero k de esta lista (por supuesto, lo reemplazamos por un casillero de L), y lo convertimos en el primer casillero de la lista de elementos que tienen valor de hash k. Nuevamente, todas estas operaciones toman tiempo O(1). [**0.5 pts.**]

Búsqueda. Si el casillero k no tiene a x, entonces seguimos la cadena de punteros *next* que empieza aquí. Al hacer esta búsqueda, conviene guardar el puntero al elemento anterior al que estamos mirando (para el caso de eliminación). Esta operación **no** toma tiempo O(1). Para la búsqueda, vale el mismo argumento visto en clase para hashing con encadenamiento. [**2 pt.**]

Eliminación. Supongamos que al hacer la búsqueda anterior, en la lista L_k que empieza en el casillero k, encontramos a x en el casillero q, que el casillero anterior a q en L_k es el casillero p, y que el casillero siguiente es r. Entonces sacamos el casillero q de L_k, lo ponemos (de vuelta) en L, y actualizamos L_k (haciendo que el casillero p apunte al casillero r). [**1.5 pts**]

2017-1-I3-P2–Función de Hash (1/2)

2. Hashing universal es una técnica para generar buenas funciones de hash. Dado un universo de claves U, se definen la funciones $g_{a,b}(k) = (ak + b) \bmod p$ y $h_{a,b}(k) = g_{a,b}(k) \bmod m$, donde p es un primo tal que $p > k$, para cada $k \in U$, y $a \in \{1, \ldots, p-1\}$ y $b \in \{0, \ldots, p-1\}$.

   a) Una de las razones porque $h_{a,b}$ es "buena" es porque "evita colisiones antes de módulo m". Esto significa que si $k \neq k'$, entonces $g_{a,b}(k) \neq g_{a,b}(k')$. Demuestre este resultado.

   Respuesta (2 puntos): Esto se puede demostrar por contradicción, asumiremos lo contrario, es decir $k \neq k'$ y $g_{a,b}(k) = g_{a,b}(k')$. Desarrollamos la igualdad:

   $$\begin{aligned} g_{a,b}(k) &= g_{a,b}(k') \\ g_{a,b}(k) - g_{a,b}(k') &= 0 \\ (ak + b) \bmod p - (ak' + b) \bmod p &= 0 \end{aligned}$$

   Utilizando las propiedades del módulo y que la resta siempre estará en el rango $\{0, p-1\}$ llegamos a que:

   $$\begin{aligned} ((ak + b) - (ak' + b)) \bmod p &= 0 \\ (a(k - k')) \bmod p &= 0 \end{aligned}$$

   Luego, para que se cumpla la igualdad se debe cumplir alguna de las siguientes condiciones:

   - $a \bmod p = 0$, lo cual no es cierto debido a que $a \in \{1, \ldots, p-1\}$ y $b \in \{0, \ldots, p-1\}$.
   - $(k - k') \bmod p = 0$, lo cual no es cierto ya que $p > k$ y por lo tanto la diferencia nunca estará fuera del rango $\{0, p-1\}$. Además como $k \neq k'$ la resta nunca será 0.
   - $a(k - k') = cp$ con c entero. Tampoco es cierto, ya que si lo fuera a o $(k - k')$ serían divisores de p, lo cual sumando el hecho de que $a < p$, y $(k - k') < p$ contradice el hecho de que p es primo.

   Por lo tanto llegamos a una contradicción, ya que es imposible que se cumpla la igualdad.

   Asignación de puntaje:

   - **1 punto** por demostrar que la expresión no es igual a 0 antes del módulo.
   - **1 punto** por demostrar que la expresión no es igual a un múltiplo de p.

   b) El teorema de *a*) se puede demostrar sin obligar a que p sea primo, pero imponiendo otras restricciones sobre $g_{a,b}(k)$. Diga cuáles y demuestre su respuesta.

   Respuesta (2 puntos): Se mantienen la condiciones anteriores pero agregando la condición de que a sea primo relativo a p, es decir que a no tenga divisores en común con p. La demostración es equivalente a la de *a)*, solo que ahora $a(k-k') = cp$ no puede ser cierto ya que significaría que un factor primo de a está en p.

   Asignación de puntaje:

   - **2 puntos** por respuesta correcta junto a su demostración.
   - **1.5 puntos** por respuesta correcta con errores en la justificación.
   - **1 punto** por solo decir que a no sea un divisor de p (Esto no es suficiente, podría pasar que a sea un factor de cp)
   - **0.5 puntos** por dar condiciones demasiado restrictivas, por ejemplo restringir que $|a(k - k')|$ sea menor a p.

2017-1-I3-P2–Función de Hash (2/2)

c) Muestre que si relajamos la restricción "$p > k$, para cada $k \in U$" es posible que $g_{a,b}(k) = g_{a,b}(k')$ incluso cuando $k \neq k'$.

Respuesta (2 puntos):
Ahora se puede dar que $k - k' = cp$ y por lo tanto es posible que $(a(k - k')) \bmod p = 0$. También se puede demostrar con un contraejemplo, un caso sería $a = 1, b = 0, p = 3, k = 1, k' = 4$.

$$g_{a,b}(k) = (1 \cdot 1 + 0) \bmod 3 = 1$$
$$g_{a,b}(k') = (1 \cdot 4 + 0) \bmod 3 = 1$$

La asignación de puntos es binaria, se dan 0 o 2 puntos.

2017-1-Ex-P1-e—j—l–Tabla de Hash, Re-hashing (1/1)



PONTIFICIA UNIVERSIDAD CATÓLICA DE CHILE
ESCUELA DE INGENIERÍA
DEPARTAMENTO DE CIENCIA DE LA COMPUTACIÓN

IIC 2133 — Estructuras de Datos y Algoritmos
Examen
Primer Semestre, 2017

Duración: 3 hrs.

1. Para cada una de las siguientes afirmaciones, diga si es *verdadera* o *falsa*, **siempre justificando** su respuesta.

   a) Quick Sort es un algoritmo de ordenación estable. **Respuesta:** Falso. Basta con dar un contraejemplo.

   b) Sea e la segunda arista más barata de un grafo dirigido acíclico G con más de dos nodos y aristas con costos diferentes. Entonces e pertenece al árbol de cobertura de costo mínimo para G. **Respuesta:** Verdadero. Si usamos el algoritmo de Kruskal, la segunda arista más barata es siempre agregada al MST puesto que no puede formar un ciclo en el bosque construido hasta el momento.

   c) Radix Sort es un algoritmo de ordenación que puede ordenar n números enteros y cuyo tiempo de ejecución está siempre en $\Theta(n)$. **Respuesta:** El tiempo de ejecución de Radix Sort es $d(n + k)$, cuando los datos están en $[0, k]$. Basta entonces con que k sea suficientemente grande (por ejemplo, exponencial en n), para que el algoritmo no sea $\Theta(n)$

   d) Sea A un árbol rojo-negro en donde cada rama tiene n nodos negros y n nodos rojos. Si al insertar una clave nueva en A se obtiene el árbol A', entonces cada rama de A' tiene $n + 1$ nodos negros. **Respuesta:** Verdadero. Un árbol rojo negro como A corresponde a un árbol 2-4 "completamente saturado"; es decir, uno que contiene nodos con 3 claves. Al insertar una clave nueva, el árbol 2-4 aumentará su altura en 1. Esto significa que su equivalente rojo-negro debe tener un nodo negro más por cada rama.

   e) Si se tienen dos tablas de hash, una con direccionamiento abierto y otra cerrado, las dos del mismo tamaño m y el mismo factor de carga α, ambas ocupan la misma cantidad de memoria.

   f) Sea A un árbol AVL y ℓ_1 y ℓ_2 los largos de dos ramas de A. Entonces $|\ell_1 - \ell_2| \leq 1$. **Respuesta:** Falso. Basta dar un contraejemplo

   g) La operación de inserción en un árbol AVL con n datos realiza a lo más una operación *restructure* pero toma tiempo $O(\log n)$. **Respuesta:** Verdadero, puesto que la inserción debe revisar que el balance esté correcto a lo largo de la rama donde se insertó el dato. Y esa rama tiene tamaño $O(\log n)$.

   h) Si A es un ABB y n es un nodo de A, el *sucesor* de n no tiene un hijo izquierdo. **Respuesta:** Falso. La propiedad no se cumple en general cuando n no tiene un hijo derecho.

   i) Es posible modificar la implementación de la estructura de datos para conjuntos disjuntos vista en clases, de manera que permita des-unir dos conjuntos en $O(1)$.

   j) Como diccionario, una tabla de hash es más conveniente que un árbol rojo negro en cualquier aplicación. (Sin considerar la dificultad de implementación).

   k) Sea p una secuencia de dos o más números diferentes y sea q una permutación de p distinta de p. Adicionalmente, sean A_p y A_q los árboles binarios de búsqueda que resultan de, respectivamente, insertar en orden los elementos de p y q en árboles binarios de búsqueda vacíos. Entonces A_p y A_q son distintos. **Respuesta:** Falso.

   l) Re-hashing, el procedimiento que construye una nueva tabla de hash a partir de otra existente, toma tiempo $O(n)$ en una tabla con colisiones resueltas por encadenamiento de tamaño m que contiene n datos. **Respuesta:** Falso. El tiempo depende del tamaño de la tabla y del número de datos; específicamente es $O(m + n)$.

2016-2-I2-P3–Tabla de Hash (1/1)

3. Tienes el texto completo de la novela *A Tale of Two Cities*, de Charles Dickens.

> *It was the best of times, it was the worst of times, it was the age of wisdom, it was the age of foolishness ...*

Un algoritmo de reconocimiento de texto necesita obtener la lista de palabras distintas en la novela. Escribe un algoritmo que haga esta operación en tiempo O(n) promedio, en que n es el número total de palabras del texto; y justifica esta complejidad.

Respuesta:

Una solución es hacer una tabla de hash con encadenamiento sobre los posibles strings y en base a eso detectar las palabras iguales:

$TablaHash \leftarrow$ Arreglo tamaño m de listas vacías.
$Resp \leftarrow$ Lista vacía de respuesta.
for $pal \in Palabras$ **do**
  $key \leftarrow hash(pal)\%m$
  if $TablaHash[key] == \emptyset \vee pal \notin TablaHash[key]$ **then**
    $TablaHash[key].add(pal)$
    $Resp.add(pal)$
  end if
end for
return $Resp$

Donde la función de hash es tal que no produce colisiones: se puede transformar el String a un entero fácilmente usando potencias de 128 en ASCII.

Cabe destacar que la tabla si tendrá colisiones, ya que a la función de hash luego se le aplica el módulo del tamaño del arreglo “m”.

Finalmente como nuestra función de hash distribuye bien y suponiendo que satisface HUS, el riesgo de colisiones distribuye uniformemente en el caso promedio y por lo tanto la búsqueda en promedio será O(1) si tomamos un “m” lo suficientemente grande. (Se pueden hacer otras funciones o suponer que existe una, siempre y cuando se diga que satisface HUS).

El puntaje se entrega de la siguiente forma:

- 1 pto si es que se hizo un algoritmo correcto pero no de la complejidad pedida
- 3 ptos si es que se hizo una tabla de hash pero sin colisiones *
- 6 ptos por una respuesta correcta con justificación de complejidad, también existen otras soluciones como usar direccionamiento abierto.

* Es imposible hacer una tabla de hash sin colisiones ya que las palabras posibles son muchas (en teoría infinitas), notamos que además la cantidad de strings posibles crece exponencialmente según el largo, por lo que por ejemplo si hiciéramos una tabla sin colisiones para strings de largo 100 se necesitaría un arreglo de tamaño $128^{100} = 5.26 * 10^{210}$ lo cual es gigantesco y no es posible para ningún computador (Un computador normal soporta del orden de 10^{9} de memoria).

2016-1-I2-P3–Tabla de Hash (1/1)

3. [**5**] Considera una **tabla de hash** en que las colisiones se resuelven mediante encadenamiento: Indica tres estructuras de datos fundamentalmente diferentes para implementar este "encadenamiento" (una de ellas puede ser listas doblemente ligadas). Explica de manera breve y precisa las ventajas y desventajas principales de cada una.

En lugar de una lista ligada, en que la inserción se hace siempre al comienzo de la lista, pero hay que buscar secuencialmente, se podría usar un árbol binario de búsqueda, e incluso una nueva tabla de hash. El árbol permite insertar, buscar y eliminar en tiempo proporcional al logaritmo del número de ítemes que hay en el árbol; esto es en promedio, o bien en el peor caso si el árbol se mantiene balanceado. La tabla permite buscar, insertar y eliminar en tiempo constante, en promedio.

2015-2-I1-P2–Implementación de Tablas de Hash, Listas Ligadas (1/1)

2. Un supermercado quiere desarrollar una aplicación que le permita manejar en memoria principal información acerca de la venta de los productos a los clientes con tarjeta durante un fin de semana particular. El supermercado quiere llevar un registro de las cantidades vendidas, pero también quiere saber rápidamente, p.ej., en cuáles productos ha comprado un cliente y qué clientes han comprado un cierto producto. Inicialmente, se pensó en emplear una tabla, en que las filas representan a los productos y las columnas a los clientes. Pero luego quedó claro que desde el punto de vista de uso de memoria la tabla no es muy eficiente: si el supermercado tiene 20,000 clientes con tarjeta y ofrece 20,000 productos durante el fin de semana, la tabla tendría que tener unas $20{,}000 \times 20{,}000 = 400{,}000{,}000$ casillas; pero si sólo la mitad de los clientes van a comprar el fin de semana y cada uno compra en promedio 40 productos, entonces sólo se estaría usando el uno por mil de las casillas.

a) Describe una estructura de datos basada en listas ligadas para resolver este problema más eficientemente, en que sólo sea necesario ocupar una cantidad de memoria del orden de los cientos de miles de casillas ($10{,}000 \times 40 = 400{,}000$), y no de millones; preferentemente, dibuja la estructura de datos.

Por cada cliente que compra un producto, usamos un registro (objeto) de 5 campos: una identificación del cliente, una identificación del producto, la cantidad vendida (de ese producto a ese cliente), un puntero (imaginémoslo vertical hacia abajo) al objeto que representa otro producto comprado por el mismo cliente, y un puntero (imaginémoslo horizontal hacia la derecha) al objeto que representa a otro cliente que compra el mismo producto.

b) Suponiendo que ni los clientes ni los productos cambian durante el fin de semana (es decir, ni se agregan ni se eliminan), ¿cómo debiera funcionar tu estructura de datos para que el usuario de la aplicación pueda referirse a los productos por sus códigos (p.ej., los que se usan al pasarlos por la caja) y a los clientes por sus RUT's?

Usamos dos arreglos de tamaño 20,000, uno para los clientes y otro para los productos. Cada casilla de estos arreglos tiene dos punteros: uno a la información del cliente (rut, nombre) o del producto (código, nombre), respectivamente, y otro a uno de los objetos descritos en **a**). Usamos sendas funciones de hash para convertir rut's o códigos de productos en índices en estos arreglos, y usamos estos índices como los identificadores de los clientes y de los productos en los objetos de **a**).

2015-2-I1-P3–Hashing con direccionamiento abierto (1/1)

3. En el caso de hashing con direccionamiento abierto, considera que tienes una tabla de tamaño $T = 10$ y las claves A_5, A_2, A_3, B_5, A_9, B_2, B_9, C_2, en que K_i significa que al aplicar la función de hash h a la clave, obtenemos la posición i. Muestra qué ocurre al insertar las claves anteriores, en el orden en que aparecen, en los siguientes casos:

a) Usamos revisión lineal, de modo que la posición intentada es $(h(K) + i)$ modulo T.

b) Usamos revisión cuadrática, de modo que la posición intentada es $h(K)$, $h(K)+1$, $h(K)–1$, $h(K)+4$, $h(K)–4$, $h(K)+9$, ..., $h(K)+(T–1)^2/4$, $h(K)–(T–1)^2/4$, todos divididos módulo T.

	a)	**b)**
0	B_9	B_9
1		B_2
2	A_2	A_2
3	A_3	A_3
4	B_2	
5	A_5	A_5
6	B_5	B_5
7	C_2	
8		C_2
9	A_9	A_9

2015-1-I1-P1-a–Implementación de Tablas de Hash (1/1)

Estructuras de Datos y Algoritmos – IIC2133

I1

16 abril 2015

(***Algoritmo***: Secuencia ordenada y finita de pasos para llevar a cabo una tarea, en que cada paso es descrito con la precisión suficiente, p.ej., en un pseudo lenguaje de programación similar a C, para que un computador lo pueda ejecutar.)

1. *Estructuras de datos básicas.*

a) Describe una **estructura de datos** que funcione en memoria principal para majenar una pequeña biblioteca en nuestro departamento. La biblioteca tiene libros y tiene lectores. Los lectores son los estudiantes, profesores y funcionarios del departamento, todos con nombre y RUT. Los libros tienen título y tienen autor (el nombre de una persona). Tanto los lectores como los libros permanecen fijos durante el semestre (marzo a julio, agosto a diciembre). Al bibliotecario le interesa poder prestar libros a los lectores, recibir de vuelta libros que estaban prestados, y conocer la información habitual eficientemente: ¿Qué lectores hay registrados en la biblioteca? ¿Qué libros tiene la biblioteca? ¿Cuáles de ellos están disponibles y cuáles están prestados? ¿Cuál lector tiene un determinado libro prestado? ¿Cuáles libros están prestados a un determinado lector? Tu estructura puede incluir **arreglos**, **listas ligadas** en varias direcciones, y **tablas de hash**. **Justifica**.

Podemos manejar tanto los libros como los lectores en arreglos, en que cada casillero tiene varios campos; puede haber un tercer arreglo para los autores de los libros. Todos estos arreglos son en realidad tablas de hash; las claves son los RUT's de los lectores, los títulos de los libros, y los nombres de los autores. Los casilleros del arreglo de los autores son punteros a una lista ligada de punteros a cada uno de los libros (casilleros del arreglo de libros) del autor correspondiente. Para saber qué lectores y qué libros hay, simplemente recorremos los arreglos corresppndientes. Además, hay que representar los préstamos.

(Podríamos usar una tabla de doble entrada, es decir, una matriz, en que, p.ej., las columnas son los libros y las filas los lectores. Cada casilla de la matriz correspondería a un boolean, para indicar si el libro está prestado —1— o no —0— a un determinado lector. Si hay 500 lectores y 1,000 libros, la martiz tendría 500,000 casillas; pero a lo más 1,000 de estas casillas podrían tener un 1, cuando todos los libros están prestados.)

Siguiendo la sugerencia del enunciado, representamos los préstamos por listas ligadas a partir de los lectores. Cada elemento de una lista ligada es un puntero a un libro: el libro que está prestado al lector correspondiente. Como un lector puede tener varios libros prestados, ponemos todos esos elementos en una lista doblemente ligada. Además, para saber a qué lector está prestado un determinado libro, cada elemento de las listas ligadas es apuntado "de vuelta" por el libro correspondiente.

2014-1-I1-P2–Hashing con direccionamiento abierto (1/1)

2. *Hashing*.

a) En el caso de *hashing* con direccionamiento abierto, si la tabla empieza a llenarse, entonces la ejecución de las operaciones de búsqueda e inserción empieza a tomar demasiado tiempo. La solución es construir otra tabla que sea el doble de grande, definir una nueva función de *hash*, y revisar la tabla original completa, calculando el nuevo valor de *hash* para cada elemento e insertándolo en la nueva tabla.

Esta operación, llamada *rehashing*, es cara, pero en la práctica, considerando la frecuencia con que debe ejecutarse, no afecta demasiado al desempeño global de la tabla. ¿Por qué?

Supongamos que decidimos hacer *rehashing* cada vez que la tabla está 50% llena (número de claves $n = m/2$, en que m es el tamaño de la tabla); es decir, tienen que haber ocurrido por lo menos $n = m/2$ operaciones de inserción. ¿Cuánto cuesta hacer *rehashing*? Reservar una nueva tabla de tamaño $2m$ y definir una nueva función de *hash* para esta tabla es O(1); suponiendo que calcular el valor de *hash* de una clave es O(1), entonces recorrer la tabla original y calcular el nuevo valor de *hash* para cada clave almacenada es O(m). Es decir, ejecutamos una operación de costo O(m) después de haber hecho $m/2$ = O(m) operaciones de costo O(1) en promedio cada una. Por lo tanto, en términos de O( ), el costo de *rehashing* no suma al costo ya incurrido de las inserciones en la tabla original.

b) Queremos encontrar la primera ocurrencia de un string $P_1P_2…P_k$ en otro string mucho más largo $A_1A_2…A_n$ ($n > k$). ¿Cómo podemos resolver este problema usando *hashing*? Suponiendo que para un string s empleamos una función de *hash* como la siguiente, ¿qué tan eficiente, en términos de k y n, es esta solución?

```
hashValue = 0
for (i = 0; i < s.length(); i++)
   hashValue = 37*hashValue + s.ascii(i)
hashValue = hashValue % tableSize
```

La idea es calcular el valor de *hash* para el string $P_1P_2…P_k$ y luego calcular el valor de *hash* para cada substring de largo k del string $A_1A_2…A_n$. Si los valores de *hash* son distintos, entonces los strings no pueden ser iguales. Si los valores de *hash* son iguales, entonces comparamos los strings carácter por carácter (ya que hay una pequeña posibilidad de que los strings sean distintos).

En general, este método va a tomar un tiempo proporcional al tiempo que toma calcular el valor de *hash* para un string de k caracteres, digamos $f(k)$, multiplicado por el número de tales strings, $n – k$, y más lo que toma comparar carácter por carácter dos strings de k caracteres; es decir, $(n – k)f(k) + k$. En el caso de la función de *hash* dada, $f(k) = ck$; luego, el método toma tiempo proporcional a $ck(n – k) + k$; esencialmente, O(nk).

Sin embargo, observamos que el valor de *hash* del string $A_iA_{i+1}…A_{i+k–1}$, digamos h_i, puede obtenerse directamente a partir del valor de *hash* del string $A_{i–1}A_i…A_{i+k}$, digamos $h_{i–1}$, sin tener que calcularlo desde cero: $h_i = h_{i–1} – \text{ascii}(A_{i–1})*37^{k–1} + \text{ascii}(A_{i+k})$. Es decir, sólo el cálculo del valor de *hash* del primer substring $A_1A_2…A_k$ toma tiempo proporcional a k; los $n–k–1$ siguientes son todos constantes. Por lo tanto, el método toma tiempo proporcional a $k + (n–k–1) + k$; esencialmente, O($n+k$).

2013-2-I1-P1–Implementación de Tablas de Hash (1/1)

Estructuras de Datos y Algoritmos – IIC2133

I1

6 septiembre 2013

1. *Tablas ralas*. La Escuela de Ingeniería quiere desarrollar una aplicación que le permita manejar en memoria principal información acerca del desempeño de los estudiantes en los cursos que están tomando durante un semestre particular. La Escuela quiere llevar un registro de las notas parciales, pero también quiere saber rápidamente, p.ej., en cuáles cursos está inscrito un estudiante y qué estudiantes están inscritos en un cierto curso. Inicialmente, se pensó en emplear una tabla, en que las filas representan a los cursos y las columnas a los estudiantes. Pero luego quedó claro que desde el punto de vista de uso de memoria la tabla no es muy eficiente: si la Escuela tiene 4,000 estudiantes y ofrece 500 cursos al semestre, la tabla tendría que tener unas $4{,}000 \times 500 = 2{,}000{,}000$ casillas; pero si un estudiante toma en la práctica sólo 5 cursos al semestre, entonces sólo se estaría usando el 1% de las casillas—una tabla *rala*.

a) [**4 pts.**] Describe una estructura de datos basada en listas ligadas para resolver este problema más eficientemente, en que sólo sea necesario ocupar una cantidad de memoria del orden de los miles de casillas ($4{,}000 \times 5 = 20{,}000$), y no de millones; preferentemente, dibuja la estructura de datos.

Por cada estudiante inscrito en un curso, usamos un registro (objeto) de 5 campos: una identificación del estudiante, una identificación del curso, un puntero a la lista de notas parciales de ese estudiante en ese curso, un puntero (imaginémoslo vertical hacia abajo) al objeto que representa otro curso del mismo estudiante, y un puntero (imaginémoslo horizontal hacia la derecha) al objeto que representa otro estudiante en el mismo curso.

b) [**2 pts.**] Suponiendo que ni los estudiantes ni los cursos cambian durante el semestre (es decir, ni se agregan ni se eliminan), ¿cómo debiera funcionar tu estructura de datos para que el usuario de la aplicación pueda referirse a los cursos por sus códigos (p.ej., IIC2133) y a los estudiantes por sus RUT's?

Usamos un arreglo de tamaño 4,000 para los estudiantes y otro de tamaño 500 para los cursos. Cada casilla de estos arreglos tiene dos punteros: uno a la información del estudiante (rut, nombre) o del curso (código, nombre), respectivamente, y otro a uno de los objetos descritos en **a**). Usamos sendas funciones de hash para convertir rut's o códigos de cursos en índices en estos arreglos, y usamos estos índices como los identificadores de los estudiantes y de los cursos en los objetos de **a**).

2013-2-Ex-P1–Hashing con encadenamiento (1/1)

Examen

Estructuras de Datos y Algoritmos – IIC2133

2 diciembre 2013

Tiempo: 2 horas y 30 minutos

1) En el caso de *hashing* con encadenamiento, describe una forma de almacenar los elementos dentro de la misma tabla, manteniendo todos los casilleros no usados en una *lista ligada de casilleros disponibles*. Para esto, considera que cada casillero puede almacenar un *boolean* y, ya sea, un elemento más un puntero, o dos punteros. Todas las operaciones de diccionario y las que manejan la lista debieran correr en tiempo O(1) en promedio. Específicamente, explica:

a) [**1 pt.**] El papel del *boolean*

El boolean es para saber si el casillero tiene un elemento (y un puntero a otro elemento o null), o si tiene dos punteros (a los casilleros delante y detrás en la lista ***doblemente ligada*** *de casilleros disponibles).*

b) [**3 pts.**] ¿Cómo se implementan las operaciones de diccionario: *insert*, *delete* y *find*?

Llamemos ***lista de colisiones*** *a la lista ligada que se forma al colisionar elementos (similarmente al hashing con encadenamiento). Sea x el dato que nos interesa, y supongamos que al aplicarle la función de hash a x, da k.*

insert*: Si el casillero k está disponible (su boolean vale true), lo sacamos de la lista (en tiempo O(1) porque es doblemente ligada) y almacenamos ahí x. Para esto, cambiamos el boolean a false y ponemos el puntero en nil.*

Si el casillero k está ocupado (su boolean vale false), hay dos posibilidades:

> *Es el primer elemento de una lista de colisiones que comienza en el casillero k: sacamos un casillero disponible, almacenamos x en este casillero, y agregamos el casillero a la lista que comienza en el casillero k*
>
> *O es algún otro elemento en alguna lista de colisiones que comienza en otro casillero: hay que sacar el casillero k de esta otra lista, reemplazándolo por un casillero disponible, y luego almacenar x en el casillero k y poner el puntero de este casillero en nil (es el primer elemento de la nueva lista que contiene a los datos cuyo valor de hash es k)*

delete*: Sabemos que x está en la tabla. El casillero k es el primer casillero de una lista simplemente ligada tal que uno de sus casilleros contiene a x: revisamos la lista hasta llegar a este casillero; entonces, lo sacamos de esta lista (que hay que actualizar apropiadamente) y lo ponemos en la lista de casilleros disponibles.*

find*: Si el casillero k no tiene a x, entonces simplemente seguimos la cadena de punteros que empieza aquí.*

c) [**2 pts.**] ¿Por qué las operaciones de diccionario y las que manejan la lista de casilleros disponibles corren en tiempo O(1) en promedio?

Las operaciones de diccionario operan igual que en el caso de hashing con encadenamiento y, como vimos en clase, esas corren en tiempo O(1) esperado.

Las operaciones sobre la lista de casilleros disponibles corren en tiempo O(1) gracias a que es doblemente ligada (si no, el único problema ocurre cuando se saca un casillero específico de la lista de casilleros disponibles).

2013-2-Ex-P6–Tabla de Hash (1/1)

6) Considera una tabla de hash que queremos compartir entre múltiples procesos. La tabla permite dos operaciones: *insert*, para agregar un nuevo objeto a la tabla, y *find*, para leer la información de un objeto almacenado en la tabla. Como la operación *insert* modifica el contenido de la tabla, exigimos que se ejecute bajo exclusión mutua, tanto con respecto a otras operaciones *insert*, como con respecto a operaciones *find*; es decir, cuando un proceso está ejecutando un *insert*, ningún otro proceso puede estar ejecutando alguna operación. Sin embargo, permitimos múltiples operaciones *find* concurrentes; es decir, si algún proceso está ejecutando un *find*, entonces otros procesos también pueden ejecutar *find* al mismo tiempo.

Escribe los protocolos de sincronización de **a**) *insert* y **b**) *find*, que empleen, por ejemplo, semáforos binarios.

Se necesita un semáforo —r— para asegurar la exclusión mutua total de los insert; se necesita un contador —n— para saber cuántos procesos están haciendo find; y finalmente se necesita otro semáforo —s— para tener acceso a este contador.

```
int n = 0
binarySemaphore r = 1, s = 1

find:
    P(s)
        n = n+1; if (n == 1) P(r)
    V(s)
    —aquí se hace el find propiamente tal
    P(s)
        n = n–1; if (n == 0) V(r)
    V(s)

insert:
    P(r)
    —aquí se hace el insert propiamente tal
    V(r)
```

2013-1-I1-P2–Hashing con encadenamiento (1/1)

2. En el caso de hashing con encadenamiento, propón una forma de almacenar los elementos dentro de la misma tabla, manteniendo todos los casilleros no usados en una *lista ligada de casilleros disponibles*. Para esto, considera que cada casillero puede almacenar un *boolean* y, ya sea, un elemento más un puntero o dos punteros. Todas las operaciones de diccionario y las que manejan la lista debieran correr en tiempo $O(1)$ en promedio. Específicamente, explica lo siguiente:

a) **[0,5 puntos]** El papel del *boolean*.

El boolean es para saber si la casilla tiene un elemento y un puntero a otro elemento (o *null*), o si tiene dos punteros (a las casillas delante y detrás en la lista **doblemente ligada** de casillas disponibles).

b) **[4 puntos]** ¿Cómo se implementan las operaciones de diccionario: inserción, eliminación y búsqueda?

Llamemos **lista de colisiones** a la lista ligada que se forma al colisionar elementos (similarmente al hashing con encadenamiento) y **lista disponible** a la lista de casillas disponibles.

[2 puntos] Inserción: Se aplica la función de hash al nuevo elemento *x*; supongamos que da *k*. Si la casilla *k* está disponible (su boolean vale *true*), la sacamos de la lista (en tiempo O(1) porque está doblemente ligada) y ponemos ahí *x*: cambiamos el boolean a *false* y ponemos el puntero en *null*.

Si la casilla *k* está ocupada (su boolean vale *false*) por un elemento *z*, hay dos posibilidades: *z* hace hash a *k*; o *z* hace hash a un valor distinto de *k* (es decir, es parte de otra lista de colisiones). En el primer caso, hay que agregar *x* en el segundo lugar de la misma lista de *z*, usando una casilla de la lista disponible. En el segundo caso, hay que mover *z* a una casilla disponible y poner *x* en la casilla dejada por *z*, actualizando apropiadamente los punteros involucrados.

[1,25] Eliminación: Sea *k* la casilla a la que hace hash el elemento *x* a eliminar. Si *x* es el único elemento en la lista de colisiones que empieza en la casilla *k*, hay que agregar esta casilla a la lista disponible. Si *x* es el primer elemento, pero no único, en su lista de colisiones, hay que mover el segundo elemento *z* a la casilla *k* y agregar la casilla en que estaba *z* a la lista disponible. Si *x* no es el primer elemento en su lista de colisiones, hay que agregar la casilla que ocupa *x* a la lista disponible, actualizando apropiadamente los punteros.

[0,75 puntos] Búsqueda: Hay que revisar la casilla a la cual el elemento *x* hace hash. Si es una casilla disponible, entonces *x* no está en la tabla. De lo contrario, si *x* no está en la casilla, hay que seguir los punteros.

c) **[1,5 puntos]** ¿Por qué las operaciones de diccionario y las que manejan la lista de casilleros disponibles corren en tiempo $O(1)$ en promedio?

Las operaciones de diccionario operan igual que en el caso de hashing con encadenamiento y, como vimos en clase, esas corren en tiempo O(1) en promedio.

Las operaciones sobre la lista de casillas disponibles corren en tiempo O(1) gracias a que la lista es doblemente ligada (si no, el único problema ocurre cuando se saca una casilla de la lista).

2011-2-I1-P2–Hashing con direccionamiento abierto (1/1)

2. En el caso de hashing con direccionamiento abierto, supón que tienes una tabla de tamaño $T = 10$ y las claves A_5, A_2, A_3, B_5, A_9, B_2, B_9, C_2, en que K_i significa que al aplicar la función de hash h a la clave, obtenemos la posición i. Muestra qué ocurre al insertar las claves anteriores, en el orden en que aparecen, en los siguientes casos:

a) [**3 pts.**] Si usamos revisión lineal, de modo que la posición intentada es $(h(K) + i) \bmod T$.

b) [**3 pts.**] Si usamos revisión cuadrática, de modo que la posición intentada es $h(K)$, $h(K)+1$, $h(K)–1$, $h(K)+4$, $h(K)–4$, $h(K)+9$, ..., $h(K)+(T–1)^2/4$, $h(K)–(T–1)^2/4$.

	a)	**b)**
0	B_9	B_9
1		B_2
2	A_2	A_2
3	A_3	A_3
4	B_2	
5	A_5	A_5
6	B_5	B_5
7	C_2	
8		C_2
9	A_9	A_9

2011-2-Ex-P1–Función de Hash (1/1)

Estructuras de Datos y Algoritmos – IIC2133

Examen

25 noviembre 2011

1. Queremos encontrar la primera ocurrencia de un string de k caracteres de largo en otro string de n caracteres de largo, en que $n > k$. Da un algoritmo de tiempo esperado $O(k+n)$ para este problema.

Podemos aplicar una función de hash al string p, con lo que obtenemos Hp.
Luego, aplicamos la función de hash a cada secuencia de k caracteres consecutivos del string a, partiendo por la que empieza en a1, siguiendo por la que empieza en a2, luego la que empieza en a3, etc. Si el valor de una de estas funciones es igual a Hp, entonces comparamos p con la secuencia correspondiente, carácter por carácter: si son iguales, paramos; de lo contrario, seguimos con la próxima secuencia.
Calcular la función de hash del string p y de la secuencia a1…ak toma tiempo O(k) cada uno. Pero una vez que hemos calculado la función de hash de la secuencia a1…ak , calcular la función de hash de la siguiente secuencia toma tiempo O(1), ya que sólo cambian dos caracteres de la secuencia: el primero, o más significativo, y el último, o menos significativo.
Finalmente, simplemente consideramos que el número esperado de veces que la función de hash de una secuencia sea igual a Hp y que esa secuencia no sea igual al string p es muy pequeño.

2010-2-I1-P1–Hashing con encadenamiento (1/1)

Estructuras de Datos y Algoritmos – IIC2133
I1
13 septiembre 2010

1. En el caso de hashing con encadenamiento, propón una forma de almacenar los elementos dentro de la misma tabla, manteniendo todos los casilleros no usados en una lista ligada de casilleros disponibles. Pare esto, supón que cada casillero puede almacenar un boolean y ya sea un elemento más un puntero o dos punteros. Todas las operaciones de diccionario y las que manejan la lista deberían correr en tiempo esperado $O(1)$. Específicamente, explica lo siguiente:

a) [**1** pt] El papel del boolean.

El boolean es para saber si el casillero tiene un elemento (y un puntero a otro elemento o null), o si tiene dos punteros (a los casilleros delante y detrás en la lista ***doblemente ligada*** *de casilleros disponibles).*

b) [**3** pts.] ¿Cómo se implementan las operaciones de diccionario: inserción, eliminación y búsqueda?

Llamemos ***lista de colisiones*** *a la lista ligada que se forma al colisionar elementos (similarmente al hashing con encadenamiento).*

Inserción*: Se aplica la función de hash al elemento; supongamos que da* k. *Si el casillero* k *está disponible (su boolean vale* true*), lo sacamos de la lista (en tiempo O(1) porque está doblemente ligada) y almacenamos ahí el elemento. Para esto, cambiamos el boolean a* false *y ponemos el puntero en* nil.

Si el casillero k *está ocupado (su boolean vale* false*), hay dos posibilidades: es el primer elemento de la lista de colisiones que comienza en el casillero* k*; o es algún otro elemento en alguna lista de colisiones que comienza en otro casillero.*

Eliminación:

Búsqueda:

c) [**2** pts.] ¿Por qué las operaciones de diccionario y las que manejan la lista de casilleros disponibles corren en tiempo esperado $O(1)$?

Las operaciones de diccionario operan igual que en el caso de hashing con encadenamiento y, como vimos en clase, esas corren en tiempo O(1) esperado.

Las operaciones sobre la lista de casilleros disponibles corren en tiempo O(1) gracias a que es doblemente ligada (si no, el único problema ocurre cuando se saca un casillero específico de la lista de casilleros disponibles).

2010-2-Ex-P1–Hashing con direccionamiento abierto (1/1)

Estructuras de Datos y Algoritmos – IIC2133

Examen

25 noviembre 2010

1) En el caso de hashing con direccionamiento abierto, supón que tienes una tabla de tamaño $T = 10$ y las claves A_5, A_2, A_3, B_5, A_9, B_2, B_9, C_2, en que K_i significa que al aplicar la función de hash h a la clave, obtenemos la posición i. Muestra qué ocurre al insertar las claves anteriores, en el orden en que aparecen, en cada uno de los dos casos siguientes:

a) Si usamos revisión lineal, de modo que la posición intentada es $(h(K) + i) \bmod T$.

b) Si usamos revisión cuadrática, de modo que la posición intentada es $h(K)$, $h(K)+1$, $h(K)–1$, $h(K)+4$, $h(K)–4$, $h(K)+9$, ..., $h(K)+(T–1)^2/4$, $h(K)–(T–1)^2/4$.

a)

0	B_9
1	
2	A_2
3	A_3
4	B_2
5	A_5
6	B_5
7	C_2
8	
9	A_9

b)

0	B_9
1	B_2
2	A_2
3	A_3
4	
5	A_5
6	B_5
7	
8	C_2
9	A_9

4 Grafos

En esta sección se encuentran los siguientes contenidos:

- DFS
- BFS
- Ordenación Topológica
- Componetes Fuertemente Conectados

2020-2-I2-P2–Ordenación topológica, transposición de grafo (1/2)

Pregunta 2

Kojima-san es un desarrollador que quiere crear su propio videojuego. sin embargo, para realizar esta tarea primero necesita instalar un engine y todas sus librerías. Cada librería puede depender a su vez de otras librerías, a las que llamaremos dependencias. Una librería no puede ser instalada a menos que todas sus dependencias ya se encuentren instaladas. Kojima-san podría intentar instalarlas manualmente, sin embargo, él sabe que le tomaría una eternidad. Examinando los requisitos de instalación se percata que existe un total de $\boldsymbol{L}$ librerías, donde cada librería tiene a lo más $\boldsymbol{D}$ dependencias. Considera que el engine en si es una librería con sus propias dependencias.

Definimos un grafo $\boldsymbol{G(V, E)}$ donde cada vértice $\boldsymbol{v} \in \boldsymbol{V}$ corresponde a una librería, si $\boldsymbol{u}$ depende de $\boldsymbol{v}$ entonces hay una arista $\boldsymbol{(v, u)} \in \boldsymbol{E}$

(a) Dado $\boldsymbol{G(V, E)}$, describe un algoritmo $\boldsymbol{\mathcal{O}(L + LD)}$ que entregue un orden de instalación de todas las librerías.

(b) Kojima-san se percata de que ya tiene algunas de las librerías instaladas en su computador, sólo quedando $\boldsymbol{R}$ por instalar. Dado $\boldsymbol{G(V, E)}$, describe un algoritmo $\boldsymbol{\mathcal{O}(R + RD)}$ que entregue un orden de instalación de las librerías que faltan. Asume que detectar si una librería ya está instalada es $\boldsymbol{\mathcal{O}(1)}$

Solución Pregunta 2a)

Es posible notar que se está solicitando la ordenación topológica del grafo formado por las librerías y sus dependencias. De esta forma, se debe realizar el algoritmo `topsort` visto en clases para conseguir el orden de instalación. Cabe notar que si se añaden los nodos según tiempos descendientes es necesario invertir la lista resultante.

La complejidad de `topsort` es de $\boldsymbol{\mathcal{O}(L + LD)}$ debido a que se poseen $\boldsymbol{L}$ nodos y cada nodo posee a lo más $\boldsymbol{D}$ aristas. Finalmente, invertir la lista resultante de $\boldsymbol{L}$ elementos posee una complejidad de $\boldsymbol{\mathcal{O}(L)}$. De esta forma, la complejidad del algoritmo es de $\boldsymbol{\mathcal{O}(L + LD)}$.

[1,5 pt] Por describir el algoritmo correctamente. Pueden aplicar orden topológico directamente.

[0,5 pt] Por justificar la correctitud del algoritmo.

[1 pt] Por justificar la complejidad, indicando que es $\boldsymbol{\mathcal{O}(L + LD)}$.

En caso de plantear un algoritmo no topológico se debe justificar su correctitud y complejidad de manera explícita.

Si el algoritmo propuesto no es $\boldsymbol{\mathcal{O}(L + LD)}$ entonces el puntaje máximo a obtener es 1pt.

Solución Pregunta 2b)

Cabe notar que no es suficiente usar el algoritmo de `topsort` visto en clases: si pintáramos todos los nodos entonces añadiríamos un $\boldsymbol{O(L)}$ a la complejidad, y el algoritmo sería incorrecto.

En cambio, debemos transponer el grafo. Una vez transpuesto, recorremos el grafo desde el nodo inicial (***engine***) para ir verificando las dependencias instaladas. Notemos que transponer el grafo debe ser a lo

2020-2-I2-P2–Ordenación topológica, transposición de grafo (2/2)

más $\mathcal{O}(R + RD)$, por lo que no cualquier algoritmo nos servirá. En particular, asumiremos que tenemos la matriz de adyacencia del grafo, tal que transponerlo sea constante.

Durante el recorrido, cada vez que encontremos una dependencia instalada se deben eliminar o pintar todas las aristas del grafo que lleven al nodo correspondiente y posteriormente eliminar dicho nodo. También, es posible pintar el nodo, tal que lo saltemos cuando lo veamos nuevamente.

Finalmente, se debe realizar `topsort` con el grafo resultante. En el caso de aplicar el algoritmo visto en clases, es necesario invertir la lista resultante para obtener el orden adecuado.

Respecto a la complejidad, la transposición del grafo representado mediante una matriz de adyacencia es $\mathcal{O}(1)$ como ya mencionamos anteriormente. Respecto al recorrido, este tiene una complejidad asociada de $\mathcal{O}(RD)$ debido a que dependencia tiene a lo más D aristas y se visitan a lo más R nodos. Por otra parte, realizar `topsort` en el grafo resultante es $\mathcal{O}(R + RD)$ e invertir la lista retornada es $\mathcal{O}(R)$, obteniendo así una complejidad final de $\mathcal{O}(R + RD)$.

[1,5 pt] Por describir el algoritmo correctamente.

[0,5 pt] Por justificar la correctitud del algoritmo.

[1 pt] Por justificar la complejidad, indicando que es $\mathcal{O}(R + RD)$.

En caso de plantear un algoritmo no topológico se debe justificar su correctitud y complejidad de manera explícita.

2020-1-I3-P1–BFS, Dijkstra (1/3)

1. Sea $G(V, E)$ un grafo direccional con costos enteros y positivos. Se plantea el siguiente algoritmo para encontrar el costo de la ruta más corta entre dos nodos s y f:

   SPW $(G(V, E), s, f)$:

   for each arista $(u, v) \in E$:

   $w =$ el peso de la arista (u, v)

   if $w > 1$:

   Eliminar la arista (u, v) de E

   Creamos $w - 1$ vértices auxiliares $\{x_1, \cdots, x_{w-1}\}$ y los agregamos a V

   Agregamos a E las aristas (x_i, x_{i+1}) para todo i tal que $1 \leq i < w - 1$

   Agregamos a E las aristas (u, x_1) y (x_{w-1}, v)

   Ejecutamos ***BFS*** sobre el grafo, partiendo desde s y buscando la ruta a f

   return profundidad a la que se encontró f

   a. Justifica por qué este algoritmo es correcto
   b. Calcula su complejidad
   c. ¿En qué casos conviene usar Dijkstra para este mismo problema y por qué?

2020-1-I3-P1–BFS, Dijkstra (2/3)

Pontificia Universidad Católica de Chile
Escuela de Ingeniería
Departamento de Computación
IIC2133 – Estructuras de datos y algoritmos
Primer Semestre 2020

Pauta Interrogación 3

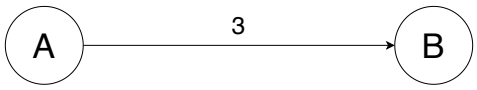
Problema 1

Parte A

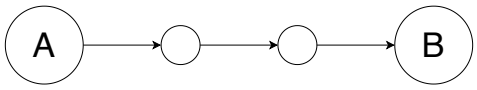
Este algoritmo usa BFS para buscar la ruta más barata de s a f. El problema es que BFS no ve los costos de las aristas, sino que busca la ruta más barata en términos de **cantidad de aristas**. Esto significa que para una arista cualquiera con costo w, BFS la ve como una arista de costo 1.

[1 pts] Por mencionar esta propiedad de BFS.

Lo que hace el for al principio del algoritmo es eliminar las aristas con costo $w > 1$ y reconectar los nodos mediante w aristas de manera que BFS vea el costo de ir entre ese par de nodos como w (aristas) y no 1.



Distancia según BFS: 1



Distancia según BFS: 3

[1 pts] Por mencionar como el algoritmo transforma el grafo para que BFS pueda funcionar.

La pregunta pedía justificar. Si demuestran también se considera correcto.

Parte B

En primer lugar, la transformación recorre todas las aristas, por lo que tenemos E pasos.

Para cada arista de costo $w > 1$, se crean $w - 1$ vértices y w aristas, lo que son $2w - 1$ pasos.

Definimos W como la suma de los costos de todas las aristas del grafo.
La transformación va a incurrir en $\mathcal{O}(W)$ pasos para crear todos los nodos y aristas auxiliares.
[1pts] por calcular correctamente la complejidad de la transformación o primera mitad.

(Es igualmente válido reemplazar W por $|E| * W$ si W se define como el promedio de los costos.)
(Es posible que algunos consideren que esto toma $\mathcal{O}(1)$. Después W aparece en la complejidad de BFS así que

2020-1-I3-P1–BFS, Dijkstra (3/3)

no afecta resultado final, pero primera mitad no obtiene puntaje.)

Luego de la transformación, el grafo va a tener $V' = V + W - E$ = V + O(W) vértices, y $E' = W$ aristas.

Por lo tanto, BFS se va a demorar $\mathcal{O}(V' + E') = \mathcal{O}(V + W + W) = \mathcal{O}(V + W)$

Por lo que el algoritmo en total va a demorar:

$$E + \mathcal{O}(W) + \mathcal{O}(V + W + W)$$

$$\mathcal{O}(V + W + W)$$

[No es necesario pero correcto mencionar que $E \in \mathcal{O}(W)$, por lo que la complejidad queda $\mathcal{O}(V + W)$]

[1pts] por reemplazar correctamente V' y E' y expresar la complejidad final.

Parte C

Si queremos obtener el costo de la ruta más corta entre s y f, es llegar y ejecutar Dijkstra sin modificar el grafo.
[1 pts] Por explicar como usar Dijkstra para este problema.

La complejidad de Dijkstra (usando un min-heap) es

$$\mathcal{O}((V + E) \cdot log(V))$$

Convendrá usar Dijkstra cuando tome menos tiempo que SPW, es decir:

$$(V + E) \cdot log(V) < V + E + W$$

Despejando W, tenemos que nos va a convenir usar Dijkstra cuando

$$W > (V + E) \cdot log(V) - V - E$$

O si simplificaron la E,

$$W > (V + E) \cdot log(V) - V$$

Esto se puede interpretar como que conviene usar Dijkstra cuando el costo de usar la Open es menor que el costo de transformar el grafo y realizar el BFS en SPW. Esto se traduce en cuando los números de vértices son bajos y, sobre todo, cuando los costos totales o promedio son altos.
[1pts] Por llegar a condición a partir de comparación planteada y explicar cuándo conviene Dijkstra

2020-1-I3-P2–Ordenación topológica (1/3)

2. Después de un estresante semestre, Patrick decidió que ya no quiere seguir estudiando computación, y que lo suyo es la Ingeniería Comercial, ya que ahí sí que están las lucas. Estuvo revisando la malla, con sus cursos y requisitos, y quiere saber en cuántos semestres como mínimo podría terminar la carrera. Patrick es un genio así que no hay límite a la cantidad de cursos que puede hacer en un mismo semestre.

   Diseña un algoritmo $\boldsymbol{O}(|\text{cursos}| + |\text{requisitos}|)$ que permita resolver este problema, entregando la lista de cursos que debe tomar en cada semestre. Justifica su complejidad. Puedes suponer que todos los cursos se dictan todos los semestres.

3. Se tiene un grafo direccional $\boldsymbol{G}(\boldsymbol{V}, \boldsymbol{E})$. Se define el grafo $\boldsymbol{G}'(\boldsymbol{V}, \boldsymbol{E}')$ como un grafo que tiene el mismo grafo de componentes que $\boldsymbol{G}$ y cuya cantidad de aristas $|\boldsymbol{E}'|$ es mínima. Escribe un algoritmo eficiente que calcule $\Delta\boldsymbol{E} = |\boldsymbol{E}| - |\boldsymbol{E}'|$.

2020-1-I3-P2–Ordenación topológica (2/3)

Problema 2

Este problema se parece mucho a los que estudiamos en clases cuando vimos orden topológico. Si queremos aplicar algoritmos de grafos para este problema, debemos primero convertirlo en un grafo.

Definimos el grafo $G(V, E)$ de la malla de la siguiente forma:

- Cada vértice $v \in V$ corresponde a un curso distinto.
- Si el curso a tiene como requisito el curso b, entonces existe una arista $(a, b) \in E$

[1 pts] Por definir correctamente el grafo.

Esto se parece mucho a los ejemplos de orden topológico vistos en clases. El problema es que el algoritmo de orden topológico **no tiene noción de tareas que pueden realizarse simultáneamente**, por lo que no podemos usarlo directamente.

[1 pts] Por mencionar este problema con orden topológico

Lo único que sabemos es que los nodos a los que no les llega ninguna arista se pueden tomar en el primer semestre, pero a partir de ahí no es trivial como procedemos.

Opción 1: Algoritmo Cuadrático $\mathcal{O}(V^2)$ [1 pts]

Una vez que tenemos los nodos del primer semestre, podemos eliminarlos a ellos y a las aristas que salen de ellos, y los nodos a los que no les lleguen aristas serán los nodos del 2do semestre. Repitiendo esta estrategia para cada semestre:

```
mallaPorSemestres(G(V, E)):
    for v ∈ V:
        v.incoming ← 0
    for (u, v) ∈ E:
        v.incoming ← v.incoming + 1
    S ← lista de listas, vacía
    while V ≠ ∅:
        x ← lista vacía
        for v ∈ V:
            if v.incoming = 0:
                Insertar v al final de x
                Eliminar v de V
        for x ∈ X:
            foreach arista (x, v) que sale de x:
                v.incoming ← v.incoming − 1
        Agregar x al final de S
    return S
```

2020-1-I3-P2–Ordenación topológica (3/3)

Opción 2: Algoritmo *Linearítmico* $\mathcal{O}(V \cdot log(V))$ [2.5 pts]

La misma estrategia anterior, pero usando un heap para que sea más eficiente

```
mallaPorSemestres(G(V, E)):
    for v ∈ V:
        v.incoming ← 0
    for (u, v) ∈ E:
        v.incoming ← v.incoming + 1
    Convertir V en un min − heap, ordenando por v.incoming
    S ← lista de listas, vacía
    while V ≠ ∅:
        x ← lista vacía
        while V ≠ ∅  ∧  V.root.incoming = 0:
            Extraer v de V
            Insertar v al final de x
        for x ∈ X:
            foreach arista (x, v) que sale de x:
                v.incoming ← v.incoming − 1
        Agregar x al final de S
    return S
```

Opción 3: Algoritmo Lineal $\mathcal{O}(V + E)$ [4 pts]

Recorrer el grafo con DFS/BFS a partir de los cursos del primer semestre, entrando a un curso (vértice) sólo una vez que ya se entró a **todos** sus requisitos (ancestros). El semestre de un curso será 1 más que el semestre de su requisito con el mayor semestre.

```
mallaPorSemestres(G(V, E)):
    for v ∈ V:
        v.incoming ← 0
        v.semestre ← 0
    for (u, v) ∈ E:
        v.incoming ← v.incoming + 1
    Open ← lista vacía
    for v ∈ V:
        if v.incoming = 0:
            Agregar v a Open
            v.semestre ← 1
    while Open ≠ ∅:
        Extraer un vértice x cualquiera de Open
        foreach arista (x, v) que sale de x:
            v.incoming ← v.incoming − 1
            if v.incoming = 0:
                Insertar v en Open
            if x.semestre + 1 > v.semestre:
                v.semestre ← x.semestre + 1
```

Al terminar el algoritmo, cada vértice tiene asociado su semestre. Podemos recorrer los vértices una última vez

2020-1-I3-P3–Componentes Fuertemente Conectadas, algoritmo Kosaraju (1/3)

3. Se tiene un grafo direccional $\boldsymbol{G}(\boldsymbol{V}, \boldsymbol{E})$. Se define el grafo $\boldsymbol{G}'(\boldsymbol{V}, \boldsymbol{E}')$ como un grafo que tiene el mismo grafo de componentes que $\boldsymbol{G}$ y cuya cantidad de aristas $|\boldsymbol{E}'|$ es mínima. Escribe un algoritmo eficiente que calcule $\Delta \boldsymbol{E} = |\boldsymbol{E}| - |\boldsymbol{E}'|$.

2020-1-I3-P3–Componentes Fuertemente Conectadas, algoritmo Kosaraju (2/3)

Problema 3

Podemos separar las aristas E' en dos grupos:

1. Las aristas que están **dentro** de una CFC, E'_o
2. Las aristas que están **fuera** de todas las CFC, E'_x

En el primer caso, tenemos que por cada CFC G con $n > 1$ nodos, E'_o tiene n aristas, ya que es necesario preservar **un** solo ciclo que pase por todos los nodos de la CFC para que siga siendo una CFC en G'. Es decir, si definimos

$$C = \{c \mid c \text{ es una CFC de } G\}$$

Entonces

$$|E'_o| = \sum_{c \in C,\ |c| > 1} |c|$$

[2 pts] Por identificar cuantas aristas aportan las CFC; si no menciona el caso borde de las CFCs de un nodo, 1pt. (1).

En el segundo caso, tenemos que las aristas que no están dentro de ninguna CFC son las aristas que van de una CFC a otra. En G' no pueden haber múltiples aristas que conecten un mismo par de CFCs, ya que en el grafo de componentes se reducen a 1, por lo que sobrarían aristas.

Es decir, si $H(C, F)$ es el grafo de componentes de G,

$$|E'_x| = |F|$$

[2 pts] por identificar cuantas aristas quedan fuera de las CFC; si asume grafo conexo, 1pt. (2).

Teniendo esto, podemos calcular $|E'|$ sin necesidad de computar un G'. Pero sí va a ser necesario calcular el grafo de componentes. El algoritmo es como sigue:

Primero, para identificar las CFCs, usamos el algoritmo de Kosaraju como se vió en clases, pero le hacemos una pequeña modificación a la función que asigna representantes para calcular el tamaño de la CFC:

```
assign(u, rep):
    total ← 0
    if u.rep = ∅:
        u.rep ← rep
        foreach v in α[u]:
            total ← total + assign(v, rep)
    return total
```

2020-1-I3-P3–Componentes Fuertemente Conectadas, algoritmo Kosaraju (3/3)

Teniendo eso podemos calcular $|E'_o|$ sumando la cantidad de nodos de cada CFC con más de un nodo, mientras que para calcular $|E'_x|$ simplemente computamos F:

```
|E'|(G(V, E)):
    |E'_o| ← 0
    Crear lista L vacía
    Ejecutar dfs(G) con tiempos
    Insertar vértices en L en orden decreciente de tiempos f
    for each u in L:
        n ← assign(u, u)
        if n > 1:
            |E'_o| ← |E'_o| + n
    F ← ∅
    for each (u, v) ∈ E:
        if u.rep ≠ v.rep:
            if (u.rep, v.rep) ∉ F:
                Agregar (u.rep, v.rep) a F
    return |E'_o| + |F|
```

Y una vez obtenido $|E'|$, podemos fácilmente retornar $\Delta E = |E| - |E'|$

Este algoritmo es $\mathcal{O}(V + E)$, ya que Kosaraju es $\mathcal{O}(V + E)$ y computar F es $\mathcal{O}(E)$

[4 pts] Por algoritmo correcto (Correctitud)

[2 pts] por que sea eficiente, siempre y cuando sea correcto (Eficiencia)

El puntaje de la pregunta será `max((1)+(2),(Correctitud)) + (Eficiencia)`.

2019-2-I2-P2-1–DFS, BFS (1/1)

PONTIFICIA UNIVERSIDAD CATÓLICA DE CHILE
ESCUELA DE INGENIERÍA
DEPARTAMENTO DE CIENCIA DE LA COMPUTACIÓN

IIC2133 — Estructuras de Datos y Algoritmos — 2/2019
Solución I2P2

Pregunta 1

Algoritmo para obtener el diámetro de un árbol usando DFS.
Partimos escribiendo un algoritmo que use DFS para obtener el nodo mas lejano a un nodo cualquiera v. (También es válido incluir DFS directamente en el algoritmo para obtener el diámetro, o implementar BFS).

DFS(v, parent, dist):

```
max_dist = dist
furthest = v
for vecino in α[v] do
  if vecino ≠ parent then
    dist_lejana, nodo_lejano = DFS(vecino, v, dist + 1)
    if max_dist < dist_lejana then
      max_dist = dist_lejana
      furthest = nodo_lejano
    end if
  end if
end for
return max_dist, furthest
```

Luego, usando esta función, podemos encontrar el nodo mas lejano al nodo elegido, el cual será un extremo del diámetro, el otro extremo (junto con el valor del diámetro) lo encontramos ejecutando la misma función, pero empezando por un extremo.

DIAMETRO(G(V, E))

```
v ← nodo extraído al azar
extremo, _ = DFS(v, NULL, 0)
_, diametro = DFS(extremo, NULL, 0)
return diametro
```

[2 pts] Implementar BFS o DFS.
[2 pts] Encontrar un extremo del diámetro a partir de un nodo al azar.
[2 pts] Encontrar el largo del diámetro a partir de un extremo.

2019-2-I2-P2-2–Ruta de menor costo, Dijkstra, DFS (1/2)

Pregunta 2

Dado un grafo dirigido con costos $G(V, E)$ tal que el costo de cada arista puede ser 1 o 2.

a) Explica cómo encontrar la ruta de menor costo entre dos nodos dados usando BFS. Puedes modificar el grafo o el algoritmo para resolver este problema.

b) Demuestra que tu solución en a) es correcta.

c) Calcula la complejidad del algoritmo propuesto en a) con respecto a $|V|$ y $|E|$

Solución:
Existen varias soluciones diferentes y correctas, veremos 2 de ellas pero si lo hicieron de otra forma válida se considera correcto también:

1) La primera es modificar el gráfo:

a) Se reemplazan cada una de las aristas de costo 2 por 2 aristas de costo 1 y un nodo intermedio. Luego podemos utilizar BFS ya que tenemos un grafo con todas las aristas de costo 1 equivalente a el grafo anterior.

b) Para demostrar que esto es correcto es importante mencionar 2 cosas. Primero las rutas del grafo original tienen el mismo costo que las rutas del grafo modificado. Segundo, está demostrado que BFS encuentra la ruta más corta en un grafo sin costos (que es igual que un grafo con todas las aristas del mismo costo) por lo que la ruta obtenida es la óptima.

c) Sabemos que la complejidad de realizar BFS en un grafo de $|V|$ nodos y $|E|$ aristas es de $\mathcal{O}(|V| + |E|)$. Si el grafo inicial tenía $|V|$ nodos y $|E|$ aristas y reemplazamos todas las aristas de costo 2 por 2 aristas de costo 1 y un nuevo nodo intermedio, en el peor caso todas son de costo 2 por lo que se duplican las aristas y se agregan tantos nodos como aristas habían, quedando $|V| + |E|$ nodos y $2 \cdot |E|$ aristas. Entonces la complejidad de BFS quedaría $\mathcal{O}(|V| + 3 \cdot |E|)$ que es equivalente a $\mathcal{O}(|V| + |E|)$.

2) La segunda es modificar el algoritmo:

a) La modificación que se hace el algoritmo es al orden en que se revisan los nodos en la cola. BFS revisa los nodos en orden de profundidad pero entre nodos en la misma profundidad no hace una diferencia que en este caso, con aristas con costos mayores, si afecta el orden en que se revisan los nodos para llegar a la optimalidad. Por lo tanto, al revisar un nodo se almacenará el costo acumulado en llegar a ese nodo. Luego, en la cola de los nodos por revisar, se revisará primero el nodo cuyo costo acumulado para alcanzar es el más bajo. Si al expandirlo encontramos una forma menos costosa de llegar a uno de los nodos que queda por expandir, le cambiamos el costo por el menor de los dos. Se encontrará una solución cuando el nodo que se expandirá es el nodo final.
Esto es equivalente al algoritmo de Dijkstra.

b) Para demostrar que el algoritmo es correcto hay que demostrar que se expande cada nodo con la ruta a el de costo mínimo. Para demostrar esto realizaremos una inducción.

Caso Base: En la primera iteración del algoritmo entra a la cola el nodo inicio y se expande. Es trivial que al expandir el nodo inicio, se tiene la ruta óptima hacia él de costo 0.

Hipotesis de Inducción: En la iteración n del algoritmo, (en que se han sacado n nodos de la cola) asumimos que cada nodo fue sacado con su costo mínimo.

PD: Para el nodo que se expande en la siguiente iteración se llega con el costo mínimo posible.
A partir de que en la iteración anterior se había llegado al nodo expandido de forma óptima tenemos los siguientes casos en la cola:

2019-2-I2-P2-2–Ruta de menor costo, Dijkstra, DFS (2/2)

1) Nodos que tienen el mismo costo de descubrimiento que el que se acaba de expandir.
2) Nodos con costo de descubrimiento +1 que el que se acaba de expandir
3) Nodos con costo de descubrimiento +2 que el que se acaba de expandir

Actualmente estamos sacando los nodos de 1) y agregando sus vecinos a 2) y a 3) según corresponda el costo de la arista para llegar a ellos. Todos los nodos de 1) fueron descubiertos con su costo mínimo ya que fueron descubiertos por algún nodo que ya salió de la cola y nuestra hipótesis de inducción nos dice que los que salieron de la cola fueron descubiertos con el costo mínimo.

Los nodos en 2) se separan en 2 casos: Los nodos que fueron descubiertos con una arista de costo 1 por un nodo a la profundidad actual y los nodos que fueron descubiertos por una arista de costo 2 por un nodo de la iteración anterior. En ambos casos el costo con el que saldrán de la cola es el mínimo para ese nodo.

Finalmente los nodos de 3) están en esa cola porque fueron descubiertos por un nodo a la profudidad actual por una arista de costo 2. Es posible que sacando nodos de 1) encontremos una arista de costo 1 que redescubra uno de esos nodos a menor profundidad. En ese caso el nodo se cambia su costo de descubrimiento y cambia al caso 2). En el caso en que no se redescubra de esta manera entonces la manera más barata de llegar al nodo es con esa arista final de costo 2.

Los nodos del caso 1) se expanden primero, luego del caso 2) y luego del caso 3). Por lo que queda demostrado que se expandiran con costo mínimo por lo que el algoritmo funciona.

Otra forma de demostrarlo es hablando de ordenar la cola segun el costo de descubrimiento y demostrar por que con eso se llega con costo mínimo a cualquier nodo.

No es correcto revisar todas las rutas posibles al nodo destino utilizando BFS por que BFS para funcionar asume que cuando encuentra la ruta a un nodo y lo expande esa ruta es la más corta. (Con costos 1 y 2 esto no se puede asegurar).

c) La complejidad de la adaptación del algoritmo depende de la implementación de la cola (o colas) para llevar a cabo el algoritmo. Si se utiliza un Heap para ordenar la cola entonces la complejidad del algoritmo sería de $\mathcal{O}(|V| + |E| + |V| \cdot log(V)$ por ordenar el heap para cada uno de los nodos que se revisan. Si se utiliza una cola para almacenar los nodos de cada uno de los casos 1), 2) y 3) entonces la complejidad es de $\mathcal{O}(|V| + |E|)$

2019-1-C4-1–DFS modificado (1/1)

Estructuras de Datos y Algoritmos - IIC2133
Control 4
6 de mayo, 2019

1) Un *puente* en un grafo no direccional es una arista (*u, v*) tal que al sacarla del grafo hace que el grafo quede desconectado -o, más precisamente, aumenta el número de componentes conectadas del grafo; en otras palabras, la única forma para ir de *u* a *v* en el grafo es a través de la arista (*u, v*). **[4 pts.]** Explica cómo usar el algoritmo DFS para encontrar eficientemente los puentes de un grafo no direccional; y **[2 pts.]** justifica qué tan eficientemente. Recuerda que DFS asigna tiempos de descubrimiento (y finalización) a cada vértice que visita.

R: Para hallar los puentes en el grafo lo que se hará, básicamente, es buscar aquellas aristas que no pertenezcan a algún ciclo dentro de este. Es posible encontrar dichas aristas utilizando el algoritmo DFS. Teniendo nuestro bosque, generado por el algoritmo, supongamos que tenemos una arista (*u, v*) y que en el bosque no es posible llegar desde un descendiente de *v* hasta un ancestro de *u*, entonces diremos que dicha arista no pertenece a ningún ciclo y en consecuencia, es un puente. Para detectar de manera eficiente si una arista es un puente, lo que se hará es guardar, en cada nodo, el menor de los tiempos de descubrimiento de cualquier nodo alcanzable (no necesariamente en un paso) desde donde me encuentre, llamémoslo *v.low*. Si la arista es la (*u, v*), esto es:

$$u.low = min\{u.d, v.low\}$$

Ahora, al momento de retornar el método *dfsVisit*, lo que se hará es comparar dicho valor con el tiempo de descubrimiento del nodo en el que estoy parado. Digamos estoy parado en el nodo *u* y visité *v*, si se tiene que $u.d < v.low$ entonces no se puede alcanzar ningún ancestro de *u* desde algún descendiente de *v*, en consecuencia dicha arista no pertenece a ningún ciclo y es un puente.

Esta forma de detectar puentes en el grafo posee la misma complejidad que el algoritmo DFS, O(|*V*| + |*E*|). Esto ya que lo único que se está haciendo es actualizar un valor y compararlo, lo que no suma mayor complejidad (se hace en tiempo constante).

- **[4 pts]** Se explica un algoritmo o bien las modificaciones que se le deben realizar a DFS para lograr el objetivo de manera clara y concisa. Solución debe ser eficiente. Si no se cumple con los puntos anteriores no hay puntaje.
- **[2 pts]** Solo si el algoritmo es correcto (efectivamente encuentra los puentes) y se justifica el por qué de la eficiencia mostrada.

2019-1-Ex-P4–Componentes Fuertemente Conectados (1/3)

4. Tienes que realizar un conjunto de tareas en un computador y queremos determinar el orden en que deben realizarse, teniendo en cuenta que existe una lista de dependencias (a, b), que representan el hecho de que la tarea a debe hacerse antes que la tarea b. De esta manera, se forma un grafo direccional que, sin embargo, podría tener ciclos; interpretamos estos ciclos como que todas las tareas que forman un determinado ciclo pueden ser realizadas al mismo tiempo (en paralelo).

2019-1-Ex-P4–Componentes Fuertemente Conectados (2/3)

Explica cómo puedes determinar el orden en que deben realizarse las tareas de manera que se cumplan todas las dependencias, indicando además los subconjuntos de tareas que se pueden hacerse en paralelo.

Solución:

Si no hubiera ciclos, el orden de las tareas es simplemente el orden topológico del grafo. Pero como el orden topológico no existe si el grafo tiene ciclos, debemos hacer algo al respecto. **[0.5pt]**

Lo primero que hay que hacer es identificar cuales son las tareas que deben ser realizadas al mismo tiempo. El detalle es que la cantidad de ciclos puede ser exponencial, por lo que no sirve “encontrar todos los ciclos”. Lo que hay que hacer en lugar de eso es determinar la cantidad de ejecuciones en paralelo que hay que hacer. Llamaremos a cada una de esas una **super-tarea (ST)**: un conjunto de tareas que deben ser ejecutadas en paralelo.

La propiedad que nos interesa es que las STs corresponden a las componentes fuertemente conectadas del grafo con más de un elemento. [2.5pt]

Demostración:

Sabemos que las tareas de un ciclo corresponden a una ST, pero una ST podría contener a más de un ciclo.

Tomamos la siguiente definición de ciclo: si “a” pertenece a un ciclo, significa que existe una ruta de “a” hasta “a” pasando por al menos un nodo distinto de “a”.

Si “x” pertenece a la misma ST que “a”, significa que existe un ciclo que los contiene a ambos. Esto a su vez significa que existe una ruta de “a” a si misma pasando por “x”.

Sea T = {a, x_1, … , x_n} una ST que contiene a este nodo arbitrario “a”

{x_1, … , x_n} corresponden a todas las tareas que están el la misma ST que “a”.

Esto significa que para cada “x_i” existe una ruta de “a” a “x_i”, y una ruta de “x_i” a “a”.

Esto significa que para cada $i \neq j$, existe una ruta de “x_i” a “x_j”, pasando por “a”.

Es decir, T es un conjunto de nodos donde todos los nodos son alcanzables entre ellos.

Esa es la definición de componente fuertemente conectada.

$\therefore$ T es una CFC.

[1.5pt por demostrar o justificar correctamente esta propiedad]

2019-1-Ex-P4–Componentes Fuertemente Conectados (3/3)

Podemos usar el algoritmo para encontrar las CFC que se vió en clases para encontrar todas las CFC del grafo. Sea $\{T_1, \dots, T_n\}$ el conjunto de CFCs con más de un elemento, es decir un conjunto de STs.

Una vez encontradas todas las STs podemos eliminar del grafo los nodos que pertenecen a alguna ST, así como las dependencias entre elementos que pertenecen a la misma ST. Luego agregamos las STs como nuevas tareas al grafo, de la siguiente manera:

Si un nodo “u” pertenece a la ST T_i , y un nodo “v” pertenece a la ST T_j , y existe la dependencia (u, v), esta se elimina del problema y se agrega la dependencia entre STs, (T_i , T_j). Luego cada dependencia de la forma (a, u), donde a no pertenece a una ST, se elimina del problema y se agrega la dependencia (a, T_i). Por otro lado, cada dependencia de la forma (u, a), donde a no pertenece a una ST, se elimina del problema y se agrega la dependencia (T_i , a).

[1pt por la construcción de este nuevo grafo]

Ahora que tenemos un grafo acíclico podemos determinar su orden topológico usando el algoritmo visto en clases, que es el orden de tareas que se pide. **[0.5pt]**

2018-2-I3-P2–Ordenación topológica (1/2)

2. **Ordenación topológica**

En clase estudiamos un algoritmo para ordenar topológicamente un grafo direccional acíclico $G = (V, E)$; el algoritmo consiste esencialmente en hacer un recorrido DFS del grafo.

a) ¿Cuál es la complejidad del algoritmo estudiado en clase?

Respuesta: [1.5 ptos] La complejidad del algoritmo es de $O(|V|+|E|)$.

2018-2-I3-P2–Ordenación topológica (2/2)

Considera ahora el siguiente algoritmo (que fue sugerido por algunos de ustedes en esa misma clase) para ordenar topológicamente un grafo direccional acíclico $G = (V, E)$.

1. Encontrar un vértice que no tenga aristas que lleguen a él.
2. Imprimir el vértice, y sacarlo del grafo junto a todas las aristas que salen de él.
3. Si aún quedan vértices en el grafo, volver a 1.

Si representamos el grafo mediante listas de vértices adyacentes, tal como vimos en clase, responde:

b) Explica claramente cómo implementar el algoritmo anterior —es decir, ¿cómo encuentras un vértice que no tenga aristas que lleguen a él?, y ¿cómo lo "sacas" del grafo junto a todas las aristas que salen de él?

Respuesta: [1.5 ptos] Es correcta cualquier implementación, en la medida que esté bien explicada.

Se descuenta puntaje por detalles en cuanto a la implementación.
Se dio 0 puntos si el algoritmo estaba incorrecto.

c) … y deduce su complejidad en términos del número de vértices, $|V|$, y del número de aristas, $|E|$.

Respuesta: [1.5 ptos] La complejidad debe ser correcta dependiendo de la implementación en (b).

d) Explica cómo mejorar la implementación del algoritmo, en b), de modo que la complejidad sea ahora $O(|V|+|E|)$. En el caso que tu algoritmo en b) ya tenga esa complejidad demuéstrala formalmente.

Respuesta: [1.5 ptos] Una posible forma de hacer que el algoritmo tenga una complejidad de $O(|V|+|E|)$ es agregar, a cada nodo, un contador que indique la cantidad de aristas que llegan hacia el. Al comienzo del algoritmo, setear los valores del contador toma $O(|V|+|E|)$, luego se pueden recorrer todos los nodos, en $O(|V|)$, e ir agregando a una cola (o un *stack*) aquellos nodos que tengan su contador en 0. Luego, se van sacando los nodos de la cola. Cuando se saca un nodo de la cola se debe recorrer su lista de adyacencia y para cada nodo al cual se conecta, disminuir el contador en 1, si el contador llega a 0, agregamos dicho nodo a la cola. El algoritmo termina cuando no hay nodos en la cola. Como solo se está haciendo un recorrido por las listas de adyacencia de cada nodo y sacar y agregar los nodos a la cola toma O(1), el algoritmo posee una complejidad de $O(|V|+|E|)$.

Esta implementación es solo una de las posibles, puede haber más. Lo importante era que tuviese la complejidad solicitada.

2018-1-I2-P3–DFS modificado (1/3)

3. Búsqueda en profundidad (DFS)

La siguiente en una versión más general del algoritmo visto en clases para recorrer un grafo G a partir de un vértice v:

```
DFS(G, v):
    marcar v
    S ß v                  —inicializa un stack de vértices, S, con v
    while S no está vacío :
            w ß S    —extrae el vértice en el top de S y asígnalo a w
            "visitar" w
            for all x tal que (w, x) es una arista de G :
            if x no está marcado:
            marcar x
            S ß x
```

2018-1-I2-P3–DFS modificado (2/3)

a) Supongamos que G es no direccional (las aristas no tienen dirección). Decimos que G es **biconec-tado** si no hay ningún vértice que al ser sacado de G desconecta el resto del grafo. Por el contrario, si G no es biconectado, entonces los vértices que al ser sacados de G desconectan el grafo se conocen co-mo **puntos de articulación**.

Describe un algoritmo eficiente para encontrar todos los puntos de articulación en un grafo conectado. Explica cuál es la complejidad de tu algoritmo.

Respuesta.

Para obtener un algoritmo eficiente que resuelve este problema, utilizamos una versión adaptada de DFS que pueda detectar aristas hacia atrás en el grafo no direccional. El principio sobre el que se basa la solución es que luego de hacer DFS en un grafo no dirigido hay dos posibilidades para que un vértice sea punto de articulación:

1. La raíz del árbol DFS generado es un punto de articulación: esto ocurre cuando la raíz tiene más de un hijo, pues significa que la única forma de descubrir todos estos hijos, fue partir una nueva rama de la búsqueda desde la raíz. Si no fuera así y la raíz no es punto de articulación, entonces al sacarla los hijos deberían tener otro camino que los una. *Este será el primer chequeo: verificar si la raíz tiene más de un hijo en el árbol DFS.* **[1 pt. Por caso raíz]**
2. Un vértice que no es raíz es punto de articulación: esto ocurre si para un nodo u todos sus descendientes no cuentan con un camino que llegue a un antecesor de u en el árbol DFS. Esto se puede verificar en tiempo constante si se agrega un atributo a cada nodo que se define según

   $$u.low = min\{u.disc, w.low \text{ para algún } w \text{ descendiente de } u\}$$

   Este atributo guarda el tiempo de descubrimiento de u o el de un ancestro alcanzable con alguna arista hacia atrás. Con esta estrategia, los puntos de articulación (no raíz) son aquellos que tienen el atributo *low* menor que el de sus descendientes directos. **[1 pt. Por estrategia general]**

Con esto, el algoritmo DFS que incorpora el atributo *low* a cada vértice seguido de una revisión para cada vértice permite entregar los puntos de articulación de todo el grafo. En resumen:

1. Ejecutar DFS que registra tiempos de descubrimiento y *low* (paso completo en $O(V+E)$)
2. Revisar cada vértice en el árbol DFS generado y comparar *low* con sus descendientes. Agregar a una lista aquellos vértices que no cumplan la condición buscada (paso completo en $O(V+E)$).
3. Revisar cuántos hijos tiene la raíz del árbol y agregarla si tiene más de uno (paso completo en $O(1)$).

Por lo tanto, la complejidad del algoritmo es $O(V+E)$ para un grafo representado con listas de adyacencia.

[1 pt. Por explicación de complejidad]

b) Considera el siguiente grafo G direccional (las aristas tienen dirección), representado mediante sus listas de adyacencias:

[0]: 5–1	[1]:	[2]: 0–3	[3]: 5–2	[4]: 3–2	[5]: 4	
[6]: 9–4–0	[7]: 6–8	[8]: 7–9	[9]: 11–10	[10]: 12	[11]: 4–12	[12]: 9

Ejecuta el algoritmo DFS anterior a partir del vértice $v = 0$. Muestra el orden en que los vértices van siendo marcados y el contenido del stack S cada vez que cambia.

2018-1-I2-P3–DFS modificado (3/3)

Respuesta (posible solución).

Partiendo de 0, lo marcamos y lo ponemos en *S*; *S*= [0].

[A partir de ahora, cada paso = **0.5 pts.**]

Al sacar 0 (*S*= [ ]) y asignarlo a *w*, vemos que sus vecinos son 5 y 1; los marcamos y los ponemos en *S*; *S*= [5, 1].
Al sacar 5 (*S*= [1]) y asignarlo a *w*, vemos que su único vecino es 4; lo marcamos y lo ponemos en *S*; *S*= [4, 1].
Al sacar 4 (*S*= [1]) y asignarlo a *w*, vemos que sus vecinos son 3 y 2; los marcamos y los ponemos en *S*; *S*= [3, 2, 1].
Al sacar 3 (*S*= [2, 1]) y asignarlo a *w*, vemos que sus vecinos son 5 y 2, ambos ya marcados; *S*= [2, 1].
Al sacar 2 (*S*= [1]) y asignarlo a *w*, vemos que sus vecinos son 0 y 3, ambos ya marcados; *S*= [1].
Al sacar 1 (*S*= [ ]) y asignarlo a *w*, vemos que no tiene vecinos; *S*= [ ] y terminamos.

2017-1-I3-P1–Componentes Fuertemente Conectados (1/1)

PONTIFICIA UNIVERSIDAD CATÓLICA DE CHILE
ESCUELA DE INGENIERÍA
DEPARTAMENTO DE CIENCIA DE LA COMPUTACIÓN

IIC 2133 — Estructuras de Datos y Algoritmos
Interrogación 3
Primer Semestre, 2017

Duración: 2 hrs.

1. *a*) (1/3) Sea $G = (V, E)$ un grafo *dirigido acíclico*. ¿Cuántas componentes fuertemente conexas encuentra el algoritmo de Kosaraju al ser ejecutado en G? Justifique.

   Respuesta (1 punto): El algoritmo de Kosaraju encuentra todas las componentes fuertemente conexas de un grafo.
   Un grafo dirigido acíclico es un grafo que (valga la redundancia) no posee ningún ciclo y sus nodos se conectan en un solo sentido. Al no existir ciclos, no existe camino que permita a un nodo conectarse con si mismo.

   Por otro lado, una componente fuertemente conexa, se define como un grupo de nodos en el que todos los nodos pertenecientes a esta son alcanzables desde cualquier nodo de la componente.

   Dado esto, no pueden haber componentes fuertemente conexas si no se permiten ciclos en un grafo. Por ende, El grafo G no posee componentes fuertemente conexas. *Hasta acá se obtiene todo el puntaje.* Si se define que un nodo es alcanzable por si mismo, entonces la cantidad de componentes fuertemente conexas es $|V|$.

   b) (2/3) Suponga que el algoritmo de abajo se ejecuta con un grafo **dirigido acíclico** G.

   i. Ejecute DFS sobre G, almacenando los tiempos de finalización.
   ii. Ejecute DFS-visit(G, u), donde u es el nodo de G que tiene el mayor tiempo de finalización.
   iii. Si todos los nodos están marcados negros, retorne TRUE. En caso contrario, retorne FALSE.

   ¿Qué propiedad tiene G cuando el algoritmo retorna TRUE? Justifique.

   Respuesta (2 puntos): El nodo u cumple la propiedad de no tener ancestros y ser ancestro de todos los nodos del grafo (por ende todos los nodos $-\{u\}$ tienen mínimo un ancestro). **(1 punto)** ¿Por qué?

   Por el paso i (DFS), se almacenan todos los tiempos de finalizacón. El nodo que se cierra último (último en volverse negro), en un grafo acíclico, es el primero en el orden topológico correspondiente y este nodo no tiene ancestros. **(0,25 puntos)** DFS-visit es el algoritmo DFS pero que parte de un nodo y sigue solo los caminos alcanzables por ese nodo. Luego se detiene. **(0,25 puntos)**

   Un nodo se marca negro cuando visitó a todos sus descendientes. Si todos los nodos están marcados negros, se visitó a todos los nodos del grafo. Por ende, si un DFS-visit(G,u) marca a todos los nodos negros, significa que se logró alcanzar a todos los nodos del grafo desde u. Dado que este grafo no tiene ciclos, u no tiene ancestros. **(0,5 puntos)**

   Estos son puntajes parciales, en caso de no tener la respuesta completa o correcta.

2016-2-I2-P1–DFS modificado (1/1)

Estructuras de Datos y Algoritmos – iic2133

I2

3 octubre 2016

Contesta sólo 4 preguntas

1. Un *puente* en un grafo no direccional es una arista que, si se saca, separaría el grafo en dos subgrafos disjuntos. Describe un algoritmo eficiente que encuentre todos los puentes de un grafo no direccional; justifica la eficiencia y la corrección de tu algoritmo.

Respuesta:

Una arista (u, v) es un puente Û (u, v) no pertenece a un ciclo; dfs encuentra los ciclos. Por lo tanto, ejecutemos dfs y miremos el bosque dfs producido. Recordemos que dfs en un grafo no direccional sólo produce aristas de árbol y hacia atrás; **no** produce aristas hacia adelante ni cruzadas.

Una arista de árbol (u, v) en el bosque dfs es un puente Û no hay aristas hacia atrás que conecten un descendiente de v a un ancestro de u. ¿Cómo determinamos esto? Recordemos el tiempo de descubrimiento $v.d$ que dfs asigna a cada vértice v : una arista hacia atrás (x, y) conectará un descendiente x de v a un ancestro y de u si $y.d < x.d$.
Luego, si *v.d* es menor que todos los tiempos *d* que pueden ser alcanzados mediante una arista hacia atrás desde cualquier descendiente de *v*, entonces (*u*, *v*) es un puente.

Es decir, basta modificar un poco `dfsVisit` visto en clase: asignar a cada vértice v un tercer número $v.min$ que sea el menor de todos los $y.d$ alcanzables desde v mediante una secuencia de cero o más aristas de árbol seguida de una arista hacia atrás; si al retornar `dfsVisit`(v), $v.d = v.min$, entonces (u, v) es un puente. Esta modificación no cambia la complejidad de dfsVisit.

El puntaje distribuye así:

- 4 ptos por el algoritmo (Se toma en cuenta cuán eficiente es).
- 1 pto por justificar correctamente la eficiencia, independiente que tan bueno sea el algoritmo y suponiendo que este es correcto.
- 1 pto por justificar la correctitud del algoritmo.

2015-2-I2-P2–DFS, Componentes Fuertemente Conectados (1/2)

2. **a**) Considera un grafo no direccional. Queremos pintar los vértices del grafo, ya sea de azul o de amarillo, pero de modo que ***dos vértices conectados por una misma arista queden pintados de colores distintos***.

Da un algoritmo eficiente que pinte los vértices del grafo según la regla anterior, o bien que se dé cuenta de que no se puede; justifica la corrección de su algoritmo.

La idea es usar DFS, que es O(V+E):

1) Recordemos que al aplicar DFS a un grafo no direccional sólo aparecen aristas de árbol y hacia atrás; no hay aristas hacia adelante ni cruzadas.

2) Por lo tanto, en el árbol DFS resultante, si a partir de un vértice v *surgen varias ramas, significa que en el grafo original cualquier ruta que va de los vértices en una de esas ramas a los vértices en otra necesariamente pasa por* v.

3) Por lo tanto, a partir de la raíz del árbol DFS podemos pintar los vértices alternando colores a medida que bajamos por cada rama.

4) Finalmente, hay que revisar las aristas hacia atrás del árbol DFS, cuya presencia refleja la existencia de ciclos en el grafo original:
- *Si alguna arista hacia atrás conecta (directamente) vértices pintados del mismo color, entonces el grafo* ***no se puede pintar*** *como se pide en el enunciado.*
- *Pero si el árbol DFS no tiene aristas hacia atrás, o si ninguna arista hacia atrás conecta vértices pintados del mismo color, entonces es posible pintar los vértices del grafo original como se pide en el enunciado; la asignación de colores definida en el paso 3) es una forma concreta de hacerlo.*

b) Considera el siguiente grafo direccional G, representado mediante listas de vértices adyacentes:

[0] → 5, 1	[4] → 3, 2	[8] → 7, 9	[12] → 9
[1] →	[5] → 4	[9] → 11, 10	
[2] → 0, 3	[6] → 9, 4, 0	[10] → 12	
[3] → 5, 2	[7] → 6, 8	[11] → 4, 12	

Determina las componentes fuertemente conectadas de G ***ejecutando el algoritmo estudiado en clase***:

1) Realizamos DFS de G, para calcular los tiempos de finalización, $u.f$, de cada vértice
2) Determinamos G^{T}
3) Realizamos DFS de G^{T}, pero en el ciclo principal consideramos los vértices en orden decreciente de $u.f$ calculado antes

2015-2-I2-P2–DFS, Componentes Fuertemente Conectados (2/2)

El DFS del ***paso 1*** *se puede realizar a partir de cualquier vértice de G; el resultado del paso 1, en términos de los tiempos u.f asignados a cada vértice, va a depender de cuál vértice partimos. Supongamos que partimos de* 0*, luego de* 6*, y finalmente de* 7:

```
0.d = 1
    5.d = 2
        4.d = 3
            3.d = 4
                2.d = 5
                2.f = 6
            3.f = 7
        4.f = 8
    5.f = 9
    1.d = 10
    1.f = 11
0.f = 12
```

```
6.d = 13
    9.d = 14
        11.d = 15
            12.d = 16
            12.f = 17
        11.f = 18
        10.d = 19
        10.f = 20
    9.f = 21
6.f = 22
7.d = 23
    8.d = 24
    8.f = 25
7.f = 26
```

El ***paso 2*** *consiste en transponer G, es decir, invertir la dirección de las aristas; vale tanto el dibujo del grafo, como la representación mediante listas de vértices adyacentes*:

[0] → 2, 6	[4] → 5, 6, 11	[8] → 7	[12] → 10, 11
[1] → 0,	[5] → 0, 3	[9] → 6, 8, 12	
[2] → 3, 4	[6] → 7	[10] → 9	
[3] → 2, 4	[7] → 8	[11] → 9	

En el ***paso 3*** *hacemos DFS del grafo transpuesto, pero en el ciclo principal de DFS consideramos los vértices en orden decreciente de u.f calculado en el paso 1, es decir, en este caso:*

partimos con el vértice 7 (7.f = 26)*, a partir del cual sólo podemos llegar al* 8*, de modo que estos dos vértices forman una componente;*

seguimos con el vértice 6 (6.f = 22)*, a partir del cual no podemos ir a ningún otro vértice (excepto* 7*, pero ya fue considerado), de modo que* 6 *forma una componente por sí solo;*

seguimos con 9 (9.f = 21)*, a partir del cual podemos llegar a* 12*,* 11 *y* 10 *(también a* 6*, pero ya lo consideramos), de modo que* 9*,* 10*,* 11 *y* 12 *forman otra componente;*

… similarmente encontramos la componente 0*,* 2*,* 3*,* 4 *y* 5*;*

… y por último, la componente formada sólo por 1.

2015-2-Ex-P1–Ordenación topológica (1/1)

IIC2133 Estructuras de Datos y Algoritmos

Examen

24 noviembre 2015

1. Considera un grafo direccional $G = (V, E)$ con costos y **acíclico** (y que, por lo tanto, puede ser ordenado topológicamente):

a) [**3 pts.**] Da un algoritmo de tiempo O(V+E) para encontrar las rutas más cortas en G desde un vértice de partida s.

1. *ordenar G topológicamente* —visto en clase; necesario para poder hacer un algoritmo eficiente
2. *para cada vértice v de G:* —este es el procedimiento *init*(s) visto en clase
   $d[v] = \infty$
   $\pi[v] = nil$
   $d[s] = 0$
3. *para cada vértice u de G,* ***en el orden producido por la ordenación topológica:*** —esta condición es esencial para la corrección del algoritmo
   para cada vértice v adyacente a u: —este es el procedimiento *reduce*(u, v) visto en clase
   if ($d[v] > d[u] + w(u, v)$)
   $d[v] = d[u] + w(u, v)$
   $\pi[v] = u$

b) [**3 pts.**] Justifica que tu algoritmo encuentra las rutas más cortas y que toma tiempo O(V+E).

[**1.5 pts.**] El algoritmo es de tiempo O(V+E), como se justifica a continuación:
- ordenar G topológicamente (paso 1) es O(V+E), ya que esencialmente es hacer un recorrido DFS
- el ciclo "*para*" que sigue (paso 2) es O(V), por lo que no agrega complejidad
- el ciclo "*para*" que sigue (paso 3) es O(V+E), ya que mira cada vértice u y, para cada uno, mira cada arista que sale de u (los vértices v adyacentes a u); y, en cada caso, hace una operación de complejidad constante (un *reduce*)

[**1.5 pts.**] El algoritmo efectivamente calcula las rutas más cortas desde s a todos los otros vértices (alcanzables desde s); en el paso 3:
- cada arista (u, v) es reducida exactamente una vez, cuando u es procesado (cuando la iteración "pasa" por u), dejando $d[v] \leq d[u] + w(u, v)$
- esta desigualdad se mantiene de ahí en adelante, hasta que el algoritmo termina, ya que $d[u]$ no cambia: como los vértices son procesados en orden topológico, ninguna arista que apunte a u va a ser reducida después de procesar u

2015-1-I2-P4–DFS, Componentes Fuertemente Conectados (1/2)

4. Considera el siguiente grafo no direccional, representado mediante sus listas de adyacencias:

[0]: 6 – 2 – 1 – 5	[1]: 0	[2]: 0	[3]: 5 – 4	[4]: 5 – 6 – 3
[5]: 3 – 4 – 0	[6]: 0 – 4	[7]: 8	[8]: 7	[9]: 11 – 10 – 12
[10]: 9	[11]: 9 – 12	[12]: 11 – 9		

Determina las **componentes conectadas** de este grafo, ejecutando el algoritmo basado en DFS estudiado en clase. En particular, muestra el orden en que se van produciendo cada una de las llamadas recursivas, si la primera llamada es DFS(0); marca los puntos de retorno de las llamadas en que se completa la detección de una componente conectada; y para cada componente conectada detectada lista sus vértices.

Respuesta: Próxima página

2015-1-I2-P4–DFS, Componentes Fuertemente Conectados (2/2)

dfs(0)
    dfs(6)
        0 √ –*ya lo había descubierto*
        dfs(4)
            dfs(5)
                dfs(3)
                    5 √
                    4 √
                3 **fin** –*terminé de descubrir todo lo que podía desde* 3
                4 √
                0 √
            5 **fin**
            6 √
            3 √
        4 **fin**
    6 **fin**
    dfs(2)
        0 √
    2 **fin**
    dfs(1)
        0 √
    1 **fin**
    5 √
0 **fin –> *aquí se completa una componente conectada, formada por* 0, 1, 2, 3, 4, 5, 6**
dfs(7)
    dfs(8)
        7 √
    8 **fin**
7 **fin –> *aquí se completa otra componente conectada:* 7, 8**
dfs(9)
    dfs(11)
        9 √
        dfs(12)
            11 √
            9 √
        12 **fin**
    11 **fin**
    dfs(10)
        9 √
    10 **fin**
    12 √
9 **fin –> *aquí se completa la última componente conectada:* 9, 10, 11, 12**

2015-1-I2-P5–BFS, Ordenación topológica (1/1)

5. Considera un grafo direccional acíclico. Dados dos vértices, v y w, de este grafo, describe un algoritmo para determinar el *primer ancestro común* de v y w. El **primer ancestro común** de v y w es un vértice que no tiene descendientes que también sean ancestros de v y w.

Una posibilidad: Sea G el DAG original. Primero, determinamos G^{T}, el grafo transpuesto de G: v y w tienen un ancestro común u en G si u es alcanzable (usando BFS) tanto desde v como desde w en G^{T}. Si u y u' son ancestros comunes de v y w en G, entonces el *primer* ancestro común será u si u' es alcanzable desde u en G^{T} (y viceversa). Más en general, si hay un conjunto de vértices que son ancestros comunes de v y w en G, entonces considera el subgrafo de G^{T} formado por esos vértices y por las aristas que los unen, y realiza una ordenación topológica de este subgrafo: la raíz resultante es el primer ancestro común.

2014-1-I2-P4–BFS (1/1)

Grafos.

4) Kevin Bacon es un actor que ha trabajado en muchas películas. En el mundo del cine, asignamos el *número Kevin Bacon* a un actor de la siguiente manera. El propio Kevin Bacon recibe un 0; un actor que ha trabajado en una película junto a Kevin Bacon recibe un 1; un actor que ha trabajado en una película junto a un actor cuyo número es 1, recibe un 2; etc. (Por supuesto, si un actor califica para dos o más números distintos, se le asigna el menor de esos números.) Describe y justifica un esquema eficiente basado en grafos no direccionales para determinar el número Kevin Bacon de un actor.

Respuesta:

Construimos un grafo en que los nodos son las películas y los actores (dos tipos distintos de nodos, pero todos nodos al fin), y las aristas unen las películas con los actores correspondientes; no hay aristas entre películas ni aristas entre actores.
A partir del nodo correspondiente al actor que nos interesa, ejecutamos BFS hasta llegar al nodo que corresponde a Kevin Bacon; el número Kevin Bacon del actor que nos interesa es un medio de la longitud del camino encontrado (en número de aristas).

2014-1-Ex-P1–Ordenación topológica (1/1)

Estructuras de Datos y Algoritmos – IIC2133

Examen

8 julio 2014

1) Supongamos que construimos un grafo direccional acíclico $G = (V, E)$ de la siguiente manera. Los vértices representan tareas que deben ser realizadas, y las aristas representan restricciones de orden entre tareas: una arista (u, v) indica que la tarea u debe realizarse antes que la tarea v. Además, asignamos a *cada vértice* un costo, que representa las unidades de tiempo necesarias para realizar esa tarea.

Una ruta en este grafo representa una secuencia de tareas que deben ser realizadas en un orden particular. Una ***ruta crítica*** es una ruta *más larga*, y corresponde al mayor tiempo necesario para realizar una secuencia ordenada de tareas. El costo de la ruta crítica es una cota inferior para el tiempo total necesario para realizar todas las tareas (típicamente, todas las tareas correspondientes a un mismo proyecto).

Da un algoritmo eficiente para encontrar una ruta crítica en G; en particular,

a) ¿Cómo se puede determinar *eficientemente* las rutas *más cortas* desde un vértice s a todos los demás en el grafo direccional acíclico? ¿Cuál es la complejidad de este algoritmo?

En [Cormen et al., 2009], se demuestra que a partir de la ordenación topológica de los vértices de un grafo direccional acíclico (como vimos en clase), se puede determinar eficientemente las rutas *más cortas* desde un vértice s a todos los demás: después de la ordenación topológica, inicializamos el grafo (con *Init*), y luego, recorriendo cada vértice en orden topológico, reducimos (con *Reduce*) una vez cada arista que sale del vértice. Esto tiene sentido, ya que, si recorro los vértices en orden topológico, una vez que llego al vértice v, he reducido todas las aristas que llegan a él y está garantizado al recorrer los restantes vértices que no voy a volver a v (el grafo es acíclico). Este algoritmo toma tiempo $O(V+E)$. *Init* y *Reduce* son tales que, una vez terminada la ejecución del algoritmo, es posible reconstruir fácilmente cada ruta más corta.

b) ¿Cómo se puede convertir el grafo G en otro, G', en que los costos estén en las aristas? Da un algoritmo de tiempo $O(V+E)$, basado en convertir cada vértice en un trío < *vértice'*, *arista*, *vértice''* >.

Asignamos costo 0 a cada arista original; y convertimos cada vértice original en un trío < *vértice'*, *arista*, *vértice"* >, en que *vértice'* recibe las mismas aristas que el vértice original, asignamos a la *arista* el costo del vértice original, y desde *vértice"* salen las mismas aristas que del vértice original.

c) ¿Cómo se pueden determinar eficientemente las rutas *más largas* en G'? ¿Cuál es la complejidad de tu algoritmo?

Las rutas *más largas* se pueden determinar similarmente a las rutas más cortas (sólo en grafos acíclicos), negando primero los costos de cada arista, o bien reemplazando primero ∞ por $-\infty$ en *Init* y los ">" por "<" en *Reduce*.

2013-2-I3-P2–DFS (1/1)

2) Considera un grafo no direccional. Queremos pintar los vértices del grafo, ya sea de azul o de amarillo, pero de modo que ***dos vértices conectados por una misma arista queden pintados de colores distintos***. Describe un algoritmo eficiente que pinte los vértices del grafo según la regla anterior, o bien que se dé cuenta de que no se puede; justifica la corrección de su algoritmo.

Respuesta:

1) La idea es usar DFS, que es O(V+E).

2) Recordemos que al aplicar DFS a un grafo no direccional sólo aparecen aristas *de árbol* y *hacia atrás*; no hay aristas *hacia adelante* ni *cruzadas*.

3) Por lo tanto, en el árbol DFS resultante, si a partir de un vértice v surgen varias ramas, significa que en el grafo original cualquier ruta que va de los vértices en una de esas ramas a los vértices en otra necesariamente pasa por v.

4) Por lo tanto, a partir de la raíz del árbol DFS podemos pintar los vértices alternando colores a medida que bajamos por cada rama.

5) Finalmente, hay que revisar las aristas hacia atrás del árbol DFS, cuya presencia refleja la existencia de ciclos en el grafo original:
   - Si alguna arista hacia atrás conecta (directamente) vértices pintados del mismo color, entonces el grafo no se puede pintar como se pide en el enunciado.
   - Pero si el árbol DFS no tiene aristas hacia atrás, o si ninguna arista hacia atrás conecta vértices pintados del mismo color, entonces es posible pintar los vértices del grafo original como se pide en el enunciado; la asignación de colores definida en el paso 4) es una forma concreta de hacerlo.

2013-2-Ex-P4–DFS modificado (1/1)

4) Considera un grafo no direccional conectado G. Un *puente* es una arista que, si se saca, separaría G en dos subgrafos disjuntos.

a) [**3 pts.**] Describe un algoritmo eficiente para encontrar todos los puentes de un grafo no direccional.

Una arista (u, v) es un puente ⇔ (u, v) no pertenece a un ciclo (simple del grafo); DFS encuentra los ciclos en un grafo.

Luego, ejecutemos DFS sobre el grafo, y miremos el bosque DFS producido: una arista de árbol (u, v) en este bosque es un puente ⇔ no hay aristas hacia atrás que conecten un descendiente de v a un ancestro de u.

¿Cómo determinamos esto? Recordemos el número —la marca de tiempo— v.d que DFS asigna a cada vértice v : una arista hacia atrás (x, y) conectará un descendiente x de v a un ancestro y de u si y.d < v.d.

Luego, si v.d es menor que todos los números d que pueden ser alcanzados mediante una arista hacia atrás desde cualquier descendiente de v, entonces (u, v) es un puente.

Es decir, basta modificar un poco dfsVisit visto en clase: asignar a cada vértice v un tercer número v.low que sea el menor de todos y.d alcanzables desde v mediante una secuencia de cero o más aristas de árbol seguida de una arista hacia atrás; si al retornar dfsVisit(v) v.d = v.low, entonces (u, v) es un puente.

b) [**2 pts.**] Justifica que tu algoritmo efectivamente encuentra todos los puentes.

Hay que demostrar la premisa de a): Una arista (u, v) es un puente ⇔ (u, v) no pertenece a un ciclo (simple del grafo). La demostración "sale" de la definición de "puente":

=> Si (u,v) es un puente, entonces por definición no puede formar parte de un ciclo

<= Si (u,v) no pertenece a un ciclo, entonces no hay otra manera de ir de u a v que no sea a través de (u,v); por lo tanto, si sacamos (u,v), u y v quedan desconectados entre ellos; y, por lo tanto, todos los vértices alcanzables desde u quedan desconectados de todos los vértices alzanzables desde v; es decir, el grafo queda desconectado

c) [**1 pt.**] Determina la complejidad de tu algoritmo.

Lo que hicimos fue modificar un poco dfsVisit: asignamos a cada vértice v un tercer número v.low que es el menor de todos y.d alcanzables desde v mediante una secuencia de cero o más aristas de árbol seguida de una arista hacia atrás; si al retornar dfsVisit(v) v.d = v.low, entonces (u, v) es un puente. Esta modificación no cambia la complejidad de dfsVisit.

2013-1-I3-P2–Ordenación topológica (1/1)

2) Considera el siguiente algoritmo de ordenación topológica un grafo direccional acíclico (DAG):

> Identifica un vértice *fuente* (un vértice al que no "llega" ninguna arista), asígnale el número 0, sácalo del grafo (junto a las aristas que salen de él). Repite este proceso para el grafo resultante, pero ahora usa el número 1; luego, el 2; y así sucesivamente.

Sugiere las estructuras de datos necesarias y su funcionamiento para implementar este algoritmo eficientemente. En particular:

a) Justifica que todo DAG tiene al menos una fuente.

Resp. Parémonos en un vértice cualquiera, v_0, y recorramos "hacia atrás" una arista que llega a él; como el grafo es acíclico, el vértice al que llegamos, v_1, es distinto de v_0. Hagamos lo mismo en v_1; vamos a llegar a otro vértice, v_2, que, por la misma razón, debe ser distinto de v_0 y v_1. Sigamos así hasta tener una secuencia de $n = |V|$ vértices distintos (suponiendo que en los n–1 primeros vértices de la secuencia siempre encontramos al menos una arista que llega al vértice). Si en este punto el vértice actual, v_n, tuviera una arista que llegara a él, ésta sólo podría venir de alguno de los vértices ya visitados, y por lo tanto formaría un ciclo; como el grafo es acíclico, este último vértice no puede tener ninguna arista que llegue a él.

b) ¿Cómo identificamos los vértices fuente la primera vez?

Resp. Supongamos que el grafo está representado por sus listas de adyacencias. Definamos un arreglo x, indexado según los vértices del grafo, que va a contar el número de aristas que llegan a cada vértice; inicializamos todos los elementos de x están en 0. Ahora recorremos las listas de adyacencias una a una; cada vez que llegamos a un vértice v, incrementamos el contador $x[v]$. Al terminar de recorrer las listas de adyacencias, los elementos de x que estén en 0 corresponden a los vértices fuente iniciales. Este procedimiento toma tiempo $O(V+E)$.

c) ¿Dónde guardamos los vértices fuente que vamos identificando a lo largo de la ejecución del algoritmo?

Resp. Cada vez que procesamos un vértice fuente (le asignams el número 0, 1, 2, …, lo sacamos del grafo, y también sacamos las aristas que salen de él), producimos nuevos vértices fuente: elementos de x que quedan en 0 (ver d). ¿Cómo hacemos para identificarlos y procesarlos? Una posibilidad es que cada vez que vamos a buscar un nuevo vértice fuente, recorramos todo el arreglo x buscando elementos que estén en 0 y que no sean vértices fuente procesados anteriormente. Esto *no* es muy eficiente.

Es mejor usar una cola (o un *stack*) de vértices fuente aún no procesados. Inicializamos la cola con los vértices fuente identificados en b). Para procesar el próximo vértice fuente, simplemente sacamos el primero de la cola. Durante el procesamiento de un vértice fuente, cuando producimos uno nuevo, ponemos éste en la cola.

d) ¿Cómo identificamos los (nuevos) vértices fuente en el grafo que queda después de que sacamos un vértice fuente?

Resp. Sacamos un vértice fuente de la cola y recorremos su lista de adyacencias: para cada vértice v que encontramos en esta lista, decrementamos $x[v]$; si $x[v]$ queda en 0, agregamos v a la cola. Este procedimiento toma tiempo $O(V+E)$.

2012-1-I3-P1–DFS, BFS (1/3)

IIC2133 - Estructuras de datos y algoritmos I-2012 – Interrogación 3

Estructuras de datos y algoritmos I-2012

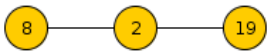
Interrogación n° 3

Instrucciones generales

- Ingresa tu nombre en todas las hojas de respuesta
- Entrega 1 hoja por pregunta, independiente si no respondiste la pregunta
- Lee cuidadosamente cada pregunta
- Dispones de 120 minutos

Pregunta 1 – Grafos (20 pts)

a) (8 pts) Escribe una función que permita ir de un vértice A a un vértice B pasando por el menor número de vértices como intermediarios. Tu función debe recibir como parámetros el grafo, el vértice A y el vértice B – de acuerdo a lo visto en clases, puedes elegir como deseas recibir el grafo y los vértices. Al principio, tu función debe imprimir “Comenzando en el vértice i” donde i es el identificador del vértice A. Luego, cada vez que tu función visite un vértice camino al vértice B, imprimir el identificador del vértice que está visitando. Finalmente, cuando llega al vértice B, imprimir “Llegando al vértice j”, donde j es el identificador del vértice B.

Por ejemplo, en el grafo: 

La función, al ir del vértice 8 al vértice 19 debiera imprimir:

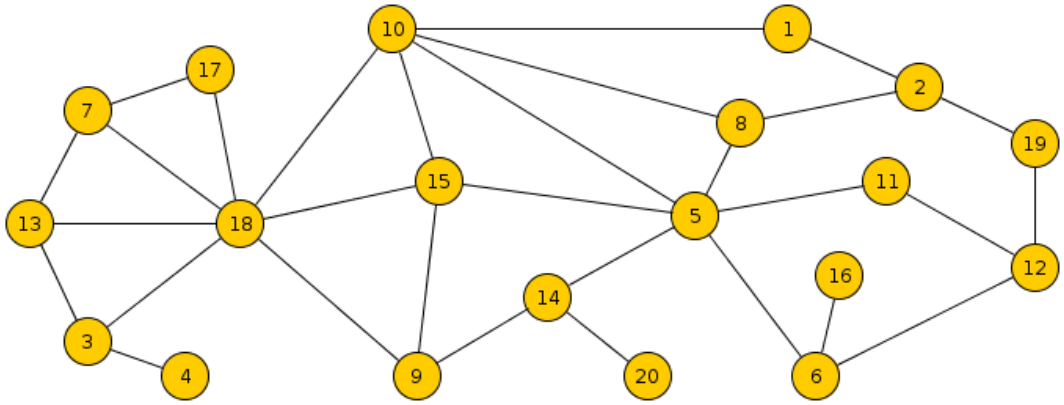
```
Comenzando en el vértice 8
2
Llegando al vértice 19
```

Este ejercicio se puede resolver aplicando BFS para encontrar el camino más corto entre A y B. Sin embargo, esto tiene una complicación adicional y es que no basta con recorrer el grafo usando BFS, sino que además hay que guardar el camino (lo que se puede conseguir guardando para cada vértice el vértice padre del árbol resultante). Si se utiliza una matriz de adyacencia, y se considera que un vértice i NO está conectado al vértice j si y sólo si matriz[i][j] == 0 (o, dicho de otro modo, matriz[i][j] != 0 significa que hay una arista ij), es posible guardar en la misma matriz los ids de los vértices padres del camino BFS, cambiando el valor [i][j] (que debiera ser 1) por -1 si en el recorrido BFS se llegó al vértice j desde i. Luego, para imprimir el camino es cuestión de comenzar del vértice A y seguir al vértice j tal que matriz[A][j] == -1, y así hasta llegar a matriz[k][B] con misma condición.

b) (6 pts) Escribe un algoritmo que recorra en profundidad (DFS) un grafo. Luego, considera el siguiente grafo y utilizando tu algoritmo, recórrelo desde el vértice 1. Escribe la secuencia de nodos que visitas.

2012-1-I3-P1–DFS, BFS (2/3)

IIC2133 - Estructuras de datos y algoritmos I-2012 – Interrogación 3



Este ejercicio consiste en efectuar un DFS clásico y aplicarlo al grafo. Sólo para cambiar un poco, voy a usar una lista de adyacencia, representada como una lista de listas.

(dado que podía ser en seudo código, me voy a dar el gusto de escribir mi solución en Java :D)

```
void dfs(List<List<Integer>> graph, int source) {
        boolean[] visited = new boolean[graph.size()];
        // en Java los boolean se inicializan en false por defecto
        rdfs(graph, visited, source);
}
void rdfs(List<List<Integer>> graph, boolean[] visited, int source) {
        // he visitado source!
        visited[source] = true;
        List<Integer> adjVertices = graph.get(source);
        for(int adj : adjVertices) {
                // adj es un vertice adyacente
                if (!visited[adj]) {
                        // aún no he visitado adj. A por ello.
                        rdfs(graph, visited, adj);
                }
                // si hubo algun vertice adyacente que falto por visitar, el for se encargará.
        }
}
```

2012-1-I3-P1–DFS, BFS (3/3)

IIC2133 - Estructuras de datos y algoritmos I-2012 – Interrogación 3

Ahora, la ejecución de mi dfs dependerá del orden en que estén los vértices en la lista. Supondré que la lista de nodos adyacentes siempre está ordenada del nodo menor al mayor.

Así, la ejecución será:

1 2 8 5 6 12 11 19 16 10 15 9 14 20 18 3 4 13 7 17

c) (6 pts) Imagina que debes escribir una función simple, que reciba un grafo, dos vértices y retorne true o false si están directamente conectados (si hay una arista entre los vértices). Sin embargo, considera que el grafo tendrá entre 50.000 y 100.000 vértices y cada vértice tentrá como máximo un grado de 4. ¿Cómo representarías el grafo para resolver este problema y por qué? También debes escribir la función.

Acá lo importante que debes notar es que:

- *No puedes usar una matriz de adyacencia porque, si suponemos que un vértice ocupa n bytes, la matriz ocuparía entre 2.500.000.000n y 10.000.000.000n bytes (si suponemos que un vértice es un int de 32 bits, es decir, ocupa 4 bytes, entonces la matriz usa entre 10GB y 40GB de memoria)*
- *Si hay hasta 100.000 vértices de hasta grado 4, entonces habrá hasta 200.000 aristas, por lo que una matriz de incidencia tendría hasta 100.000 x 200.000 datos, lo que es aún menos viable que la matriz de adyacencia.*

Por lo tanto, por el número de vértices hay que usar o bien una lista de incidencia (que tendrá hasta 200.000 x 2 datos, es decir, hasta 400.000 datos, lo cual es un número bien manejable: 1.6 MB de memoria con int de 32 bits), o bien una lista de adyacencia (que tendrá hasta 100.000 x 4 datos, es decir, también hasta 400.000 datos).

Dado que se desea saber si dos vértices están conectados, si usamos la lista de incidencia cada vez habría que buscar entre todas las aristas, mientras que con la lista de adyacencia la lista será, en el peor de los casos, buscar entre 4 elementos. Por lo tanto representaría el grafo con una lista de adyacencias.

```
boolean isConnected(int[][] adjList, int i, int j) {
        for (int k = 0; k < adjList[i].length; k++) {
                if (adjList[i][k]  == j) return true;
        }
        return false;
}
```

2011-2-I2-P3–Ordenación topológica (1/1)

3. Considera el problema de ordenar topológicamente un grafo direccional acíclico (DAG).

a) [**2 pts.**] Justifica que todo DAG tiene al menos una ***fuente*** (un vértice al que no "llega" ninguna arista) y al menos un ***sumidero*** (un vértice del que no "sale" ninguna arista).

Respuesta: Demostramos la existencia del sumidero. Parémonos en un vértice cualquiera, v_0, y recorramos una arista que sale de él; como el grafo es acíclico, el vértice al que llegamos, v_1, es distinto de v_0. Hagamos lo mismo en v_1; vamos a llegar a otro vértice, v_2, que, por la misma razón, debe ser distinto de v_0 y v_1. Sigamos así hasta tener una secuencia de $n = |V|$ vértices distintos (suponiendo que en los n-1 primeros vértices de la secuencia siempre encontramos al menos una arista que salía). Si en este punto el vértice actual, v_n, tuviera una arista que sale, ésta sólo podría ir a alguno de los vértices ya visitados, formando un ciclo; como el grafo es acíclico, este último vértice no puede tener ninguna arista que salga.

La demostración de la existencia de la fuente es análoga (recorremos las aristas "hacia atrás").

b) [**Se puede resolver sin haber hecho la demostración pedida en a)**] Considera el siguiente algoritmo de ordenación topológica: Identifica un vértice fuente, asígnale el número 0, sácalo del grafo (junto a las aristas que salen de él). Repite este proceso para el grafo resultante, pero ahora usa el número 1; luego, el 2; y así sucesivamente.

Sugiere las estructuras de datos necesarias y su funcionamiento para implementar este algoritmo eficientemente. En particular:

i) [**1 pt.**] ¿Cómo identificamos los vértices fuente la primera vez?

Respuesta: Supongamos que el garfo está representado por sus listas de adyacencias. Sea x un arreglo que cuenta el número de aristas que llegan a cada vértice; inicializamos los elementos de x en 0. Ahora recorremos las listas de adyacencias una a una; cada vez que llegamos a un vértice v, aumentamos el contador $x[v]$. Al terminar de recorrer las listas de adyacencias, los elementos de x que estén en 0 corresponden a los vértices fuente iniciales. Este procedimiento toma tiempo $O(V+E)$.

ii) [**1 pt.**] ¿Dónde guardamos los vértices fuente que vamos identificando a lo largo de la ejecución del algoritmo?

Respuesta: Cada vez que procesamos un vértice fuente (esto es, le asignamos el número 0, 1, 2, …, lo sacamos del grafo, y también sacamos las aristas que salen de él), producimos nuevos vértices fuente: elementos de x que quedan en 0 (ver **iii**). ¿Cómo hacemos para identificarlos y procesarlos? Una posibilidad es que cada vez que vamos a identificar (y procesar) un nuevo vértice fuente recorramos todo el arreglo x buscando elementos que estén en 0 pero que no sean vértices fuente procesados anteriormente. Esto no es muy eficiente.

Es mejor usar una cola (o un *stack*) de vértices fuente aún no procesados. Inicializamos la cola con los vértices fuente identificados en **i**). Para procesar el próximo vértice fuente, simplemente sacamos el primero de la cola. Durante el procesamiento de un vértice fuente, cuando producimos uno nuevo, ponemos éste en la cola.

iii) [**2 pts.**] ¿Cómo identificamos los (nuevos) vértices fuente en el grafo que queda después de que sacamos un vértice fuente?

Respuesta: Sacamos un vértice fuente de la cola y recorremos su lista de adyacencias: para cada vértice v en esta lista, reducimos $x[v]$ en uno; si $x[v]$ queda en 0, agregamos v a la cola. Este procedimiento toma tiempo $O(V+E)$.

2011-2-Ex-P4–Creación de algoritmo, ordenación topológica (1/1)

4. Supón que construimos un grafo direccional acíclico de la siguiente manera. Los vértices representan tareas que deben ser realizadas, y las aristas representan restricciones de orden entre tareas: una arista (u, v) indica que la tarea u debe realizarse antes que la tarea v. Además, asignamos a cada vértice un costo, que representa las unidades de tiempo necesarias para realizar la tarea correspondiente.

Una ruta en este grafo representa una secuencia de tareas que deben ser realizadas en un orden particular. Una ***ruta crítica*** es una ruta *más larga*, y corresponde al mayor tiempo necesario para realizar una secuencia ordenada de tareas. El costo de la ruta crítica es una cota inferior para el tiempo total necesario para realizar todas las tareas.

Da un algoritmo eficiente para encontrar una ruta crítica en un grafo direccional acíclico; ¿cuál es la complejidad de tu algoritmo?

En [Cormen et al., 1990] y en [Cormen et al., 2001], se demuestra que a partir de la ordenación topológica de los vértices de un grafo direccional acíclico (como vimos en clase), se puede determinar eficientemente las rutas más cortas desde un vértice s a todos los demás: después de la ordenación topológica, inicializamos el grafo (con *Init*), y luego, recorriendo cada vértice en orden topológico, reducimos (con *Reduce*) una vez cada arista que sale del vértice. Esto tiene sentido, ya que, si recorro los vértices en orden topológico, una vez que llego al vértice v, he reducido todas las aristas que llegan a él y está garantizado al recorrer los restantes vértices que no voy a volver a v (el grafo es acíclico). Este algoritmo toma tiempo $O(V+E)$. *Init* y *Reduce* son tales que, una vez terminada la ejecución del algoritmo, es posible reconstruir fácilmente cada ruta más corta.
Las rutas más largas se pueden determinar similarmente (sólo en grafos acíclicos), negando primero los costos de cada arista, o bien reemplazando primero ∞ por $-\infty$ en *Init* y los "$>$" por "$<$" en *Reduce*.
El único tema pendiente es que antes que todo lo anterior, hay que convertir el grafo descrito más arriba en uno en que los costos estén en las aristas. Esto puede hacerse fácilmente en tiempo $O(V+E)$: asignamos costo 0 a cada arista original; y convertimos cada nodo original en un trío <nodo', arista, nodo">, en que nodo' recibe las mismas aristas que el nodo original, asignamos a la arista el costo del nodo original, y desde nodo" salen las mismas aristas que del nodo original.
El algoritmo pedido ejecuta los pasos anteriores en orden inverso. Primero, convierte el grafo original a uno con los costos en las aristas [**2 pts.**]; luego, niega los costos de las aristas, o cambia ∞ por $-\infty$ y los "$>$" por "$<$" [**2 pts.**]; y, finalmente, ejecuta la ordenación topológica, seguida por la inicialización y la serie de reducciones [**2 pts.**].

2010-2-I2-P1–Componentes Fuertemente Conectados (1/1)

Estructuras de Datos y Algoritmos – IIC2133
I2
18 octubre 2010

1. Dado un grafo direccional G, explica cómo construir en tiempo $O(V + E)$ otro grafo direccional, G', que tenga las mismas componentes fuertemente conectadas y el mismo grafo de componentes de G, pero que tenga el mínimo número posible de aristas; justifica tu respuesta.

Respuesta:

[**1 pt.**] Observación: Las SCC's son conjuntos de vértices; luego, para que G' tenga las mismas SCC's que G, G' tiene que tener los mismos vértices que G, pero no necesariamente las mismas aristas.

[**2.5 pts.**]
[**2 pts.**] Para cada SCC de G, en G' formamos un ciclo simple con los mismos vértices; así, si la SCC en G tiene k vértices, en G' habrá un ciclo con k aristas (el menor número posible de aristas de una SCC de k vértices).
[**0.5 pts.**] Identificar las SCC's de G toma tiempo $O(V + E)$; y agregar las aristas a G' para formar ciclos simples para todas las SCC's toma tiempo $O(V)$.

[**2.5 pts.**]
[**0.5 pts.**] Teniendo las SCC's de G, se puede formar el grafo de componentes de G en tiempo $O(V + E)$.
[**2 pts.**] Finalmente, para que G' tenga el mismo grafo de componentes que G, hay que hacer lo siguiente: por cada arista del grafo de componentes de G (aristas que unen pares de SCC's), en G' hay que agregar una arista que una un vértice (cualquiera) de la primera SCC con un vértice (cualquiera) de la segunda SCC. Esto toma tiempo $O(E)$.

2010-2-Ex-P5–Verificación Dijkstra, ordenación topológica (1/1)

5) Con respecto a las rutas más cortas en grafos direccionales:

a) Un compañero de curso te dice que implementó el algoritmo de Dijkstra. El programa produce las distancias d y los padres π para cada vértice del grafo. Da un algoritmo de tiempo $O(V+E)$ para verificar el resultado del programa de tu compañero. Supón que todas las aristas tienen costos no negativos. En particular, ¿qué debes verificar con respecto al vértice de parida s?; ¿qué debes verificar con respecto a los otros vértices?; y ¿cómo te puedes asegurar que el programa de tu compañero efectivamente encontró las rutas más cortas desde s?

b) Da un algoritmo de tiempo $O(V+E)$ para encontrar las rutas más cortas desde un vértice de partida s, si el grafo es acíclico (y, por lo tanto, puede ser ordenado topológicamente).

a)

Para s, hay que verificar $s.d = 0$ y $s.\pi = NIL$.
Para los otros vértices v, hay que verificar $v.d = v.\pi.d + w(v.\pi, v)$; o bien $v.d = \infty$ si y solo si $v.\pi = NIL$.
Si cualquiera de estas verificaciones falla, entonces el resultado del programa del compañero es incorrecto.
De lo contrario, todavía hay que asegurarse que haya encontrado las rutas más cortas desde s. Para esto, basta con reducir cada arista una vez: si algún $v.d$ cambia, entonces el resultado es incorrecto; de lo contrario, es correcto.

b)

```
Ordenar el grafo topológicamente
para cada vértice v del grafo {
    v.d = ∞
    v.π = NIL
}
s.d = 0
para cada vértice u del grafo, en el orden producido por la ordenación topológica
    para cada vértice v adyacente a u
        if ( v.d > u.d + w(u, v) ) {
            v.d = u.d + w(u, v)
            v.π = u
        }
```

5 Backtracking

En esta sección se encuentran los siguientes contenidos:

- Modelamiento de un problema
- Podas
- Heurísticas

2020-2-I2-P1–Coloración de grafos, backtracking (1/3)



Pontificia Universidad Católica de Chile
Escuela de Ingeniería
Departamento de Ciencia de la Computación

IIC2133 — Estructuras de Datos y Algoritmos
2020 - 2

Interrogación 2

Pregunta 1

Un grafo no dirigido $G(V, E)$ se dice **k**-coloreable ($\mathbf{k} \in \mathbb{N}$), si existe una manera de asignar a cada vértice $v \in V$, un color $c \in \{1, ..., k\}$ tal que, para todo $(u, v) \in E$ se cumple que $u.color \neq v.color$.

Considera el siguiente algoritmo que determina si un grafo es **k**-coloreable, considerando que inicialmente $v.color = 0$ para todo vértice $v \in V$, y que α son las listas de adyacencia del grafo.

```
is_k_coloreable(V, α, k):
    if V = ∅:
        return true
    Sea v un vértice cualquiera de V
    for c ← 1..k:
        valid ← true
        for each u ∈ α[v]:
            if u.color = c:
                valid ← false
                break
        if valid:
            v.color ← c
            if is_k_coloreable(V − {v}, α, k):
                return true
            v.color ← 0
    return false
```

Llamamos coloración parcial al subconjunto todos los vértices que tienen asignado un color.

Luego de cada asignación, el algoritmo ha generado una nueva coloración parcial. Justifica por qué el algoritmo nunca va a generar dos veces una misma coloración parcial con los mismos colores.

2020-2-I2-P1–Coloración de grafos, backtracking (2/3)

Solución Pregunta 1)

Opción 1: Árbol de asignaciones

El algoritmo propuesto es backtracking, ya que busca todas las asignaciones válidas posibles de cada vértice, para asignarles un color de forma recursiva.

La estructura del árbol de asignaciones que genera este algoritmo a partir de los vértices es la siguiente (se consideró una versión simplificada donde se muestran todas las opciones posibles siendo que el algoritmo recorta las que incumplen la condición de **k**-coloreable, por ejemplo, root $\rightarrow 1 \rightarrow 1$ será podado por el algoritmo por tener el mismo color en vértices vecinos)

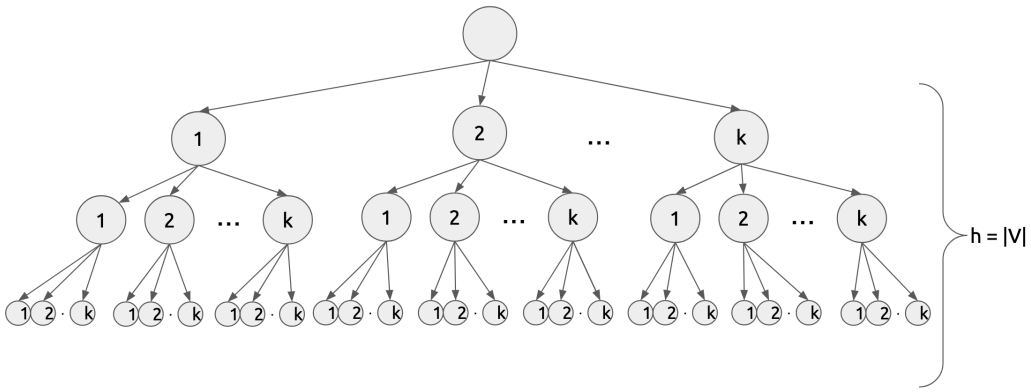


Figura 1: Árbol de asignaciones

Las coloraciones parciales en cada iteración del algoritmo representan un camino por el árbol de asignaciones. Por lo tanto, para que hayan 2 asignaciones parciales iguales en cualquier iteración del algoritmo, esto significaría que existen dos caminos iguales en el árbol de asignaciones, lo que por construcción es imposible. Además, backtracking recorrerá cada camino una sola vez. Debido a estas dos condiciones, cada coloración parcial en cada iteración del algoritmo será diferente.

[**1 pt**] Por mencionar que es un algoritmo de backtracking.

[**2 pts**] Por explicar la estructura del árbol de asignaciones.

[**3 pts**] Por justificar que rutas distintas en el árbol son coloraciones parciales distintas.

Opción 2: Fundamentación por código o casos

Importante: Esta opción no está completa debido a que no se usaron las propiedades de backtracking esperadas o la generalización de que se cumplía para todos los casos no era correctamente justificada, por lo que el puntaje máximo a obtener en este caso es 4 puntos.

2020-2-I2-P1–Coloración de grafos, backtracking (3/3)

El algoritmo is_k_coloreable es un algoritmo de backtracking. Para que dos coloraciones sean iguales, necesariamente su conjunto de vértices debe ser igual. Esto es equivalente a mencionar que dos asignaciones sucesivas son siempre distintas. Para demostrar que dos asignaciones no sucesivas son siempre distintas, se argumenta a través de la lógica o funcionamiento del algoritmo. Por ejemplo, a través del `for` sobre los colores.

[**1 pt**] Por mencionar que es un algoritmo de backtracking.

[**1 pt**] Por justificar conjuntos de vértices distintos son coloraciones parciales distintas.

[**2/4 pt**] Por mostrar que dos asignaciones no sucesivas son coloraciones parciales distintas argumentando a través de la lógica o funcionamiento del algoritmo. Ojo: El puntaje es 2/4 por la incompletitud de esta respuesta mencionada anteriormente.

2019-1-C4-2–Modelamiento, podas, heurísticas (1/3)

2) Sea $G(V, E)$ un grafo no direccional y $C \subseteq V$ un subconjunto de sus vértices. Se dice que C es un *k-clique* si $|C| = k$ y todos vértices de C están conectados con todos los otros vértices de C.

Dado un grafo cualquiera $G(V, E)$ y un número k, queremos determinar si existe un *k-clique* dentro de G. Esto se puede resolver usando *backtracking*.

a) [2pts.] Describe la modelación requerida para aplicar *backtracking* a este problema: explica cuál es el conjunto de **variables**, cuáles son sus **dominios**, y describe en palabras cuáles son las **restricciones** sobre los valores que pueden tomar dichas variables.

2019-1-C4-2–Modelamiento, podas, heurísticas (2/3)

R: Cualquier descripción que describa correctamente la modelación y explique de manera clara las variables, los dominios de las variables y restricciones sobre los valores que pueden tomar dichas variables, de tal forma que se pueda resolver con backtracking, tendrá el puntaje correspondiente. Recordar que al ser no direccional, si (v, v') ϵ E → (v', v) ϵ E. A continuación, se plantean dos posibles formas de solucionar el problema:

Propuesta 1:

Las **variables** son nodos pertenecientes al k-clique. El **dominio** son los posibles vértices que puede ir asociado al nodo. Las **restricciones** son que cada vértice que se escoge del dominio tiene que estar conectado a todos los otros nodos que se han escogido anteriormente.

Ejemplo en pseudocódigo (para guía del alumno; no era necesario implementar):

```
X = nodos 1, …, k
D = V
R = {(v, v') ϵ E ∀ v' ϵ C}

is_solvable(X, D, C):
    si X = {} return True
    x ← alguna variable de X
    para v ϵ D:
        si v no cumple R, continuar
        si is_solvable(X - {x}, D - {v}, C ∪ {x}):
            retornar True
        C = C - {x}
    retornar False
```

Luego, podemos llamar a is_solvable({1, …, k}, D, {}).

Propuesta 2:

Las **variables** son los vértices de G. El **dominio** es binario: 1 si el vértice está presente en el k-clique y 0 si no. Las **restricciones** son que el vértice asignado como presente en el k-clique (con valor 1 en la variable) tiene que estar conectados a todos los otros vértices que han sido asignados como presente en el k-clique, y el número de vértices asignados como presentes en el grafo debe ser igual a k.

Ejemplo en pseudocódigo (para guía del alumno; no era necesario implementar):

```
X = V
D = {0, 1}
R = {(v, v') ϵ E ∀ v' ϵ C}

is_solvable(X, D, C, k):
    si k = 0 return True
    si X = {} return False
    x ← alguna variable de X
    para v ϵ D:
        si v = 1:
```

2019-1-C4-2–Modelamiento, podas, heurísticas (3/3)

```
                    si v no cumple R, continuar
                    si is_solvable(X - {x}, D, C ∪ {x} , k  - 1):
                            retornar True
                    C = C - {x}
            si v = 0:
                    si is_solvable(X - {x}, D, C, k):
                            retornar True

        retornar False
```

Luego, podemos llamar a is_solvable({V, D, {}, k).

- **[1 pt]** Explica el conjunto de variables y su dominio.
- **[1 pt]** Explica las restricciones sobre los valores que pueden tomar dichas variables.

b) [2 pts.] Propón una **poda** y explica la modelación a nivel de código requerida para implementarla eficientemente.

R: Si el alumno menciona una poda acorde a su modelación que sea efectiva, y explica cómo implementarla, tendrá el puntaje correspondiente. Algunas podas posibles:

- Si el vértice que estamos revisando tiene un número de aristas tiene menos de k - 1 aristas que se conectan con él, es imposible que éste pertenezca al k-clique. Para implementarla, podemos preprocesar el grafo, agregándole a cada nodo una variable que determina la cantidad de aristas que tiene hacia otros nodos (Que toma un tiempo O(|E|). Luego, basta acceder a esta variable y si es menor a k - 1, se detiene la ejecución.
- (Propuesta 2) Si quedan menos variables a asignar, que el valor de k, podemos terminar esa ejecución ya que es imposible agregar suficientes elementos a C para que pertenezcan al k-clique. Para esto, podemos llevar un contador de cuántas variables nos quedan por asignar, y cuántos elementos llevamos en nuestro k-clique. Si la cantidad de variables que nos queda por asignar es menor a (k - (N° elementos asignados que llevamos en k-clique)), detenemos la ejecución.

- **[1 pt] Propone** una poda que efectivamente sea útil para el problema a resolver
- **[1 pt] Explica** cómo implementarla eficientemente

c) [2 pts.] Propón una **heurística para el orden de las variables** y explica la modelación a nivel de código requerida para implementarla eficientemente.

R: Si el alumno menciona una heurística acorde a su modelación que sea efectiva, y explica cómo implementarla, tendrá el puntaje correspondiente. Algunas heurísticas posibles:

- (Propuesta 2) Podemos ordenar los vértices de mayor a menor en función de la cantidad de aristas que poseen. De esta manera, es más probable que éstos sean parte de algún k-clique. Para esto, podemos preprocesar el grafo, agregándole a cada nodo una variable que determina la cantidad de aristas que tiene hacia otros nodos (Que toma un tiempo O(|E|). Luego, basta con ordenar este arreglo de vértices en función de la cantidad de aristas hacia otros nodos, mediante algún algoritmo eficiente de los que se han visto en clases (como MergeSort, QuickSort, etc.)

2018-2-I2-P1–Modelamiento, podas, heurísticas (1/1)

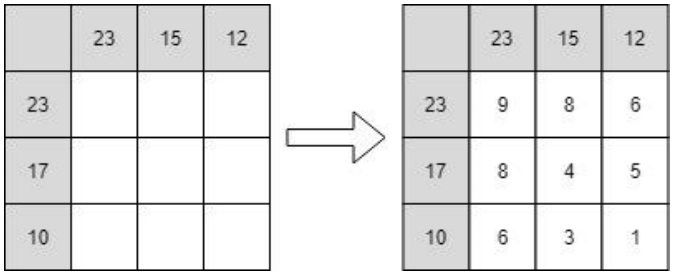
Estructuras de Datos y Algoritmos – IIC2133

I2

9 de octubre, 2018

1. Backtracking

Un tablero de *kakuro square* es una grilla de *NxM* donde cada posición tiene una celda que puede ser rellenada por un número natural distinto de 0. Además, cada fila y columna tiene un número natural que indica el valor que debe tener la suma de los números correspondientes a la fila o columna. P.ej.:



	23	15	12
23			
17			
10			

	23	15	12
23	9	8	6
17	8	4	5
10	6	3	1

El problema de rellenar la grilla cumpliendo con las restricciones se puede resolver utilizando la estrategia de backtracking.

a) Identifica las variables del problema y sus respectivos dominios.

b) Explica con tus palabras cómo se podría revisar en tiempo O(1) si la asignación de una variable rompe una restricción.

c) Explica con tus palabras una poda o una heurística que se pueda aplicar para resolver este problema más eficientemente.

Respuesta:

a) Para resolver con backtracking este problema se debe asignar a cada celda un número, por lo que las variables son las celdas [1pto].

En teoría cada celda puede tener cualquier número natural, pero ya que no puede haber un dominio infinito para resolverlo con backtracking es necesario acotar los dominios. Un domino acotado correcto puede ser los números del 1 al MAX(restriction). Ya que se sabe que nunca un número puede ser mayor al valor de su restricción [1pto].

b) Se puede mantener un contador por cada restricción del problema que cuente la suma actual de cada fila o columna. De esta manera se inmediatamente si rompo la restricción cuando el contador supera la restricción [2pts].

c) [2pts]
Podas: Ver que los contadores descritos en b) no superen el máximo antes de tener completa la fila o columna. Ver que la suma de una fila o columna actual más el número de casillas vacías de la fila o

2018-1-I2-P4–Implementación de Backtracking (1/1)

4. Dos grafos y se dicen *isomorfos* si existe una función biyectiva tal que si y sólo si . Escribe un algoritmo que determine si dos grafos son isomorfos. Considera que si dos grafos son isomorfos, al eliminar un vértice x de G_1 (y las aristas de x) y su vértice correspondiente (y aristas) en G_2, los grafos que quedan también son isomorfos.

Respuesta

F = diccionario inicialmente vacío

```
backtracking(Grafo1, Grafo2):
        // Caso base 1 pto
        If (grafo1.nodos = grafo2.nodos = vacio) return true
        // Asignar los nodos de un grafo los nodos del otro 2 pts
        n = grafo1.nodos.pop(0)
        For nodo in grafo2.nodos:
                F(n) = nodo
                // Restriccion 2 pts
                If  (cumple_restriccion(n, nodo))
                        // Hacer bien caso recursivo y UNDO 1pto
                        If  (backtracking(grafo1 sin n, grafo2 sin nodo) return true
                F(n) = null
        Return false

cumple_restricciones(nodo1, nodo2):
        If (|nodo1.vecinos| != |nodo2.vecinos|) return false
        For v in nodo1.vecinos:
                If (v esta en F):
                        If (F(v) esta en nodo2.vecinos):
                                Return false
        Return true
```

2017-2-I1-P1–Implementación de Backtracking (1/2)



Pontificia Universidad Católica de Chile
Escuela de Ingeniería
Departamento de Ciencia de la Computación

Pauta I1

IIC2133 — Estructuras de Datos y Algoritmos
Segundo semestre 2017

1 Pregunta 1

En el problema de la programación de máquinas, tenemos n tareas que realizar, cada una con su hora de inicio, s_i, y su hora de finalización, f_i, y tenemos un número suficientemente grande de máquinas para realizar las tareas; y lo que queremos es poder realizar todas las tareas ocupando el menor número posible de máquinas, respetando la restricción de que una máquina solo puede realizar una tarea a la vez (por lo tanto, dos tareas se pueden realizar en una misma máquina solo si la hora de finalización de una es menor o igual que la hora de inicio de la otra). Escribe un algoritmo de **backtracking** para resolver este problema.

Evaluación

Cabe destacar que backtracking por si solo no garantiza que la solución encontrada sea mínima en la cantidad de máquinas, por lo que los que no consideraran eso recibían solo **la mitad** del puntaje.

Se pensaron 3 formas sencillas de asegurar optimalidad:

- Limitar la cantidad de máquinas. Si no se encontraba una asignación, entonces se aumentaba el límite.
- Revisar todas las posibles asignaciones, y recordar siempre cual era la mejor.
- Ordenar las tareas por hora de inicio: luego de eso, la primera asignación que se encuentre es óptima.

Por otro lado, si el algoritmo propuesto no es de Backtracking, también se asignaba solo **la mitad** del puntaje.

En definitiva, se asignaron **3pts** a que la solución fuera de Backtracking, y otros **3pts** a que garantizara optimalidad.

Solución (recordando la mejor)

Dado un conjunto de tareas $T = \{t_1, \cdots, t_n\}$, y un conjunto de máquinas $M = \{m_1, \cdots, m_n\}$, se busca la mejor asignación S de tareas a esas máquinas, la cual usa sólo las primeras k máquinas.

Notar que en este algoritmo, tanto S como k son por referencia.

2017-2-I1-P1–Implementación de Backtracking (2/2)

```
BuscarAsignación(T, M, S, k)
    if todas las tareas en T han sido asignadas then
    |  return True
    end
    t ← una tarea sin asignar en T
    foreach máquina m ∈ M do
    |  if t no tiene tope de horario con ninguna de las tareas en m then
    |  |  Se le asigna t a m
    |  |  if BuscarAsignación(T, M, S, k) then
    |  |  |  m ← cantidad de máquinas que usa la asignación actual
    |  |  |  if m < k then
    |  |  |  |  S ← asignación actual
    |  |  |  |  k ← m
    |  |  |  end
    |  |  end
    |  |  Se des-asigna t de m
    |  end
    end
    return False
```

Para encontrar la asignación óptima, primero se setea S como la asignación trivial: cada tarea a una máquina distinta, por lo que $k = n$.

Luego se llama:

BuscarAsignación(T, M, S, k)

Al finalizar, S contendrá la asignación óptima de tareas, usando k máquinas.

2017-1-Ex-P2–Implementación de Backtracking (1/1)

2. Dadas una secuencia de números enteros positivos no necesariamente distintos $\mathbf{x} = x_1, \ldots, x_n$ y una secuencia de bits $\mathbf{b} = b_1, \ldots, b_n$, se define la función:

$$\mathbf{x} \otimes \mathbf{b} = \sum_{i=1}^{n} (-1)^{b_i} x_i$$

a) Dé el pseudocódigo de una función que utilice backtracking y que reciba una secuencia de números $\mathbf{x}$ y un número N, y retorne *true* cuando existe un $\mathbf{b}$ tal que $\mathbf{x} \otimes \mathbf{b} = N$, y que retorne *false* en caso contrario.

Respuesta:
```
1 function f(i, s)
```

b) Dé un pseudocódigo iterativo para la función de a) que sea **mucho más** eficiente que su implementación de b). Analice su tiempo de ejecución.

2016-2-I1-P1–Implementación de Backtracking (1/2)

Estructuras de Datos y Algoritmos – IIC2133

I1

31 agosto 2016

Nota: *Cuando se pide que des o describas un* ***algoritmo****, se espera algo parecido a lo siguiente*:

```
selectionSort(a):
    for k = 0 … n-1:
        min = k
        for j = k+1 … n:
            if a[j].key < a[min].key:
                min = j
        exchange(a[k], a[min])
```

1. Dado un mapa, queremos colorear las regiones de manera tal que nunca dos regiones adyacentes (limítrofes) tengan el mismo color, usando a lo más cuatro colores. Para esto, conviene representar el mapa como un grafo, en que los nodos corresponden a regiones y hay una arista entre dos nodos si y sólo si las dos regiones correspondientes son adyacentes. Generalizando, el problema de colorear un grafo consiste en determinar todas las formas diferentes en que un grafo dado puede ser coloreado usando a lo más m colores. Escribe un algoritmo de ***backtracking*** para resolver este problema.

Suponiendo que el grafo tiene n nodos, representamos los nodos simplemente por los números 1, 2, …, n; y representamos las aristas por una matriz G tal que $G[i][j] = 1$ si y sólo si hay una arista entre los nodos i y j, y $G[i][j] = 0$ en otro caso. Además, representamos los m colores por los números 1, 2, …, m, y las soluciones por las tuplas $(x_1, \ldots, x_n)$, en que x_i es el color del nodo i.

Se sugiere escribir un ciclo principal, en que asignan todos los colores válidos para el nodo k; cada vez que se asigna un nuevo color al nodo k, este ciclo principal se llama recursivamente para asignar todos los colores válidos para el nodo k+1. Y se sugiere escribir otro ciclo para hacer la asignación de un color a un nodo; este ciclo asigna el "color" 0 si no puede asignar un color válido.

Respuesta:
Como se deduce del enunciado, sigue el patrón del problema de las 8 reinas.
Inicializamos el vector x en ceros.
Primero, asignamos el color 1 al vértice 1: $x[1] = 1$; y tratamos de asignar, recursivamente y uno por uno, todos los colores válidos para el vértice 2. En este proceso, cada vez que asignamos un color válido al vértice 2, tratamos de asignar, recursivamente y uno por uno, todos los colores válidos para el vértice 3; etc. Si no podemos asignar un color válido al vértice k, entonces tenemos que cambiar la última asignación de color que hicimos al vértice k–1. Cuando asignamos un color válido al vértice n, imprimimos el vector x.

2016-2-I1-P1–Implementación de Backtracking (2/2)

En código:

```
colorear(k):
   repeat:
      proxColor(k)
      if x[k] == 0: return
      if k == n: print(x)
      else: colorear(k+1)
   until False

proxColor(k):
   repeat:
      x[k] = x[k]+1 % m+1
      if x[k] == 0: return
      for j = 1 … n:
         if G[k,j] ≠ 0 and x[k] == x[j]: break
      if j == n+1: return
   until False
```

2016-1-I1-P1–Implementación de Backtracking (1/2)

Estructuras de Datos y Algoritmos – IIC2133

I1

14 abril 2016

Nota 1: *Tienes que sumar **18 puntos** para obtener un 7.*

Nota 2: *Cuando se pide un **algoritmo**, se espera algo parecido a lo siguiente*:

```
selectionSort(a):
   for k = 0 … n-1:
      min = k
      for j = k+1 … n:
         if a[j].key < a[min].key:
            min = j
      exchange(a[k], a[min])
```

1. [**5**] Tenemos una lista de $N(N–1)/2$ números enteros que representan las distancias entre todos los pares posibles formados a partir de N puntos ubicados sobre una línea recta. Queremos determinar la posición (relativa), x_i, de cada uno de los N puntos, suponiendo que el primer punto está en la posición 0, es decir, $x_1 = 0$. Escribe un algoritmo de ***backtracking*** para resolver el problema.

P.ej., si la lista es L = [1, 2, 2, 2, 3, 3, 3, 4, 5, 5, 5, 6, 7, 8, 10], claramente $x_6 = 10$, ya que 10 es la distancia más grande en la lista (dado que el largo de la lista es 15, sabemos que $N = 6$). Como la siguiente distancia más grande es 8, entonces $x_2 = 2$ o $x_5 = 8$. Como estos dos casos son simétricos, nos da lo mismo elegir cualquiera de ellos como válido; elegimos $x_5 = 8$. La siguiente distancia más grande en L es 7, de modo que $x_4 = 7$ o $x_2 = 3$. Si $x_4 = 7$, entonces las distancias $x_6 – 7 = 3$ y $x_5 – 7 = 1$ deberían aparecer en L; y, efectivamente, aparecen. Y si $x_2 = 3$, entonces $3 – x_1 = 3$ y $x_5 – 3 = 5$ deberían aparecer; y también aparecen. De modo que por ahora no podemos saber si $x_4 = 7$ o $x_2 = 3$. Tenemos que probar una de estas dos posibilidades y ver si nos lleva a una solución; si no nos lleva a una solución, entonces tenemos que probar la otra.

Suponemos L ordenada (si L no está ordenada, primero la ordenamos) y N correctamente dado (si N no está dado, lo calculamos a partir de $N(N–1)/2$ = length(L)); declaramos el arreglo x de N componentes: x_1 a x_N.

Tal como en el ejemplo, asignamos $x_1 = 0$ y $x_N = L$[length(L)], y procesamos L en orden de $L[N–1]$ a $L[1]$, es decir, de mayor a menor. Para cada $L[k]$, $k = N–1, N–2, …, 2, 1$, hacemos lo siguiente: **inferimos los valores de algunos x_q's de modo que sean consistentes** con los valores ya asignados a otros x_p's. Si con esto todos los valores de x quedan asignados, entonces quiere decir que encontramos una solución, así que imprimimos x y terminamos. Pero si aún quedan valores por asignar, entonces pasamos al próximo $L[k]$. Si al momento de inferir los valores, hay más de una forma de hacerlo, entonces **probamos cada una ⇒ *backtracking***.

Conviene manejar L como un conjunto dinámico, de manera que sea fácil eliminar y reinsertar (para el *backtracking*) el máximo y también otros valores arbitrarios; p.ej., un árbol de búsqueda, con valores repetidos (varias distancias pueden ser las mismas, como en el ejemplo). Así, en lugar de "para cada $L[k]$, $k = N–1, N–2, …, 2, 1$", ejecutamos el equivalente a "para el $L[k]$ más grande que va quedando".

2016-1-I1-P1–Implementación de Backtracking (2/2)

En el siguiente pseudocódigo, `asignar` es el procedimiento recursivo que efectivamente implementa el backtracking; recibe como parámetros `x`, `L`, `n` y los valores enteros `izq` y `der`, que delimitan el rango de índices de las posiciones `x[i]` que aún faltan por asignar. Antes de llamar a `asignar` por primera vez, asignamos `x[1] = 0`, `x[n]` = el máximo valor en `L` (que además lo eliminamos de `L`) y asignamos tentativamente `x[n-1]` = el (nuevo) máximo valor en `L` (que también lo eliminamos de `L`). Si `x[n]-x[n-1]` es un valor en `L`, entonces quiere decir que esta última asignación es plausible y llamamos a `asignar` para ver si efectivamente lleva a una solución; al hacer la llamada, `izq` = 2 y `der` = `n-2`, porque `x[2]`, …, `x[n-2]` aún no están asignados.

```
x[1] = 0
x[n] = delMax(L)
x[n-1] = delMax(L)
if x[n]-x[n-1] ∈ L:
   eliminar de L la distancia x[n]-x[n-1]
   return asignar(x, L, n, 2, n-2)
else:
   return False

def asignar(x, L, n, izq, der):
   if vacia(L):
      return True
   solucion = False
   dmax = max(L)
   if abs(x[j]-dmax) ∈ L, j = 1,…,izq-1,der+1,…,n:
      x[der] = dmax
      for j = 1,…,izq-1,der+1,…,n:
         eliminar de L la distancia abs(x[j]-dmax)
      solucion = asignar(x, L, n, izq, der-1)
      if not solucion:
         reinsertar en L las distancias eliminadas recientemente
   if not solucion and abs(x[n]-dmax-x[j]) ∈ L, j = 1,…,izq-1,der+1,…,n:
      x[izq] = x[n]-dmax
      for j = 1,…,izq-1,der+1,…,n:
         eliminar de L la distancia abs(x[n]-dmax-x[j])
      solucion = asignar(x, L, n, izq+1, der)
      if not solucion:
         reinsertar en L las distancias eliminadas recientemente
   return solucion
```

2013-2-I1-P2–Implementación de Backtracking (1/1)

2. *Salida del laberinto*. Podemos representar un laberinto como un arreglo bidimensional de caracteres: los pasajes se marcan con '0's, los muros con '1's, y la única salida con 'e'; todo el perímetro del laberinto está marcado con '1's, excepto la salida.

Describe un algoritmo para encontrar la salida del laberinto a partir de una cierta posición inicial, si es que existe un camino de '0's entre la posición inicial y la salida. La idea es que, estando en una posición cualquiera, hay que intentar seguir por cada una de las cuatro direcciones posibles: izquierda, derecha, arriba, abajo—siempre en el mismo orden, aunque el camino encontrado no sea óptimo. Si es imposible seguir (y aún no estamos en la salida), entonces hay que retroceder a la posición anterior e intentar una nueva dirección a partir de ahí (p.ej., si desde esta posición ya se había intentado seguir por la izquierda y luego por la derecha, ahora hay que intentar seguir por arriba). Puedes suponer que cuentas con un *stack* en el que puedes almacenar posiciones del laberinto.

Algoritmo:
inicializar el stack
celdaActual = *celda de partida —llamamos "celda" a cada posición del laberinto*
while (celdaActual *no es* celdadeSalida)
marcar celdaActual *como visitada*
poner en el stack las celdas vecinas no visitadas de celdaActual
if (*stack está vacío*)
"no se encontró salida"
else
celdaActual = *sacar una celda del stack*
"encontramos la salida"

6 Algoritmos Codiciosos

En esta sección se encuentran los siguientes contenidos:

- Subestructura óptima
- Árbol de Cobertura Mínimo
- Algoritmo Prim
- Algoritmo Dijkstra
- Algoritmo Kruskal
- Conjuntos disjuntos

2020-2-I2-P3–Propiedades Dijkstra (1/3)

Pregunta 3

Dado un grafo $G(V, E)$ dirigido y costos $w(u, v) \in \mathbb{R}$, el algoritmo de Dijkstra busca las rutas más cortas desde un vértice $s \in \mathbb{R}$ otro vértice del grafo. El algoritmo opera bajo dos supuestos:

a. **El grafo no contiene aristas de costo negativo**

   Para un grafo $G(V, E)$, aristas de costo negativo, podemos modificarlo para dejar todas las aristas con costo no negativo. Para esto tomamos el costo de la arista de menor costo en $G(V, E)$, se lo restamos a los costos de todas las aristas. Así, esta arista queda con costo $\mathbf{0}$ y las demás con costo positivo.

   Demuestra mediante un ejemplo que, si hacemos este cambio, las rutas encontradas por **Dijkstra** no necesariamente son rutas más cortas en el grafo original.

b. **El costo de una ruta está definido como la suma de los costos de cada arista en la ruta**

   ¿Qué pasa si definimos el costo de la ruta entre dos nodos como la multiplicación de los costos de cada arista en la ruta?

   Podemos modificar cómo el algoritmo de Dijkstra calcula la distancia d de un vértice:

   Con suma, se define como $d(v) = d(u) + w(u, v)$, con $d(s) = 0$

   Con multiplicación, se define como $d(v) = d(u) \cdot w(u, v)$, con $d(s) = 1$

   Demuestra mediante un ejemplo que, bajo esta definición de d las rutas encontradas por **Dijkstra** no necesariamente son las más cortas.

Solución Pregunta 3a)

Se debe mostrar un Grafo el cual contenga al menos una de sus aristas con costo negativo.

Por ejemplo para buscar el camino más corto entre A y C.

Tenemos el siguiente Grafo:

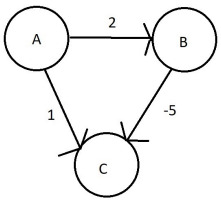


Figura 2: Ejemplo de Grafo con arista negativa

Se debe aplicar la modificación señalada en la prueba, restando el costo de la arista más negativa a **todas** las aristas.

Continuando con el ejemplo, tenemos el siguiente Grafo:

2020-2-I2-P3–Propiedades Dijkstra (2/3)

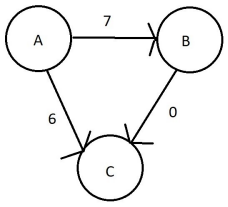


Figura 3: Ejemplo de Grafo aplicando modificación propuesta

Aplicando Dijkstra sobre ambos grafos tenemos que:

1) Sobre el primer Grafo la ruta óptima es **A-B-C**, pero aplicando Dijkstra, este encuentra como óptima la ruta **A-C** ya que *pinta* de negro **C** antes de revisar el nodo **B**.

2) Sobre el segundo Grafo la ruta óptima aplicando Dijkstra es **A-C**. En este caso es la ruta óptima ya que no tenemos costos negativos.

Se demuestra entonces con este ejemplo que haciendo la modificación, la ruta óptima del segundo Grafo no es necesariamente la ruta óptima en el primer Grafo.

Distribución de puntaje:

[**1 pt**] Mostrar ejemplo con arista negativa y realizar la modificación señalada en enunciado.

[**1.5 pt**] Aplicar Dijkstra correctamente sobre grafo con costos positivos.

- [**1 pt**] Si no hay explicación

[**0.5 pt**] Concluir que la modificación señalada en enunciado no implica que necesariamente la ruta óptima del segundo Grafo es ruta óptima en el primer Grafo.

2020-2-I2-P3–Propiedades Dijkstra (3/3)

Solución Pregunta 3b)

Se debe mostrar un Grafo adecuado sobre el cual haga sentido aplicar la modificación al algoritmo de Dijkstra planteado en el enunciado.

Por ejemplo para buscar el camino más corto entre A y C.

Tenemos el siguiente Grafo:

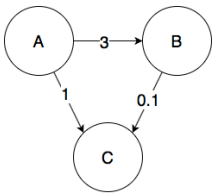


Figura 4: Ejemplo de Grafo con costo total la multiplicación de los costos de cada ruta

Podemos ver que la ruta encontrada con Dijkstra es **A-C** con un costo de 1 dado por la multiplicación entre los costos de las arista **A-C** y el costo inicial de $\mathbf{A = 1}$, es decir, $\mathbf{1 \cdot 1}$. Dijkstra encuentra esta ruta porque *pinta* de negro el nodo C antes de revisar el nodo B, ya que $\boldsymbol{d(c) < d(B)}$.

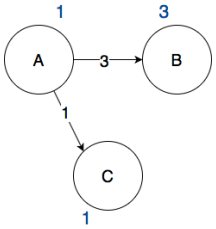


Figura 5: Ruta más corta con dijkstra

Sin embargo, la ruta óptima incluye la arista **B-C** ya que su costo de **0,1** disminuye el costo de la ruta. Por lo que el costo total mínimo para ir de A a C es la ruta **A - B - C** con costo $\mathbf{1 \cdot 3 \cdot 0{,}1 = 0{,}3}$

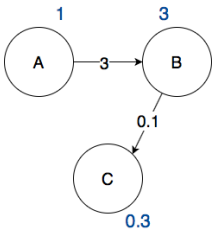


Figura 6: Ruta óptima

2020-2-I3-P1–Subestructura óptima (1/3)



Pontificia Universidad Católica de Chile
Escuela de Ingeniería
Departamento de Ciencia de la Computación

IIC2133 — Estructuras de Datos y Algoritmos
2020 - 2

Interrogación 3

Pregunta 1

La agencia publicitaria para la que trabajas ha sido contratada por XIAOMI MEJOR RELACIÓN PRECIO CALIDAD para posicionar letreros publicitarios en la ruta de Arica a Punta Arenas. Para esto tienes una lista de **n** ubicaciones de letreros disponibles, con sus distancias d_1, d_2, ... , d_n medidas desde Arica y ordenadas crecientemente, y una estimación de que el i-ésimo letrero es visto por w_i personas al día.

El cliente quiere maximizar la cantidad de personas que ven los letreros al día. Tú debes tener en cuenta que las autoridades no permiten que haya dos letreros del mismo producto o compañía a una distancia menor a k entre ellos.

¿Cuáles ubicaciones de letreros debes utilizar para esto?

a) Demuestra que este problema tiene subestructura óptima.

b) Supón la siguiente estrategia codiciosa para resolver el problema: recorrer la lista de ubicaciones de letreros de Arica a Punta Arenas, y poner un letrero en cada ubicación que cumpla con la restricción de las autoridades con respecto a la distancia con el último letrero escogido. Demuestra que, si todos los letreros tienen el mismo **w**, esta estrategia es óptima.

Solución Pregunta 1.a)

Una subestructura óptima significa que la solución óptima del problema original contiene soluciones óptimas de problemas más pequeños, pero similares.

Supongamos que tenemos la solución óptima, y tenemos que la posición del primer letrero es en **m**, donde **m** $<$ **n**. Si consideramos el sub problema, similar al original, donde existen $]\boldsymbol{m}, \boldsymbol{n}]$ ubicaciones, con distancias $]\boldsymbol{d_m}, \boldsymbol{d_n}]$, deberíamos tener una solución óptima del sub problema que pertenezca a la original, maximizando la cantidad de personas que ven los letreros al día y considerando la restricción que debe existir una distancia

2020-2-I3-P1–Subestructura óptima (2/3)

k entre ellos.

De no ser así, tendríamos una solución que permita que más personas vean letreros de XIAOMI entre la posición **m** y **n**, la cual se podría utilizar considerando el problema desde Arica desde la posición **m** en adelante. Contradiciendo así la suposición de que la solución que tenemos es óptima.

Solución alternativa: Demostración por contradicción

Asumamos que tenemos una solución óptima del problema $\boldsymbol{S}$, con las ubicaciones de los letreros que maximizan las vistas por días y cumplen la restricción de distancias.

Ahora elijamos arbitrariamente un letrero (que llamaremos $\boldsymbol{s_i}$) de S para hacer un "corte" en nuestro camino óptimo. Este letrero está a una distancia $\boldsymbol{d_i}$ de Arica. Por la restricción de distancia, el letrero anterior (en S) a $\boldsymbol{s_i}$ debe estar a una distancia menor a $\boldsymbol{d_i - k}$ y el letrero siguiente a una distancia mayor a $\boldsymbol{d_i + k}$.

Claramente el valor estimado de visitas total de nuestro camino óptimo S es igual a la suma entre; la suma de las visitas de cada letrero anterior a $\boldsymbol{x_i}$ (llamaremos a este conjunto $\boldsymbol{S_0}$), $\boldsymbol{w_i}$ y la suma de los letreros que siguen a $\boldsymbol{x_i}$ (llamaremos a este conjunto $\boldsymbol{S_1}$).

Si existiera un conjunto de letreros del conjunto original, cada uno con una distancia menor a $\boldsymbol{d_i - k}$, cuya suma de visitas fuera mayor a la suma de las visitas de $\boldsymbol{S_0}$, entonces uniendo tal conjunto con $\boldsymbol{s_i}$ y $\boldsymbol{S_1}$, obtendramos una solución con mayor número de visitas total que $\boldsymbol{S}$, contradiciendo la suposición de que $\boldsymbol{S}$ es una solución óptima. Por lo tanto $\boldsymbol{S_0}$ debe ser solución óptima del subproblema entre las distancias **0** y $\boldsymbol{d_i - k}$. Por un argumento muy similar, concluimos que $\boldsymbol{S_1}$ también es solución óptima del problema entre las distancias $\boldsymbol{d_i + k}$ y $\boldsymbol{d_n}$.

Por lo tanto demostramos que una solución óptima $\boldsymbol{S}$ está compuesta de dos subestructuras óptimas, más el letrero $\boldsymbol{s_i}$ del corte, por lo que $\boldsymbol{S}$ tiene subestructura óptima.

Solución Pregunta 1.b)

Demostración por contradicción.

Supongamos que todos los pesos de las ubicaciones son iguales y que la estrategia codiciosa de recorrer la lista y asignar las ubicaciones que cumplan las restricciones de las autoridades no es óptima.

Esto significa que, a medida que yo recorro la lista y voy asignando ubicaciones, aquellas que cumplan con la restricción de las autoridades no siempre serán óptimas.

Como la ubicación escogida no es óptima, significa que en algún momento se escogió una ubicación en vez una con mayor utilidad o se escogieron menos ubicaciones que las que podrían haberse escogido.

Como este método de recorrer la lista de ubicaciones escogiendo la primera que cumple con las restricciones, entonces siempre escoge la mayor cantidad de ubicaciones posibles desde el origen (Arica)*.

Por lo tanto nos queda solo la segunda opción. Es decir, que en algún momento, se escogió una ubicación por sobre otra que aportaba con mayor utilidad. Sin embargo, esto contradice nuestra suposición inicial.

Es por esto que queda demostrado que, dado que los pesos de utilidad son iguales, recorrer la lista de ubicaciones y escoger aquellas que cumplan con las restricciones de las autoridades es una estrategia óptima de asignación.

2020-2-I3-P1–Subestructura óptima (3/3)

*Demostración de que siempre se escoge la mayor cantidad de ubicaciones posibles. Este es un problema de programación dinámica con sub-estructura óptima. Por lo tanto, se demostrará por inducción.

[**BI**]

Si hay una sola ubicación, trivialmente se cumple que escogerla (también es la más cercana al punto de origen) cumple con las restricciones y maximiza la cantidad de ubicaciones asignables posible. Por lo tanto, se cumple el caso base.

[**HI**]

Supongamos que tenemos las ubicaciones ordenadas por distancia al origen (Arica) en una lista de la forma U = $\{u_0, u_1, u_2, ..., u_n\}$. Supongamos también que estamos en la ubicación u_i que cumple las restricciones y que agregar u_i a la solución final S maximiza la distancia restante. Es decir $D - d(u_j) < D - d(u_i)\ \forall i < j$ con D la distancia de Arica a Punta Arenas.

[**TI**]

Ahora, tomamos la siguiente ubicación factible que cumple con las restricciones, u_{i+q}, tendremos que $D - d(u_{i+q}) < D - d(u_{i+q-e})$ con $1 < e < q + 1$, pero estas no son soluciones factibles porque incumplen las restricciones de distancia mínima entre ubicaciones. Por otro lado, $D - d(u_{i+q+h}) < D - d(u_{i+q})$ con $1 < h < n - i - q$ porque $i + q < i + q + h$ y las ubicaciones están ordenadas de menor a mayor. Por lo tanto, $D - d(u_{i+q})$ maximiza la distancia restante.

Queda entonces demostrado que escoger la ubicación más cercana al punto de origen que cumpla las restricciones maximiza la utilidad total.

[2pts] Si hay problemas de formalidad

[3pts] Si está correcta la demostración

2020-2-I3-P2–Modificaciones a Kruskal (1/2)

Pregunta 2

La gente de la tierra de Omashu se toma los grupos de amigos muy en serio. Tan en serio, que podemos describirlos matemáticamente (son clases de equivalencia):

- Si a es amigo de b entonces b es amigo de a
- Cada persona es amiga de sí misma y cada persona pertenece a un solo grupo de amigos
- Si a forma parte del grupo X, y b es amigo de a, entonces necesariamente b forma parte de X.

a) Dada una lista F de pares de forma (a, b) que indican amistad entre la persona a y la persona b, describe un algoritmo lineal en el número de pares que calcule la cantidad de grupos de amigos distintos que existen en Omashu.

b) Además de la lista F anterior, se te da una lista U con tríos de la forma (a, b, w). Cada trío indica que a y b no son amigos, pero podrían serlo si se les paga una cantidad positiva w de dinero. Describe un algoritmo a lo más linearítmico (es decir, del tipo $nlogn$) que calcule el costo mínimo necesario para que todos los habitantes de Omashu formen un solo gran grupo de amigos.

Solución Pregunta 2a)

Se puede apreciar que cada grupo de amigos corresponde a un conjunto disjunto, donde nunca va existir un amigo que este en dos grupos distintos. Para la resolución del problema, se cuenta con un grafo donde cada nodo es una persona y las aristas son los pares (a, b) de la lista F

Podemos notar que se pide calcular la cantidad de conjuntos disjuntos, los cuales se representan por los grupos de amigo, es por esto que se puede utilizar el algoritmo visto en clases **Kruskal** con modificaciones, tal como se muestra a continuación

```
1:  procedure OMASHU(F)
2:      groups := 0
3:      for (a, b) ∈ F do
4:          if a = b then
5:              make_set(a)
6:              groups := groups + 1
7:          end if
8:      end for
9:      for (a, b) ∈ F do
10:         if find_set(a) ≠ find_set(b) then
11:             union(a, b)
12:             groups := groups − 1
13:         end if
14:     end for
15:     return groups
16: end procedure
```

2020-2-I3-P2–Modificaciones a Kruskal (2/2)

En primer lugar, como cada persona es amiga de si misma, se inicializa a cada nodo como su propio representante y junto con esto, se tiene un contador que se le va sumando uno por cada vez que se encuentra una nueva persona, de esta forma, después de haber recorrido todos los pares pertenecientes a $\boldsymbol{F}$, el contador $\boldsymbol{groups}$ tiene el total de nodos que tiene el grafo, que inicialmente van a ser nuestros conjuntos disjuntos

Luego, se vuelve a recorrer todos los pares de $\boldsymbol{F}$ y si se encuentra que el representante de $\boldsymbol{a}$ es distinto al de $\boldsymbol{b}$, se une los conjuntos, ya que pertenecen al mismo grupo, dejando al mismo representante para ambos. Como se unen los conjuntos, disminuye la cantidad de conjuntos disjuntos, por lo que hay que restarle uno a $\boldsymbol{groups}$

Finalmente, después de haber recorrido todos los pares de $\boldsymbol{F}$, se tiene la cantidad total de grupos de amigos distintos que existen en Omashu

[**1,5 pt**] Por describir el algoritmo correctamente.

[**0,5 pt**] Por justificar la correctitud del algoritmo.

[**1 pt**] Por justificar la complejidad lineal

En caso de plantear un algoritmo que no sea a lo más lineal, el puntaje máximo a obtener es 1 pto

Solución Pregunta 2b)

Este es similar al anterior. Solo que ahora primero se unirán ambas listas antes de seguir con el resto del algoritmo. Ahora en vez de calcular el número de grupos, se irá sumando el costo cada vez que haya una unión, retornando finalmente el costo total. Este algoritmo entrega complejidad $\boldsymbol{n \log n}$ dada la demostración vista en clases.

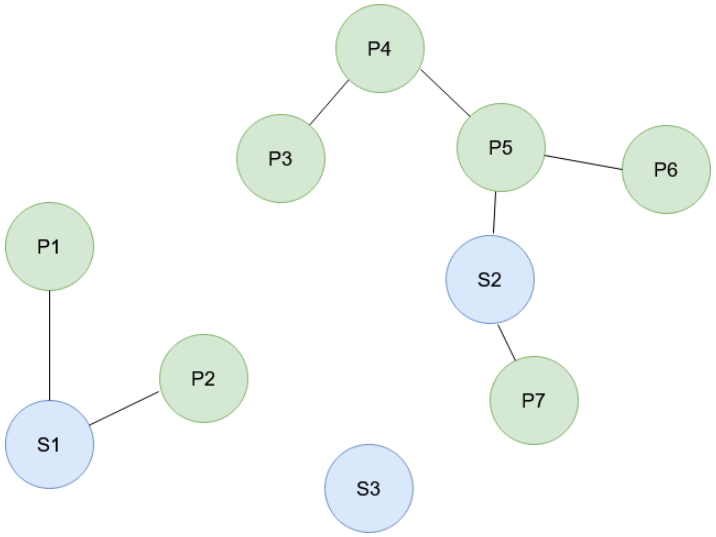
```
1:  procedure OMASHU(F, U)
2:      for (a, b) ∈ F do
3:          union(U, (a, b, 0))
4:      end for
5:      sort(U) de menor a mayor w
6:      cost := 0
7:      for (a, b, w) ∈ U do
8:          if a = b then
9:              make_set(a)
10:         end if
11:     end for
12:     for (a, b, w) ∈ U do
13:         if find_set(a) ≠ find_set(b) then
14:             union(a, b)
15:             cost := cost + w
16:         end if
17:     end for
18:     return cost
19: end procedure
```

2020-1-Ex-P2–Árbol de Cobertura Mínima (MST), Prim, Kruskal (1/2)

2. Luego de una traumática experiencia en Ingeniería Comercial, Patrick decidió renunciar y dedicarse a su verdadera pasión: la jardinería. Planificando el sistema de riego para los jardines del tacaño duque Reginald XII, Patrick se encontró con un problema de optimización que le trajo recuerdos:

   Dada una serie de puntos $\boldsymbol{P}$ que deben ser regados, y una serie de puntos $\boldsymbol{S}$ de tomas de agua, se debe disponer de cañerías entre puntos de manera que desde cada punto $\boldsymbol{p} \in \boldsymbol{P}$ exista una **única** ruta mediante cañerías a algún punto $\boldsymbol{s} \in \boldsymbol{S}$. Considerando que el costo de una cañería es proporcional a su largo, se quiere resolver este problema **minimizando** el costo total de las cañerías dispuestas.

   Considera el siguiente ejemplo de una solución:



   Dado $\boldsymbol{S}, \boldsymbol{P}$ y el grafo no dirigido y con costos $\boldsymbol{G}(\boldsymbol{V}, \boldsymbol{E})$, con $\boldsymbol{V} = \boldsymbol{S} \cup \boldsymbol{P}$ y $\boldsymbol{E} = \boldsymbol{V} \times \boldsymbol{V}$; es decir, un grafo completo. Diseña un algoritmo que resuelva este problema en tiempo $\boldsymbol{O}(\boldsymbol{E} \cdot \log \boldsymbol{V})$. Dicho algoritmo puede ser en prosa o en pseudocódigo: evita lenguajes de programación.

2020-1-Ex-P2–Árbol de Cobertura Mínima (MST), Prim, Kruskal (2/2)

Problema 2

Lo que se pide es, básicamente, encontrar un *bosque de cobertura mínimo*. Es decir, un conjunto de $|S|$ árboles que cubren el grafo, tal que cada árbol contiene exactamente un nodo de S, y el costo total de las aristas de todos los árboles es mínimo.

Este problema se puede resolver de varias maneras, y son todas equivalentes:

Opción 1

Conectamos todos los nodos de S a un nuevo nodo $s^\star$ mediante aristas de costo 0, y luego aplicamos el algoritmo de Prim o Kruskal tal como se vieron en clases.

Teniendo el MST del grafo, eliminamos $s^\star$ y las aristas que lo conectan con los nodos de S para separar el árbol en sub-árboles, formando así el bosque que se pide.

Agregar y quitar este nodo y sus aristas no agrega complejidad a los algoritmos, por lo que están dentro de la complejidad esperada.

Opción 2

En lugar de agregar un nodo extra y aristas de costo cero, podemos modificar uno de los algoritmos para poder manejar este caso:

Opción 2.1: Prim

El algoritmo de Prim parte poniendo de un nodo "de partida" en el heap y genera el árbol a partir de ahi, marcando cuales nodos ya fueron "incluidos". Como para este problema debemos generar un árbol por cada nodo de S, simplemente hay que partir desde todos al mismo tiempo y construir los árboles del bosque mínimo de manera simultánea. Para eso basta con partir poniendo **todos** los nodos de S en el heap con prioridad mínima, para que todos partan siendo incluidos.

Estos nodos eventualmente iban a ser agregados al heap: forzar que se agreguen al comienzo no modifica la complejidad del algoritmo.

Opción 2.2: Kruskal

El algoritmo de Kruskal parte diciendo que cada nodo del grafo es su propio conjunto, y va agregando aristas hasta que todos los nodos son parte del mismo conjunto. Sabemos que los nodos de S no pueden pertenecer al mismo sub-arbol, asique sus conjuntos jamás deberian unirse. Para esto, al principio del algoritmo, podemos unir todos los nodos de S en un mismo conjunto: así jamás se eligiran aristas que los conecten.

Estos nodos eventualmente iban a ser unidos al conjunto final, forzar su union al comienzo no modifica la complejidad del algoritmo.

[6pts] Por proponer un algoritmo correcto y de complejidad correcta
[4pts] Si el algoritmo es correcto pero la complejidad no lo es
[1pts] Por calcular correctamente la complejidad, pero el resto está mal.
[1pts] Por identificar las características de la solución (*bosque de cobertura mínimo*), pero no resuelve el problema.

2019-1-C5–Conjuntos disjuntos, Kruskal (1/1)

Estructuras de Datos y Algoritmos - IIC2133
Control 5

1) Se tiene un *stream* s = d1 d2 d3 … de largo indefinido donde cada dato d = (u,v) representa una arista no direccional que se agrega a un grafo G inicialmente vacío. La idea es que G permanezca acíclico, por lo que si la arista a agregar forma un ciclo se detiene la lectura del stream y se retorna G. **Explica en detalle** cómo llevar a cabo este proceso, determinando de manera eficiente en cada paso si la arista a agregar forma un ciclo en G. **Cuidado**: no conoces todos los vértices por adelantado.

Propuesta de solución:

Mencionar que se puede resolver mediante una implementación de un algoritmo similar a Kruskal, mediante conjuntos disjuntos. En este caso, no es necesario ordenar las aristas (dado que llegan a través del stream).

Para cada elemento del stream d = (u,v), se revisa si ya existen. Debo revisar en una tabla de hash/arreglo dinámico de manera que la complejidad de determinar si los vértices fueron o no explorados anteriormente sea O(1) **[1 punto]**.

Si alguno no existe, se realiza makeset(u) y/o makeset(v) y se indica que estos son su propios representantes. Se agregan a la tabla de hash/arreglo dinámico **[1 punto]**.

Se hace find(u) y find(v), identificando a sus representantes **[1 punto]**.

Si tienen el mismo representante, significa que pertenecen al mismo conjunto y no los uno
Justificar su correctitud. Si dos vértices tienen el mismo representantes, implica necesariamente que pertenecen al mismo subconjunto (fueron unidos en algún momento por una arista del stream). De esta manera, el algoritmo debe finalizar su ejecución sin incorporar la arista que genera el ciclo. En caso de que no tengan el mismo representante, implica que no pertenecen al mismo subconjunto del grafo G, lo que implica que unirlos no generaría un ciclo **[2 puntos]**.

Si tienen diferente representante, los uno con Union(u,v) **[1 punto]**.

2018-2-I3-P3–Algoritmo Prim, Árbol de cobertura mínimo (1/3)

3. **MSTs**

Considera el siguiente grafo no direccional con costos, representado matricialmente:

	a	***b***	***c***	***d***	***e***	***f***	***g***
a		2	4	1			
b	2			3	10		
c	4			2		5	
d	1	3	2		7	8	4
e		10		7			6
f			5	8			1
g				4	6	1	

	¿sol?	*dist*	*padre*
a	*F*	0	–
b	*F*	∞	–
c	*F*	∞	–
d	*F*	∞	–
e	*F*	∞	–
f	*F*	∞	–
g	*F*	∞	–

Ejecuta paso a paso el algoritmo de Prim para determinar un árbol de cobertura de costo mínimo, tomando ***a*** como vértice de partida. En cada paso muestra la versión actualizada de la tabla a la derecha, en que *¿sol?* indica si el vértice ya está en la solución: *V* o *F*.

2018-2-I3-P3–Algoritmo Prim, Árbol de cobertura mínimo (2/3)

	0		
	¿sol?	dist	padre
a	*F*	0	–
b	*F*	∞	–
c	*F*	∞	–
d	*F*	∞	–
e	*F*	∞	–
f	*F*	∞	–
g	*F*	∞	–

	1		
	¿sol?	dist	padre
a	*V*	0	–
b	*F*	2	a
c	*F*	4	a
d	*F*	1	a
e	*F*	∞	–
f	*F*	∞	–
g	*F*	∞	–

	2		
	¿sol?	dist	padre
a	*V*	0	–
b	*F*	2	a
c	*F*	2	d
d	*V*	1	a
e	*F*	7	d
f	*F*	8	d
g	*F*	4	d

	3		
	¿sol?	dist	padre
a	*V*	0	–
b	*V*	2	a
c	*F*	2	d
d	*V*	1	a
e	*F*	7	d
f	*F*	8	d
g	*F*	4	d

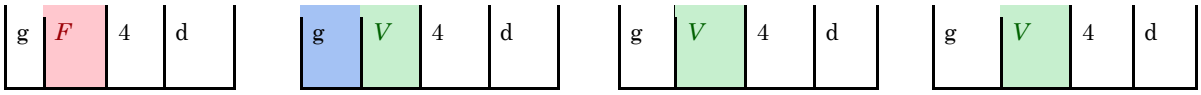
	4		
	¿sol?	dist	padre
a	*V*	0	–
b	*V*	2	a
c	*V*	2	d
d	*V*	1	a
e	*F*	7	d
f	*F*	5	c

	5		
	¿sol?	dist	padre
a	*V*	0	–
b	*V*	2	a
c	*V*	2	d
d	*V*	1	a
e	*F*	6	g
f	*F*	1	g

	6		
	¿sol?	dist	padre
a	*V*	0	–
b	*V*	2	a
c	*V*	2	d
d	*V*	1	a
e	*F*	6	g
f	*V*	1	g

	7		
	¿sol?	dist	padre
a	*V*	0	–
b	*V*	2	a
c	*V*	2	d
d	*V*	1	a
e	*V*	6	g
f	*V*	1	g

2018-2-I3-P3–Algoritmo Prim, Árbol de cobertura mínimo (3/3)



- 0.85 ptos por cada paso correcto (si no mencionó la columna padre en todas las etapas, entregar puntaje igual) del 1 al 7 (el paso 0 no daba puntaje)
- Si ejecutó el algoritmo incorrectamente pero la tabla final es correcta se entregará el máximo entre 3 y 0.85*N pasos correctos
- Nota, el paso 3 y 4 son intercambiables, no importaba el orden en que se hicieran

2018-2-Ex-P3–Demostración de estrategia codiciosa (1/1)

3. Algoritmos codiciosos

Supongamos que quieres hacer una caminata por un parque nacional, que te va a tomar varios días. Tu estrategia es caminar lo más posible de día, pero acampar cuando oscurece. El mapa proporcionado por la oficina de turismo muestra muchos buenos lugares para acampar, y tú has decidido que cada vez que llegues a uno de estos lugares vas a calcular (correctamente) si alcanzas a llegar al próximo antes de que oscurezca; en caso afirmativo, sigues; de lo contrario, te detienes y acampas.

Argumenta rigurosamente que esta estrategia minimiza el número de detenciones para acampar que vas a tener que hacer.

Por contradicción.

Primero, algunos supuestos y definiciones. Sean L la longitud total del camino, d la distancia que puedes caminar en un día, y $x_1, x_2, \ldots x_n$ los puntos de detención que muestra el mapa (distancias desde la entrada del camino). Decimos que un conjunto de puntos de detención es *válido* si
- la distancia entre cualquier par de puntos adyacentes (en el conjunto) es a lo más d,
- la distancia desde la entrada al primer punto es a lo más d, y
- la distancia desde el último punto a la salida del camino es a lo más d.

Sean $R = \{x_{p1}, x_{p2}, \ldots, x_{pk}\}$ el conjunto (válido) de puntos de detención elegidos por tu estrategia codiciosa, y $S = \{x_{q1}, x_{q2}, \ldots, x_{qm}\}$, con $m < k$, un conjunto válido de menos puntos que R.

Primero, los puntos de R están más lejos que los puntos de S; es decir, para cada $j = 1, 2, \ldots, m$, tenemos que $x_{pj} \geq x_{qj}$, lo que demostramos por inducción sobre j.

> Para $j = 1$, sale de la definición de la estrategia codiciosa: tú viajas lo más posible el primer día antes de detenerte.
>
> Para $j > 1$ y suponiendo que la afirmación es válida para todo $i < j : x_{qj} - x_{qj-1} \leq d$, y también $x_{qj} - x_{qj-1} \geq x_{qj} - x_{pj-1}$, lo que implica que $x_{qj} - x_{pj-1} \leq d$. Es decir, una vez que tú sales de x_{pj-1}, podrías caminar hasta x_{qj} en un día, por lo que x_{pj}, que es donde finalmente vuelves a detenerte, solo puede estar más lejos que x_{qj}.

Segundo, como $m < k$, entonces $x_{pm} < L - d$; de lo contrario no tendría sentido que te detengas en x_{pm+1}. Como además $x_{qm} \leq x_{pm}$, por la demostración anterior, resulta que $x_{qm} < L - d$, lo que contradice que S sea un conjunto válido de puntos de detención.

2018-2-Ex-P5–Rutas disjuntas (1/1)

5. Rutas disjuntas

Un conjunto de rutas es disjunto si sus conjuntos de aristas son disjuntos, es decir, ningún par de ru-tas comparte una arista (aunque muchas rutas pueden pasar por algunos de los mismos nodos).

Dado un grafo direccional acíclico $G = (V, E)$, en que un nodo s no tiene aristas que llegan a él, y otro nodo t no tiene aristas que salen de él, el problema de las rutas direccionales disjuntas es encontrar el número máximo de rutas disjuntas de s a t en G.

El problema puede resolverse usando flujos en redes. Primero, construimos una red de flujo a partir de G: s y t son la fuente y el sumidero, respectivamente, y cada arista tiene capacidad 1.

a) Demuestra que si hay k rutas disjuntas en el grafo original G, entonces el valor del flujo máximo de s a t en la red construida a partir de G es a lo menos k.

   Podemos hacer que cada una de las rutas disjuntas lleve una unidad de flujo: el flujo en una arista vale 1 si la arista pertenece a una de las rutas, y vale 0 para todas las otras aristas. Este flujo cumple con las condiciones de (i) no sobrepasar las capacidades de las aristas, y (ii) no producir ni acumular flujo en los nodos distintos de s y t.

b) ¿Qué otra propiedad es necesario poder demostrar para poder concluir que el valor del flujo máximo en la red corresponde al número de rutas disjuntas en G? Solo enuncia la propiedad, de manera clara y precisa; no se pide que la demuestres.

   En a) demostramos que si hay k rutas disjuntas, entonces el flujo es a lo menos k. Por lo tanto, lo que falta por ser demostrado es que **si en la red hay un flujo de valor k, entonces en G hay k rutas disjuntas**.

c) ¿Qué propiedad del problema del flujo máximo da pie para la siguiente afirmación: "El número máximo de rutas disjuntas de s a t en G es igual al número mínimo de aristas (de G) que es necesario sacar para separar s de t (es decir, para que no queden rutas de s a t en G)."?

   La propieded de que el flujo máximo es igual a la capacidad del corte de capacidad mínima (*max-flow min-cut*).

2018-1-I3-P1–Dijkstra modificado (1/1)

Estructuras de Datos y Algoritmos – IIC2133

I3

30 de mayo, 2018

1. Algoritmo de Dijkstra sobre un grafo direccional $G = (V, E)$

El algoritmo de Dijkstra puede resumirse así: Primero, colocamos el vértice fuente s en la solución — un árbol de rutas mínimas, T, originalmente vacío. Luego, construimos T de a una arista a la vez, agregando siempre a continuación la arista que da una ruta más corta desde s a un vértice que no está en T; es decir, agregamos vértices a T en el orden de sus distancias (a través de T) a s. Esta es la versión abstracta del algoritmo.

a) Una versión concreta particular del algoritmo es la que estudiamos en clase, y que toma tiempo $O(E \log V)$. Este tiempo es conveniente cuando el número de aristas de G es $O(V)$, o en general, significativamente menor que V^2. Pero si G es muy denso, es decir, si el número de aristas es más bien $O(V^2)$, entonces sería preferible una versión del algoritmo que tome tiempo $O(V^2)$, y que por lo tanto es lineal en el número de aristas de G. Describe una versión del algoritmo con esta propiedad y justifica que es así.

Respuesta: Mantenemos un arreglo, indexado por los números de los vértices, con las distancias de cada vértice a s, todas inicialmente *infinito*, excepto la del propio s (que es 0). Cada vez que agregamos un vértice v a T, actualizamos, via la operación *reduce*, las distancias a s de los vecinos de v — cada actualización toma tiempo $O(1)$ y a lo más hay $|V|-1$ actualizaciones— y luego buscamos el vértice w que quedó más cerca de s —hay que hacer $|V|-1$ comparaciones—; w hará las veces de v en la próxima iteración. Es decir, en cada iteración hacemos $O(V)$ operaciones; y en total hacemos $|V|-1$ iteraciones.

b) Muestra que en general el algoritmo de Dijkstra efectivamente no encuentra (todas) las rutas más cortas a partir de s si G tiene algunas aristas con costos o pesos negativos.

Respuesta: Basta con dar un (contra)ejemplo. Lo primero que hace el algoritmo es mirar a todos los vecinos de s y agregar a T el más cercano de estos. Supongamos que los vecinos son dos, u y v, y que los costos de las aristas son $w(s, u) = 2$ y $w(s, v) = 3$. Entonces el algoritmo agrega u a T, con distancia 2. Sin embargo, si hay una arista (u, v) con costo $w(u, v) = -1$, significa que la distancia de s a v es realmente 1 ($= 2 + -1$, yendo a través de u) y no 3 (yendo directo), y por lo tanto v está más cerca de s que u y debería haber sido agregado a T antes que u. O bien, si hay una arista (v, u) con costo $w(v, u) = -2$, entonces la distancia de s a u es 1 ($= 3 + -2$, yendo a través de v) y no 2 (yendo directo).

2018-1-I3-P3–Kruskal, DFS, Conjuntos disjuntos, Árbol de costo mínimo (1/1)

3. MST de un grafo no direccional $G = (V, E)$

El algoritmo de Kruskal se puede resumir así: Primero, ordenamos las aristas, según sus pesos o costos, de menor a mayor, y el árbol T que queremos encontrar está inicialmente vacío. Luego, vamos considerando una a una las aristas en el orden dado, y unimos una arista a T a menos que cierre un ciclo. El algoritmo termina cuando hay $|V|-1$ aristas en T.

a) ¿Cuál es la complejidad de este algoritmo si simplemente usamos DFS para decidir si una arista cierra un ciclo? Deduce paso a paso tu respuesta.

Respuesta: Primero, ordenar las aristas de $G \rightarrow E \log E$; luego, aplicar DFS en un grafo con a lo más V aristas (el número de aristas que finalmente va a tener el árbol T) $\rightarrow V$; como esto último hay que repetirlo en las E iteraciones $\rightarrow EV$; así, en total $\rightarrow E \log E + EV = O(EV)$.

b) Si el algoritmo es implementado usando conjuntos disjuntos, ¿cuál es la relación entre los conjuntos disjuntos —específicamente, cuando los representamos como árboles— y los árboles que constituyen las componentes conectadas de T, en cada paso del algoritmo?

Respuesta: Los árboles que representan los conjuntos disjuntos son las mismas componentes —es decir, tienen los mismos vértices— que los árboles que constituyen las componentes conectadas de T ; solo que las formas de los árboles pueden ser diferentes.

c) [Aparte de a y b] Sea $G(V, E)$ un grafo conectado con costos. Si C es cualquier ciclo de G, entonces demuestra que la arista más costosa (o pesada) de C no puede pertenecer a un MST de G.

Respuesta: Partimos de la propiedad del "corte": dado un corte en G, la arista de menor costo que cruza el corte pertenece a algún mst de G, y todo mst de G contiene una arista de menor costo que cruza algún corte.

2018-1-I3-P4–Operación union y find de conjuntos disjuntos (1/1)

4. Conjuntos disjuntos

a) Para cada una de las siguientes implementaciones de conjuntos disjuntos, ¿cuál es la complejidad —peor caso— de ejecutar una única operación *union*? ¿y una única operación *find*? Justifica, preferentemente ayudado por un dibujo.

	union	*find*
arreglo simple	**O(*n*)**	**O(1)**
lista ligada lineal	**O(1)**	**O(*n*)**
árboles con raíz (representados mediante arreglos)	**O(1)**	**O(*n*)**
árboles con raíz, en que el árbol más pequeño apunta al más grande	**O(1)**	**O(log*n*)**

b) Considera el problema de crear un laberinto a partir de un arreglo rectangular de $n \times m$ celdas, en el que la entrada está en la celda de la esquina de arriba a la izquierda y la salida está en la celda de la esquina de abajo a la derecha; las celdas están separadas de sus celdas vecinas por murallas. Inicialmente, están todas las murallas, excepto a la entrada y a la salida. La idea es repetidamente elegir una muralla de manera aleatoria y botarla si las celdas que la muralla separa no están conectadas entre ellas, y así las conectamos (por supuesto, nunca botamos las murallas del perímetro del arreglo). Terminamos cuando las celdas de la entrada y la salida queden conectadas entre ellas (aunque para obtener un "mejor" laberinto conviene seguir botando murallas hasta que cualquier celda sea alcanzable desde cualquier otra celda).

Explica cómo puedes resolver este problema usando una estructura de datos para conjuntos disjuntos.

Respuesta:

Cualquiera de las estructuras de datos para conjuntos disjuntos sirve para resolver el problema; lo importante es entender bien lo que representan los conjuntos y el rol de las operaciones *find* y *union*.

Inicialmente, dado que no hay dos celdas conectadas entre ellas, cada celda está en un conjunto por sí misma.

Entonces, repetidamente, elegimos una muralla de manera aleatoria —representada por las dos celdas adyacentes a la muralla— y ejecutamos dos operaciones *find* para determinar a qué conjunto pertenece cada celda.

Si las celdas pertenecen al mismo conjunto, significa que ya están conectadas entre ellas y por lo tanto no conviene botar esa muralla.

Por el contrario, si las celdas pertenecen a conjuntos distintos, significa que no están conectadas entre ellas, por lo que botamos la muralla, es decir, realizamos una operación *union* entre los conjuntos.

Podemos detener el algoritmo cuando la celda de la entrada y la celda de la salida queden en el mismo conjunto, o, mejor, cuando solo quede un conjunto (todas las celdas están conectadas entre ellas).

2018-1-Ex-P1–Demostración de estrategia codiciosa (1/1)

Estructuras de Datos y Algoritmos – IIC2133

Examen

27 de junio, 2018

1. Algoritmos codiciosos

Tú debes conducir un auto de Santiago a Puerto Montt, suponemos que en línea recta por la autopista. Cuando el estanque de bencina del auto está lleno, te permite viajar k kilómetros sin detenerte. Tú tienes un mapa con las distancias entre bombas de bencina a lo largo de la autopista; las distancias entre bombas consecutivas son todas $\leq k$ kilómetros. Tú inicias el viaje con el estanque lleno, y quieres hacer el menor número posible de detenciones para cargar bencina durante el viaje. Una solución factible es una lista de bombas de bencina en las que tienes que detenerte y que te permiten llegar a Puerto Montt; y una solución óptima es una solución factible con el menor número de bombas.

a) Propón una elección codiciosa —cómo eliges la próxima bomba de bencina en la que tienes que detenerte— que te permita minimizar localmente el número de detenciones [**1.5 pts.**].

Respuesta. Las bombas posibles son aquellas que están a una distancia $\leq k$ km desde mi posición actual, ya que tengo el estanque lleno; elijo codiciosamente la bomba que está más lejos entre éstas (y allí lleno nuevamente el estanque).

b) Da un contraejemplo que muestre que tu elección codiciosa no minimiza el número total de detenciones [**1 pt.**]; o, por el contrario, demuestra que sí lo hace, respondiendo **c**) y **d**).

Me salto **b**): la elección codiciosa de **a**) minimiza el número de detenciones, como se demuestra en **c**) y **d**).

c) Demuestra que hay una solución óptima que se obtiene haciendo la elección codiciosa de **a**) [**2 pts.**].

Respuesta. Supongamos que las próximas bombas son S, …, T, …, U, en orden de cercanía a nuestra posición actual, y que T es la más lejana que no está a más de k km de distancia (es decir, T es la próxima bomba en que deberíamos detenernos según nuestra elección codiciosa).

Y supongamos que hay una solución óptima en que la próxima bomba es S (más cercana que T) y la que le sigue es U (más lejana que T); es decir, la solución óptima es $\{S, U, \ldots\}$. Es fácil ver que en esta solución óptima podemos cambiar la elección de detenernos en S por la de detenernos en T, sin aumentar el número total de detenciones: si desde S se puede llegar a U sin detenerse (entre medio), entonces desde T también se puede llegar a U sin detenerse, ya que T está más cerca de U que S; y, por supuesto, podemos llegar desde nuestra posición actual a T sin detenernos, ya que la distancia no es mayor que k km.

d) Finalmente, demuestra que si combinas la elección codiciosa de **a**) con una solución óptima del subproblema que te queda por resolver (una vez hecha la elección codiciosa), obtienes una solución óptima al problema original [**2.5 pts.**].

Respuesta. Este argumento vale para cualquier solución óptima, independientemente de si la primera elección se hizo codiciosamente o no. En una solución óptima, en que la primera detención es en una bomba R a los j km ($j \leq k$), **el resto del recorrido tiene que ser una solución óptima** (es decir, que minimiza el número de detenciones) **al problema de viajar desde R hasta Puerto Montt** [esto que está en *negrita* es lo que hay que demostrar, p.ej., por contradicción, como sigue]. La razón es que si el resto del recorrido no fuera óptimo, entonces podríamos encontrar otra secuencia de detenciones para viajar desde R hasta Puerto Montt con menos detenciones, y podríamos usar esta secuencia en el viaje de Santiago a Puerto Montt, después de detenernos en R, reduciendo el número total de detenciones para este viaje, y contradiciendo así la suposición de que la solución que tenemos es óptima.

2018-1-Ex-P5–Prim, Dijkstra (1/2)

5. Algoritmos de Dijkstra y Prim

El algoritmo de Prim produce como resultado un árbol que conecta todos los nodos y que tiene costo mínimo. También el algoritmo de Dijkstra, ejecutado hasta llegar a todos los nodos del grafo, da como resultado un árbol de rutas mínimas, con las rutas desde el nodo inicial hasta cada uno de los otros nodos del grafo. La duda que surge es si estos árboles tienen algo en común.

a) Demuestra con un contraejemplo que el árbol producido por Dijkstra no es necesariamente mínimo.

2018-1-Ex-P5–Prim, Dijkstra (2/2)

b) Demuestra con un contraejemplo que el árbol producido por Prim no necesariamente tiene las rutas más cortas desde el nodo inicial al resto de los nodos.

Respuesta:

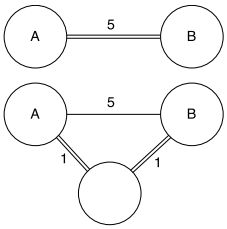
Un posible grafo es V = {A, B, C}, E = {{A,B,4}, {A, C, 2}, {B, C, 3}}. Si partimos haciendo Dijkstra desde A el árbol resultantes tiene las aristas {A, B, 4} y {A, C, 2} y el árbol resultante de hacer Prim desde A tiene las aristas {A, C, 2}, {B, C, 3}.
a) El árbol de Dijkstra no es mínimo ya que tiene costo 6 mientras que el de Prim tiene costo 5 **[3pts].**

b) El árbol de Prim no tiene la ruta óptima a C ya que la ruta tiene costo 5 mientras que en la ruta de Dijkstra tiene costo 4 **[3pts].**

2017-1-I3-P3–Árbol de Corbetura Mínimo, Kruskal (1/1)

3. Sea $G = (V, E)$ un grafo no dirigido y $T \subseteq E$ el único árbol de cobertura de costo mínimo (MST) de G. Suponga ahora que G' se construye agregando nodos a G y aristas que conectan estos nuevos nodos con nodos de G. Finalmente, sea T' un MST de G'.

   a) (1/3) Demuestre que no necesariamente $T \subseteq T'$.
   Respuesta: Basta con dar un contraejemplo:



   Las aristas dobles muestran el árbol del grafo. En la primera imagen el árbol T es $\{(A, B)\}$, mientras que en el segundo grafo el árbol T' es $\{(A, C), (B, C)\}$. Es evidente que $T \notin T'$
   Asignación de puntaje: Si se da un contraejemplo o se hace una demostración formal: **1pto**. Si se hace una demostración formal pero tiene algun error pequeño: **0.5pts**. Else: **0pts**.

   b) (2/3) Dé una condición necesaria y suficiente para garantizar que $T \subseteq T'$. Demuéstrela.
   Respuesta: Para asegurar que $T \in T'$, se debe asegurar para cada arista $(u, v) \in T$ que: Si se genera un camino c nuevo que conecta u con v, entonces al menos existe una arista $a \in c$ tal que el costo de a es mayor al costo de (u, v).

   Demostración:
   Suficiencia: Dado que el algoritmo de kruscal es correcto, podemos decir lo siguiente:
   Dados dos nodos $u, v \in G$ tal que $(u, v) \in T$, se tiene que existe un camino nuevo que conecta u con v en el cual existe una arista a más cara que (u, v). En algun paso de la ejecución del algoritmo de kruscal se tendrá que u y v están en dos grupos no conectados de nodos. Se puede asegurar que el algoritmo de kruscal no conectará los grupos de nodos de u con el de v por la arista a ya que primero se revisan las aristas más bartas, por lo que primero se conectarían a través de (u, v). Por lo tanto, esta propiedad es suficiente.

   Necesaria: Si no se cumple esta propiedad entonces ejecutando el algoritmo de kruscal se conectaría u con v a través del nuevo camino, por lo que no se agregaría (u, v) al árbol T'. Por lo tanto, si la propiedad, puede pasar que $T \notin T'$.

   Respuesta equivalente: Para todo corte del grafo G' tal que existen nodos de G en ambos lados del corte, se debe cumplir que la arista más barata que cruza el corte pertenece al árbol T.

2017-1-Ex-P1-b—i–Kruskal, Conjuntos Disjuntos (1/1)



PONTIFICIA UNIVERSIDAD CATÓLICA DE CHILE
ESCUELA DE INGENIERÍA
DEPARTAMENTO DE CIENCIA DE LA COMPUTACIÓN

IIC 2133 — Estructuras de Datos y Algoritmos
Examen
Primer Semestre, 2017

Duración: 3 hrs.

1. Para cada una de las siguientes afirmaciones, diga si es *verdadera* o *falsa*, **siempre justificando** su respuesta.

   a) Quick Sort es un algoritmo de ordenación estable. **Respuesta:** Falso. Basta con dar un contraejemplo.

   b) Sea e la segunda arista más barata de un grafo dirigido acíclico G con más de dos nodos y aristas con costos diferentes. Entonces e pertenece al árbol de cobertura de costo mínimo para G. **Respuesta:** Verdadero. Si usamos el algoritmo de Kruskal, la segunda arista más barata es siempre agregada al MST puesto que no puede formar un ciclo en el bosque construido hasta el momento.

   c) Radix Sort es un algoritmo de ordenación que puede ordenar n números enteros y cuyo tiempo de ejecución está siempre en $\Theta(n)$. **Respuesta:** El tiempo de ejecución de Radix Sort es $d(n + k)$, cuando los datos están en $[0, k]$. Basta entonces con que k sea suficientemente grande (por ejemplo, exponencial en n), para que el algoritmo no sea $\Theta(n)$

   d) Sea A un árbol rojo-negro en donde cada rama tiene n nodos negros y n nodos rojos. Si al insertar una clave nueva en A se obtiene el árbol A', entonces cada rama de A' tiene $n + 1$ nodos negros. **Respuesta:** Verdadero. Un árbol rojo negro como A corresponde a un árbol 2-4 "completamente saturado"; es decir, uno que contiene nodos con 3 claves. Al insertar una clave nueva, el árbol 2-4 aumentará su altura en 1. Esto significa que su equivalente rojo-negro debe tener un nodo negro más por cada rama.

   e) Si se tienen dos tablas de hash, una con direccionamiento abierto y otra cerrado, las dos del mismo tamaño m y el mismo factor de carga α, ambas ocupan la misma cantidad de memoria.

   f) Sea A un árbol AVL y ℓ_1 y ℓ_2 los largos de dos ramas de A. Entonces $|\ell_1 - \ell_2| \leq 1$. **Respuesta:** Falso. Basta dar un contraejemplo

   g) La operación de inserción en un árbol AVL con n datos realiza a lo más una operación *restructure* pero toma tiempo $O(\log n)$. **Respuesta:** Verdadero, puesto que la inserción debe revisar que el balance esté correcto a lo largo de la rama donde se insertó el dato. Y esa rama tiene tamaño $O(\log n)$.

   h) Si A es un ABB y n es un nodo de A, el *sucesor* de n no tiene un hijo izquierdo. **Respuesta:** Falso. La propiedad no se cumple en general cuando n no tiene un hijo derecho.

   i) Es posible modificar la implementación de la estructura de datos para conjuntos disjuntos vista en clases, de manera que permita des-unir dos conjuntos en $O(1)$.

   j) Como diccionario, una tabla de hash es más conveniente que un árbol rojo negro en cualquier aplicación. (Sin considerar la dificultad de implementación).

   k) Sea p una secuencia de dos o más números diferentes y sea q una permutación de p distinta de p. Adicionalmente, sean A_p y A_q los árboles binarios de búsqueda que resultan de, respectivamente, insertar en orden los elementos de p y q en árboles binarios de búsqueda vacíos. Entonces A_p y A_q son distintos. **Respuesta:** Falso.

   l) Re-hashing, el procedimiento que construye una nueva tabla de hash a partir de otra existente, toma tiempo $O(n)$ en una tabla con colisiones resueltas por encadenamiento de tamaño m que contiene n datos. **Respuesta:** Falso. El tiempo depende del tamaño de la tabla y del número de datos; específicamente es $O(m + n)$.

2017-1-Ex-P5–Prim, Dijkstra, Heap, Árbol rojo-negro (1/2)

5. En sus tiempos libres, Gianna Zecca, alumna de programación avanzada, ha estado estudiando algunos temas del libro de Cormen, Leiserson, Rivest y Stein. A pesar de que las estructuras de datos "Heap binario" y "Árbol Rojo-Negro" le parecen apasionantes, no logra entender la utilidad de los heaps. Según ella, ha encontrado una forma más o menos ingeniosa de usar árboles rojo-negro en vez de heaps dentro de los algoritmos de Prim y Dijkstra de tal manera de obtener *la misma complejidad asintótica*.

   a) ¿Está Zecca en lo correcto respecto a la complejidad asintótica de Prim y Dijkstra con árboles rojo-negro versus heaps? Para responder esta pregunta, piense en una forma "más o menos ingeniosa" de usar árboles dentro de esos algoritmos. **Respuesta:** Las dos operaciones que realizan Dijkstra/Prim sobre la cola son: extraer el mínimo y cambiar prioridad. Al implementar la cola con árboles, extraer el mínimo resulta simplemente de eliminar el mínimo, que es $O(\log n)$. Extraer el mínimo, en un heap, también es $O(\log n)$.

   Para el cambio de prioridad, en un árbol lo podemos hacer con una eliminación si (ingeniosamente) mantenemos un arreglo de punteros P tal que $P[v]$ contiene un puntero al nodo del árbol que representa al nodo v. Esta operación, en un árbol balanceado, toma $O(\log n)$. En el caso de un heap, la operación también toma $O(\log n)$. Concluimos que Zecca está en lo correcto.

   b) Escriba una batería de sólidos argumentos a favor de alguna de las dos afirmaciones:

      i. Es desventajoso implementar el algoritmo de Dijkstra (o Prim) usando árboles binarios de búsqueda balanceados.

2017-1-Ex-P5–Prim, Dijkstra, Heap, Árbol rojo-negro (2/2)

ii. Jorge Baier no debiera complicarse en enseñar heaps, porque (1) los heaps no tienen ventaja alguna sobre los árboles (al menos en el caso de los algoritmos de Prim y Dijkstra) y (2) heapsort no tiene niguna ventaja práctica sobre Quick Sort.

Respuesta: Argumentamos en favor de i. Aunque la complejidad asintótica de ambas implementaciones es la misma, usar los heaps, para este algoritmo, tienen ventajas sobre árboles. Veamos el caso de la memoria. Un heap ocupa menos memoria que un árbol balanceado (cada nodo del árbol necesita almacenar punteros).

Respecto al tiempo, las operaciones de heap debieran ser más rápidas que con un árbol. En particular, la disminución de la prioridad necesita en el heap mover el elemento hacia arriba en el árbol; es decir, no se necesita revisar una rama completa. Esto no ocurre si usamos un ABB, porque la eliminación de un nodo requiere computar el sucesor (al menos), lo que implica en muchos casos revisar una rama completa. En la práctica esto se puede reflejar en los tiempos de ejecución de manera importante.

Para la extracción del más pequeño podemos dar un argumento similar.

2016-2-I1-P4-a–Dijsktra (1/2)

4. **a)** Considera las expresiones aritméticas de la forma (1 + ((2+3) * (4*5))), es decir, "totalmente parentizadas". (Estas expresiones se pueden definir formalmente de la manera recursiva: una *expresión aritmética* es ya sea un número, o un paréntesis abierto seguido por una expresión aritmética seguido por un operador seguido por otra expresión aritmética seguido por un paréntesis cerrado.)

E.W. Dijkstra desarrolló un algoritmo para evaluar este tipo de expresiones, empleando dos *stacks*, uno para los operandos y otro para los operadores, y revisando la expresión de izquierda a derecha:

- Coloca (*push*) los operandos en el stack de los operandos.
- Coloca (*push*) los operadores en el stack de los operadores.
- Ignora los paréntesis abiertos.
- Al encontrar un paréntesis cerrado, saca (*pop*) un operador, saca (*pop*) dos operandos, aplica el operador a los operandos, y coloca (*push*) el resultado en el stack de operandos.
- Cuando se procesa el último paréntesis cerrado, queda un valor en el stack de operandos; ese es el valor de la expresión.

i) [**1**] Ejecuta el algoritmo de Dijkstra para evaluar la expresión (1 + ((2+3) * (4*5))). Muestra el contenido de cada uno de los stacks a medida que vas procesando cada elemento de la expresión.

ii) [**2**] Analiza el algoritmo. ¿Cuántas operaciones básicas, o pasos, debe ejecutar en general, como función del largo de la expresión? ¿Cuál es la profundidad máxima que puede llegar a tener cada uno de los stacks, es decir, de qué propiedades de la expresión depende la profundidad y cómo depende?

2016-2-I1-P4-a–Dijsktra (2/2)

Respuesta:

i)

stack de operandos	stack de operadores
Ø	Ø
1	Ø
1	+
1	+
1	+
1 2	+
1 2	+ +
1 2 3	+ +
1 5	+
1 5	+ *
1 5	+ *
1 5 4	+ *
1 5 4	+ * *
1 5 4 5	+ * *
1 5 20	+ *
1 100	+
101	

ii) Sea n el largo de la expresión y |h| que dice cuántas veces aparece h en la expresión.
Podemos notar por la estructura de la expresión que |(| = |)| = |operadores| = x.
Además que |operandos| = x+1.

Luego n = 4x+1, x = (n-1)/4

Por lo tanto:

|operadores| = (n-1)/4
|operandos| = (n+3)/4

Tamaño stacks:
El máximo del stack de operadores va a ser cuando estén todos los operadores en el stack, lo cual es (n-1)/4, lo mismo ocurre con el de operandos, el cual es (n+3)/4.
Cantidad operaciones:
- Por cada operando/operador se ejecuta un push (1 operación)
- Por cada) se ejecuta 3 pop, se calcula el valor y luego 1 push (5 operaciones)
Por lo tanto la cantidad de operaciones es:
x + (x+1) + 5x = 6x+1 = (3n-1)/2.
(1+(2*3))
n = 9, realizará (3*9-1)/2 = 13 operaciones.

2015-2-Ex-P3–Kruskal, solución óptima, subproblema óptimo (1/1)

3. Demuestra que dado un grafo no direccional $G = (V, E)$ con costos en las aristas, el problema de encontrar un árbol de cobertura de costo mínimo (MST) para G se puede resolver mediante un algoritmo codicioso —en particular, mediante el algoritmo de Kruskal. Para ello, sigue estos pasos:

a) Formula el problema como uno en el cual haces una elección y te quedas con un subproblema del mismo tipo para resolver (¿qué tipo de elección haces en el caso del algoritmo de Kruskal?).

Una manera de construir el árbol es partir con un árbol vacío e ir agregando una arista a la vez, hasta completar $|V|-1$ aristas que cubran todos los vértices de G (y que no formen un ciclo). [El desafío es encontrar una estrategia para ir eligiendo las aristas, como se explica en **b**) y **c**)].

b) Demuestra que hay una solución óptima al problema original que hace la **elección codiciosa** (¿cuál es la elección codiciosa que haces en el algoritmo de Kruskal?).

La elección codiciosa consiste en elegir la arista más liviana de todas las aristas de G; llamémosla e. ¿Es e parte del MST? Sí, como de demuestra a continuación.

Supongamos que tenemos un MST de G, llamémoslo T, y que T no incluye la arista e. ¿Qué pasa si agregamos e a T? Claramente, formamos un ciclo, ya que T es un árbol que cubre **todos** los vértices de G. De este ciclo podemos sacar cualquier arista y volvemos a obtener un árbol de cobertura. Si sacamos una arista distinta de e, y por lo tanto más pesada que e, el árbol resultante tiene un costo total que no es mayor que el costo de T.

c) Demuestra que, habiendo hecho la elección codiciosa, lo que queda es un subproblema tal que si combinas una solución óptima al subproblema con la elección codiciosa hecha, obtienes una solución óptima al problema original.

2015-2-Ex-P8–Propuesta de algoritmo codicioso, complejidad (1/1)

8. En clase resolvimos el siguiente problema de optimización: Dado un conjunto de tareas, cada una con un plazo y una ganancia, encontrar un subconjunto de tareas tal que todas las tareas del subconjunto pueden ser hechas dentro de sus plazos (subconjunto *factible*) y que maximiza la suma de las ganancias (subconjunto *óptimo*). Las tareas sólo pueden ser hechas usando una única máquina por una unidad de tiempo —por lo tanto, sólo se puede hacer una tarea a la vez— y las ganancias se obtienen sólo si las tareas son hechas dentro de sus plazos.

Demostramos que el problema puede resolverse si ordenamos las tareas de mayor a menor ganancia, y empleamos la **estrategia codiciosa** de agregar a la solución la próxima tarea que siendo factible hace aumentar más la ganancia acumulada. Esta solución requiere poder determinar la factibilidad de un subconjunto de tareas, lo que resolvimos explicando que **basta probar la factibilidad de una sola permutación de las tareas del subconjunto: cualquiera en que las tareas estén ordenadas crecientemente por plazos**.

a) ¿Qué complejidad tiene este algoritmo? Justifica.

Es posible mejorar este desempeño usando otro método para determinar la factibilidad de un subconjunto de tareas. Si J es un subconjunto factible de tareas, entonces podemos asignar los tiempos de procesamiento de cada tarea de la siguiente manera: si para la tarea t aún no hemos asignado un tiempo de procesamiento, entonces le asignamos el *slot* $[k–1, k]$, en que k es el mayor entero menor o igual que el plazo de t y el *slot* $[k–1, k]$ está vacío —es decir, postergamos la tarea t lo más que podemos. Así, al construir J de a una tarea a la vez, no movemos las tareas que ya tienen sus *slots* asignados para acomodar una nueva tarea: **si para esta nueva tarea no encontramos un k como el que definimos antes, entonces la tarea no puede programarse**.

b) Demuestra esta última afirmación.

c) Explica cómo se puede implementar esta nueva forma de programar las tareas usando una estructura de conjuntos disjuntos.

d) ¿Qué complejidad tiene este nuevo algoritmo? Justifica.

2015-1-Ex-P3–Dijkstra (1/1)

3. El algoritmo de Dijkstra para determinar las rutas más cortas desde un vértice a todos los otros vértices en un grafo direccional con costos no negativos es el siguiente:

```
void Dijkstra(Vertex s)
   Init(s)
   S = ∅
   Queue q = new Queue(V)
   while ( !q.empty() )
      Vertex u = q.xMin()
      S = S ∪ {u}
      for ( each v in α[u] )
         reduce(u,v)
```

Este algoritmo es un algoritmo codicioso; **demuestra que efectivamente resuelve el problema**:

a) Demuestra que el problema tiene subestructura óptima.

Trivial. Lo hicimos varias veces en clases.

b) Demuestra que después de hacer una elección, te quedas con un problema del mismo tipo, pero más pequeño.

Como vimos en clase, un algoritmo codicioso funciona en **etapas** y usa una **medida de optimización**. P.ej., en este caso, determinemos las rutas más cortas una por una (las etapas) y sumemos los costos de todas las rutas encontradas hasta ahora (la medida); para que esta suma sea minimizada, cada ruta individual debe ser de mínimo costo. Así, si hasta el momento hemos construido k rutas más cortas, entonces (usando nuestra medida de optimización) la próxima ruta a ser construida deberá ser la próxima ruta de longitud mínima más corta.

c) Demuestra que cada vez que tienes que hacer una elección, la elección codiciosa es óptima.

Lo primero que hay que establecer es **cuál** elección codiciosa (p.ej., elegir la arista de menor costo no sirve). Por supuesto, en el caso de Dijkstra la respuesta es elegir, entre aquellos vértices que aún no han sido elegidos, el vértice que está más cerca del vértice s de partida (siguiendo la estrategia explicada en **b**). La demostración de que esta elección es correcta aparece en Cormen et al. [2009] y se basa en el hecho de que los costos de las aristas son no negativos; la hacemos por contradicción.

Sea u el primer vértice que al ser seleccionado no cumple que $u.d = \delta(s,u)$, según la notación vista en clase —es decir, la elección codiciosa no es óptima. ¿Cuál es la situación justo antes de seleccionar a u? La ruta más corta, p, entre s y u pasa por un vértice w, el primer vértice en p que aún no ha sido seleccionado, y sea x el predecesor de w en p: entonces, podemos descomponer p en un tramo de s a x, seguido por la arista (x,w), seguido por el tramo de w a u.

Sabemos que $x.d = \delta(s,x)$ cuando x fue seleccionado; en ese momento, el algoritmo aplica *reduce* a (x,w), por lo que $w.d = \delta(s,w)$ cuando u es seleccionado.

Entonces, como w aparece antes que u en la ruta más corta de s a u y los costos de todas las aristas son no negativos —en particular, los del tramo w a u— tenemos que $\delta(s,w) \leq \delta(s,u)$, por lo que $w.d = \delta(s,w) \leq \delta(s,u) \leq u.d$. Pero ni u ni w habían sido seleccionados cuando seleccionamos a u, por lo que si seleccionamos (codiciosamente) primero a u, significa que $u.d \leq w$.d, lo que sólo es posible si $w.d = \delta(s,w) = \delta(s,u) = u.d$, lo que efectivamente contradice nuestra forma de seleccionar a u.

2015-1-Ex-P5–Kruskal, Árbol de Cobertura mínimo (1/1)

5. Considera el algoritmo de Kruskal para encontrar un árbol de cobertura de costo mínimo (MST) para un grafo $G = (E, V)$ no direccional con costos. Recuerda que Kruskal primero ordena las aristas de menor a mayor costo. Responde como preguntas independientes:

¡ Tanto en **a**) como en **b**), si no hay justificación, el puntaje es **0** !

a) Si todos los costos de las aristas son número enteros en el rango 1 a $|V|$, ¿qué tan rápida puede ser la ejecución de Kruskal? Justifica.

Kruskal toma tiempo $O(V)$ para inicialización, $O(E \log E)$ para ordenar las aristas, y $O(E\ \alpha(V))$ para las operaciones de conjuntos disjuntos; en total, $O(E \log E)$, que es el término "más grande".

Ahora, sabiendo que los costos de las aristas son enteros en el rango 1 a $|V|$, podemos ordenarlas en tiempo $O(V+E) = O(E)$ con *countingSort*, por lo que ahora el término más grande es $O(E\ \alpha(V))$.

b) Si usamos la implementación basada en listas ligadas para representar y manejar conjuntos disjuntos, ¿qué tan rápida puede ser la ejecución de Kruskal? Justifica.

Nuevamente, Kruskal toma tiempo $O(V)$ para inicialización, $O(E \log E)$ para ordenar las aristas, y $O(E\ \alpha(V))$ para las operaciones de conjuntos disjuntos; en total, $O(E \log E)$, que es el término "más grande".

Como vimos en clase, si usamos listas ligadas para representar los conjuntos disjuntos, el tiempo para procesar los conjuntos disjuntos cambia a $O(m+n\log n)$, en que m es el número total de operaciones y n es $|V|$. Entonces, el término más grande seguirá siendo $O(E \log E)$.

2014-1-I2-P5–Algoritmo Prim (1/1)

5) Considera el algoritmo de Prim, estudiado en clase, para encontrar un árbol de cobertura mínimo para un grafo no direccional $G = (V, E)$, a partir del vértice r en E:

```
for ( cada vértice u en V ) u.key = ∞
r.key = 0
formar una cola Q con todos los vértices en V, priorizada según el campo key de cada vértice
π[r] = null
while ( !Q.empty( ) )
        u = Q.extractMin( ) —esta operación modifica la cola Q
        for ( cada vértice v en listadeAdyacencias[u] )
                if ( v ∈ Q ∧ costo(u,v) < v.key )
                        π[v] = u
                        v.key = costo(u,v) —esta operación modifica la cola Q
```

El desempeño de Prim depende de cómo se implementa la cola Q; si Q es un heap binario, entonces Prim toma tiempo $O(E \log V)$: el *for* dentro del *while* mira cada arista de G y para cada una realiza una operación *decreaseKey* sobre un vértice (cuando actualiza $v.key$, en la última línea).

Si los costos de todas las aristas de G son números enteros entre 0 y una constante W (tal que es factible declarar un arreglo de tamaño W+1), describe una forma de implementar la cola Q, tal que Prim corra en tiempo $O(E)$, es decir, que cada operación *descreaseKey* tome tiempo O(1). (Recuerda que si W es constante, es decir, no depende ni de E ni de V, entonces recorrer un arreglo de tamaño W+1 toma tiempo $O(1)$.)

Respuesta:

- Implementamos Q como un arreglo Q[0], Q[1], …, Q[W], Q[W+1]. Cada elemento del arreglo es una lista doblemente ligada de vértices, en que Q[k] contiene todos los vértices cuyo *key* vale k; Q[W+1] contiene los vértices cuyo *key* vale infinito (∞).
- Entonces, *extractMin* consiste simplemente en recorrer Q buscando el primer elemento no vacío; por lo tanto, toma $O(W) = O(1)$.
- También, *decreaseKey* toma tiempo $O(1)$: basta sacar el vértice de la lista en que está (la lista tiene que ser doblemente ligada para que sacar un vértice tome $O(1)$) y agregarlo al comienzo de la nueva lista que le corresponda según su nuevo valor de *key*.

2014-1-I2-P6–Unión Conjuntos Disjuntos (1/1)

6) En algunos lenguajes de programación es posible declarar (explícitamente) dos nombres de variables como equivalentes, es decir, son referencias al mismo objeto. Después de una secuencia de estas declaraciones, el compilador necesita determinar si dos nombres dados son equivalentes. En particular, supongamos que las declaraciones son de la forma `equivalent <id1> <id2>`. Cuando el compilador ve una de estas declaraciones, primero, tiene que determinar si, como consecuencia de declaraciones anteriores, `<id1>` e `<id2>` ya son equivalentes (p.ej., `equivalent <id1> <id3>` … `equivalent <id2> <id3>`), en cuyo caso no hace nada; si `<id1>` e `<id2>` no son equivalentes, entonces debe hacerlos equivalentes a partir de ese momento.

De acuerdo con las estructuras de datos y algoritmos estudiados en clase, ¿qué tan rápidamente puede el compilador procesar una secuencia de p declaraciones que en total involucran n nombres de variables distintos? Justifica.

Respuesta:

Se trata de unión de conjuntos disjuntos. Cada variable es inicialmente un conjunto por sí misma (*singleton*); y cada declaración `equivalent` une dos conjuntos en uno nuevo, si es que esas variables no están ya en el mismo conjunto. Así, hay n singletons y p declaraciones, cada una de las cuales implica dos operaciones *find* y posiblemente una operación *union*; en términos de la notación usada en los apuntes de clase, $m = n + 3p$.

Por lo tanto, si usamos la representación de conjuntos sugerida en la lámina 18 de los apuntes, y usamos unión por rango y compresión de ruta a medida que ejecutamos las operaciones sobre los conjuntos, el compilador puede hacer su trabajo en tiempo $O((n+3p)\alpha(n))$.

2014-1-Ex-P5–Subestructura óptima, función recursiva (1/1)

5) Tú quieres conducir un auto desde Santiago hasta Pueto Montt. Cuando el estanque de bencina del auto está lleno, te permite viajar n kilómetros. Tú tienes un mapa con las distancias entre estaciones de bencina en la ruta. Tu propósito es hacer el menor número posible de detenciones en el viaje.

a) Muestra que este problema tiene *subestructura óptima*.

(Suponemos que partimos con el estanque lleno y que las distancias entre pares consecutivos de estaciones de bencina son todas $\leq n$.)

Supongamos que tenemos la solución óptima, y que en esta solución, la primera detención es a los m kilómetros ($m \leq n$); en este punto, que llamaremos R, llenamos el estanque. El resto del recorrido tiene que ser una solución óptima (minimiza el número de detenciones) al problema de viajar desde R (m kilómetros al sur de Santiago) hasta Pueto Montt. La razón es que si no lo fuera, entonces podríamos encontrar otra secuencia de detenciones para viajar desde R hasta Puerto Montt que tuviera menos detenciones, y podríamos usar esta secuencia en el viaje desde Santiago hasta Puerto Montt, después de parar en R, reduciendo el número total de detenciones para este viaje, contradiciendo así la suposición de que la solución que tenemos es óptima.

b) Plantea una solución recursiva.

Dada la propiedad de subestructura óptima, podemos plantear la solución como sigue: Elegir de la mejor manera la próxima estación, y, de allí, encontrar una solución óptima al problema de viajar desde esa estación hasta Pueto Montt.

c) Justifica que en cualquier etapa de la recursión, una de las elecciones óptimas es la *elección codiciosa*.

La elección codiciosa es recorrer la mayor distancia que podamos antes de parar en una estación de bencina (y allí llenar nuevamente el estanque).

Supongamos que acabamos de reanudar el viaje depués de detenernos para llenar el estanque. Supongamos también que las siguientes estaciones son S, …, T, …, U, en orden de cercanía a nuestra posición actual, y que T es la más lejana que no está a más de n kilómetros de distancia. Y supongamos que en una solución óptima la próxima detención es en la estación S (más cercana que T).

Es fácil ver que en esta solución óptima podemos cambiar la elección de detenernos en S por la de detenernos en T, sin aumentar el número total de detenciones: si en la solución óptima, la próxima detención más allá de T es U, a la cual pudimos llegar desde S, entonces también podemos llegar a U desde T, ya que T está más cerca de U que S; y podemos llegar desde nuestra posición actual a T sin detenernos antes, ya que la distancia a T no es mayor que n kilómetros.

2013-2-I3-P1–Propuesta de algoritmo codicioso (1/1)

Estructuras de Datos y Algoritmos – IIC2133

I3

8 noviembre 2013

1) Se tiene que programar un conjunto de n charlas en varias salas. Cada charla ch_i tiene una hora de inicio s_i y una hora de término f_i , y puede darse en cualquier sala. Queremos programar las charlas de manera de usar el ***menor número posible de salas***.

a) [**2 pts.**] Da un ejemplo que muestre que la solución "obvia" de usar repetidamente el algoritmo codicioso `seleccion` visto en clase ***no funciona***.

b) [**4 pts.**] Da un ***algoritmo codicioso*** de tiempo $O(n + n \log n)$ para determinar cuál charla debería usar cuál sala: parte por ordenar todas las horas de inicio y todas las horas de término en una misma lista, y luego procesa esta lista en orden. Tu descripción puede ser en prosa, pero debe ser clara y precisa; numera pasos, identifica casos.

Respuesta:

a) Si las charlas son [1, 5), [3, 7), [6, 13) y [9, 12), `seleccion` toma primero [1, 5) y luego [9, 12), para una misma sala. Como [3, 7) y [6, 13) no son compatibles entre ellas (se traslapan), se necesitan dos salas más, para un total de 3 salas. Sin embargo, se podría haber asignado [1, 5) y [6, 13) a una misma sala, y así quedarían [3, 7) y [9, 12) en otra, para un total de sólo 2 salas.

b) Supongamos que las salas son A, B, C, …, y están en una lista ligada ls de salas disponibles. Primero, ordenamos todas las horas en tiempo $O(n \log n)$; en el caso del ejemplo anterior, las horas ordenadas quedan {1, 3, 5, 6, 7, 9, 12, 13}. Luego, procesamos estas horas así [solo para ayudar a entender el algoritmo + estructura de datos que aparece más abajo]:

– Tomamos la hora 1, inicio de la charla [1, 5), y asignamos la sala A a esta charla, sacándola de ls.
– Tomamos la hora 3, inicio de la charla [3, 7), y asignamos la sala B a esta charla, sacándola de ls.
– Tomamos la hora 5, término de la charla [1, 5), y devolvemos la sala A al comienzo de ls.
– Tomamos la hora 6, inicio de la charla [6, 13), y asignamos la sala A a esta charla, sacándola de ls.
– Tomamos la hora 7, término de la charla [3, 7), y devolvemos la sala B al comienzo de ls.
– Tomamos la hora 9, inicio de la charla [9, 12), y asignamos la sala B a esta charla, sacándola de ls.
– Tomamos la hora 12, término de la charla [9, 12), y devolvemos la sala B al comienzo de ls.
– Tomamos la hora 13, término de la charla [6, 13), y devolvemos la sala A al comienzo de ls.

Es decir, para cada hora, si la hora corresponde al inicio de una charla, tomamos la sala que está al ***comienzo*** de la lista ls y se la asignamos a la charla; si la hora corresponde al término de una charla, devolvemos la sala asignada a esa charla al ***comienzo*** de la lista ls. Esta iteración toma tiempo $O(n)$.

Este algoritmo es, evidentemente, codicioso. Además, produce una solución óptima, porque, cuando asigna una sala a una charla, asigna una que nunca ha sido ocupada ***solo si todas las que han sido ocupadas aún están ocupadas***: las salas se sacan del comienzo de la lista y se devuelven al comienzo de la lista.

2013-2-I3-P3–Algoritmo Prim (1/1)

3) Considera el algoritmo de Prim, que se muestra, para determinar un árbol de cobertura de costo mínimo (MST) para un grafo no direccional con aristas con costos. ***Deduce paso a paso la complejidad del algoritmo***, en notación O( ), en función del número de vértices, V, y del número de aristas, E, del grafo; explicita las suposiciones que hagas sobre las estructuras de datos usadas en el algoritmo

```
void prim(Vertex r)
    Queue q = new Queue(V)
    for ( each u in q ) u.key = ∞
    r.key = 0; π[r] = null
    while ( !q.empty() )
        Vertex u = q.xMin()
        for ( each v in α[u] )
            if ( v ∈ q ∧ w(u,v) < v.key )
                π[v] = u
                v.key = w(u,v)
```

Respuesta:

Suponiendo, como vimos en clase, que implementamos la cola priorizada **`q`** como un min-heap, las líneas antes del **`while`** corresponden a la construcción inicial del heap, que toma tiempo $O(V \log V)$: hay que poner V datos en el heap, y poner cada uno toma tiempo $O(\log V)$ —aunque más estrictamente se puede demostrar que el tiempo total es $O(V)$.

A este tiempo, hay que sumar el tiempo que toma el **`while`**.

En cada iteración, se saca un dato (el que tiene la menor clave y por lo tanto está en la primera posición de la cola) del heap (**`q.xMin()`**) y no se inserta ninguno, por lo que en total se ejecutan $|V|$ iteraciones; cada ejecución de **`xMin()`** toma tiempo $O(\log V)$, ya que hay que restaurar el heap cada vez, por lo que el tiempo total para las $|V|$ llamadas a **`xMin()`** es $O(V \log V)$.

Además, en cada iteración se ejecuta un **`for`**, que lo que hace es revisar cada uno de los vértices adyacentes al vértice **`u`** que se acaba de sacar del heap, o, equivalentemente, revisar cada una de las aristas conectadas al vértice **`u`**. Como finalmente se sacan todos los vértices del heap, entonces la totalidad de los **`for`** dentro del **`while`** lo que hace es revisar todas las aristas del grafo, dos veces cada una; es decir, el **`for`** ejecuta $2|E|$ veces en total. Como cada vez potencialmente se actualiza la clave de un vértice en el heap (**`v.key = w(u,v)`**), lo que toma $O(\log V)$, las $2|E|$ ejecuciones del **`for`** toman en total tiempo $O(E \log V)$. (La condición **`v ∈ q`** se puede implementar eficientemente en tiempo $O(1)$ agregando un bit a cada vértice.)

Así, el **`while`** en total toma tiempo $O(V \log V + E \log V) = O(E \log V)$.

Por lo tanto, el algoritmo toma en total tiempo $O(V) + O(E \log V) = O(E \log V)$.

2013-2-Ex-C–Árbol de cobertura mínimo, Kruskal (1/1)

Reemplazo de I3

C) Con respecto a los árboles de cobertura de costo mínimo (MST),

a) Demuestra o refuta la siguiente afirmación: Un árbol de extensión de costo mínimo incluye, para cada vértice v, la arista de costo mínimo entre las aristas incidentes en v.

La afirmación es verdadera; la demostramos por contradicción. Sea t el árbol; sea (v, u) la arista de costo mínimo incidente en v; y supongamos que (v, u) no está en t. Si incluimos (v, u) en t, formamos un ciclo; éste debe incluir una arista (v, x), $x \neq u$, cuyo costo es mayor que el costo de (v, u): $w(v, u) < w(v, x)$. Si sacamos (v, x) del ciclo volvemos a tener un MST cuya única diferencia con t es que la arista (v, u) reemplaza a la arista (v, x). Como $w(v, u) < w(v, x)$, este árbol tiene costo menor que t, contradiciendo la suposición de que t es un árbol de extensión de costo mínimo.

b) Si todos los costos de las aristas son números enteros en el rango 1 a W, en que W es una constante, ¿qué tan rápido, en notación O( ), se puede hacer que corra el algoritmo de Kruskal? Toma en cuenta que el algoritmo incluye una inicialización, una ordenación, y finalmente la ejecución del algoritmo propiamente tal.

*Kruskal toma O(V) para inicialización, O(E log E) para ordenar las aristas, y O(E α(V)) para las operaciones de conjuntos disjuntos (la ejecución del algoritmo propiamente tal). Por lo tanto, Kruskal es O(E log E). Ahora, bajo la supsición dada y usando countingSort, podemos ordenar las aristas en O(W + E) = O(E) —**ya que W es constante**. De modo que ahora el algoritmo toma O(E α(V)).*

c) Si todos los costos de las aristas son distintos, muestra que el MST es único.

Como todos los costos de las aristas son distintos, para cada corte hay una única arista liviana. A partir de las diapositivas #47 a 49, deducimos que el MST es único.

2013-2-Ex-D–Dijkstra (1/1)

D) Considera el ***algoritmo de Dijkstra***, que determina las rutas más cortas desde s a cada uno de los vértices v de un grafo direccional $G = (V, E)$ en que los costos de las aristas son ≥ 0:

```
void Dijkstra(Vertex s)
    Init(s); S = ∅;
    Queue q = new Queue(V)
    while ( !q.empty() )
        Vertex u = q.xMin()
        S = S ∪ {u}
        for ( each v in α[u] )
            reduce(u,v)
```

Aplícalo al grafo representado por la siguiente matriz de adyacencias, para determinar las longitudes de las rutas más cortas desde el vértice d (las casillas vacías representan ∞):

	a	b	c	d	e	f	g	h	i	j
a	0				1			10		
b		0	2							
c			0							
d	4			0				1		
e					0	3				
f		1	3			0	7		1	
g							0			
h					5			0	9	
i									0	2
j							1			0

Muestra los valores del vector **d** después de cada iteración.

d = [∞ ∞ ∞ 0 ∞ ∞ ∞ ∞ ∞ ∞] = inicial
d = [4 ∞ ∞ 0 ∞ ∞ ∞ 1 ∞ ∞] = después que sale $u = d$
d = [4 ∞ ∞ 0 6 ∞ ∞ 1 10 ∞] = después que sale $u = h$
d = [4 ∞ ∞ 0 5 ∞ ∞ 1 10 ∞] = después que sale $u = a$
d = [4 ∞ ∞ 0 5 8 ∞ 1 10 ∞] = después que sale $u = e$
d = [4 9 11 0 5 8 15 1 9 ∞] = después que sale $u = f$
d = [4 9 11 0 5 8 15 1 9 ∞] = después que sale $u = b$
d = [4 9 11 0 5 8 15 1 9 11] = después que sale $u = i$
d = [4 9 11 0 5 8 15 1 9 11] = después que sale $u = c$
d = [4 9 11 0 5 8 12 1 9 11] = después que sale $u = j$
d = [4 9 11 0 5 8 12 1 9 11] = después que sale $u = g$

2013-1-I3-P1–Demostración de de solución codiciosa (1/2)

Estructuras de Datos y Algoritmos – IIC2133

I3

14 junio 2013

1) En clase resolvimos el problema de optimización de programar tareas con plazos: Dado un conjunto de tareas, c/u con un plazo y una ganancia, encontrar un subconjunto de tareas que pueden ser hechas dentro de sus plazos (subconjunto *factible*) y que maximiza la suma de las ganancias (subconjunto *óptimo*); las tareas sólo pueden ser hechas usando una única máquina por una unidad de tiempo —por lo tanto, sólo se puede hacer una tarea a la vez— y las ganancias se obtienen sólo si las tareas son hechas dentro de sus plazos.

Demostramos que el problema puede resolverse si ordenamos las tareas de mayor a menor ganancia, y empleamos la estrategia codiciosa de agregar a la solución la próxima tarea que siendo factible hace aumentar más la ganancia acumulada. Esta solución requiere poder determinar la factibilidad de un subconjunto de tareas, lo que resolvimos demostrando que basta probar la factibilidad de una permutación del subconjunto en que las tareas estén ordenadas crecientemente por plazos.

a) ¿Qué complejidad tiene este algoritmo? Justifica.

Resp. $O(n^2)$. Para cada una de las n tareas —ya ordenadas de mayor a menor ganancia en tiempo $O(n\log n)$— hay que ver si es compatible con las otras tareas que ya están en la solución. Esto requiere ordenar las m tareas en la solución por plazos —en tiempo $O(m)$, usando ordenación por inserción en un arreglo ordenado— y luego verificar la factibilidad del conjunto, es decir, determinar si cada tarea se está haciendo dentro de su plazo: como el conjunto tiene m tareas, esta verificación toma tiempo $O(m)$. Como este tiempo $O(m)$ se repite para cada una de las n tareas, el tiempo total es $O(nm)$, que en el peor caso es $O(n^2)$.

Es posible usar otro método para determinar la factibilidad de un subconjunto de tareas. Si J es un subconjunto factible de tareas, entonces podemos asignar los tiempos de procesamiento de cada tarea de la siguiente manera: si para la tarea i aún no hemos asignado un tiempo de procesamiento, entonces asignémosle el *slot* $[k–1, k]$, en que k es el mayor entero menor o igual que el plazo de i y el *slot* $[k–1, k]$ está vacío —es decir, postergamos la tarea i lo más que podemos. Así, al construir J de a una tarea a la vez, no movemos las tareas que ya tienen sus *slots* asignados para acomodar una nueva tarea: si para esta nueva tarea no encontramos un k como el que definimos antes, entonces la tarea no puede programarse.

b) Demuestra esta última afirmación.

Resp. La única forma en que la tarea t, con plazo d_t, no pueda programarse es que todos los slots [0, 1], [1, 2], …, $[d_{t-1}, d_t]$ estén ocupados. En este caso, no hay ninguna forma de hacer espacio para la tarea t, ya que todas las tareas programadas en estos slots están postergadas lo más posible, considerando incluso slots posteriores a $[d_{t-1}, d_t]$: si el slot $[k–1, k]$, con $k > d_t$, estuviera vacío, y la tarea s programada en el slot $[j–1, j]$, con $j < d_t$, se hubiera podido programar en el slot $[k–1, k]$, entonces la tarea s se hubiera programado en este slot al considerarla por primera vez.

2013-1-I3-P1–Demostración de de solución codiciosa (2/2)

c) Explica cómo se puede implementar esta nueva forma de programar las tareas usando una estructura de conjuntos disjuntos.

Resp. Llamemos k al slot $[k–1, k]$; y sea n_k el mayor entero tal que $n_k \leq k$ y el slot n_k está vacío. Dos slots p y q están en el mismo conjunto si $n_p = n_q$; claramente, si $p < q$, entonces p, p+1, p+2, …, q están todos en el mismo conjunto; el valor n_p es un valor importante para el conjunto, por lo que lo almacenamos en un campo f en el representante del conjunto. Para programar una tarea con plazo d, buscamos el (representante del) conjunto al que pertenece el slot d, y obtenemos su valor f, que es el slot más cercano (y no mayor que d) disponible; luego, unimos este conjunto con el conjunto correspondiente al slot f–1.

d) ¿Qué complejidad tiene este nuevo algoritmo? Justifica.

Resp. La complejidad es $O(n\alpha(2n,n))$, si usamos la estructura más eficiente conocida para manejar conjuntos disjuntos: inicialmente, hay n conjuntos disjuntos, y luego realizamos n *find*'s (uno por cada tarea) y a lo más n *union*'s.

2013-1-I3-P3–Subestructura óptima, subproblemas traslapados (1/1)

3) Hay que programar un conjunto de n tareas en una máquina que sólo puede realizar una tarea a la vez. Cada tarea a_i tiene una hora de inicio s_i , una hora de término f_i , y un valor v_i . Sea A un subconjunto de tareas mutuamente compatibles (para cualquier par de tareas de A, la hora de inicio de una de ellas es mayor que la hora de término de la otra); el *valor de A* es la suma de los valores de las tareas de A. El objetivo es encontrar un subconjunto A de valor máximo, es decir, maximizar la suma de los valores de las tareas programadas (y no necesariamente el número de tareas programadas).

En particular, sea C_{ij} el conjunto de tareas que empiezan después que la tarea a_i termina y que terminan antes que empiece la tarea a_j . Sea A_{ij} una solución óptima a C_{ij} , es decir, A_{ij} es un subconjunto de tareas mutuamente compatibles de C_{ij} que tiene valor máximo.

a) Supongamos que A_{ij} incluye la actividad a_k . Demuestra que este problema tiene la propiedad de *subestructura óptima*: una solución óptima al problema contiene soluciones óptimas a subproblemas.

Resp. Podemos descomponer la solución óptima A_{ij} como $A_{ik} \cup \{a_k\} \cup A_{kj}$; es decir, la actividades que, estando en A_{ij} , terminan antes que empiece a_k , junto a las actividades que, estando en A_{ij} , empiezan después que termina a_k , junto a a_k . Así, el valor de la solución óptima A_{ij} es igual al valor de A_{ik} más el valor de A_{kj} más v_k . Siguiendo un razonamiento similar a los vistos en clase para otros problemas, concluimos que A_{ik} debe ser una solución óptima al problema definido por C_{ik} , y análogamente para A_{kj} y C_{kj} .

b) Sea $val[i, j]$ el valor de una solución óptima para el conjunto C_{ij}. Demuestra que este problema tiene la propiedad de *subproblemas traslapados*: si usamos un algoritmo recursivo para resolver el problema, este tiene que resolver los mismos subproblemas repetidamente.

Resp. De acuerdo con a), $val[i, j] = val[i, k] + val[k, j] + v_k$. Pero en realidad no sabemos que la solución óptima al problema definido por C_{ij} incluye la actividad a_k , por lo que debemos mirar todas las actividades en C_{ij} para determinar cuál elegir; es decir,

$$val[i, j] = 0, \qquad \text{si } C_{ij} = \varnothing$$
$$val[i, j] = \max_{a_k \in S_{ij}} \{ val[i, k] + val[k, j] + v_k \}, \qquad \text{si } C_{ij} \neq \varnothing$$

Aquí aparecen los subproblemas traslapados.

2013-1-I3-P4-a–Dijkstra (1/1)

4) Con respecto a las rutas más cortas en grafos direccionales:

a) Un compañero de curso te dice que implementó el algoritmo de Dijkstra. El programa produce las distancias d y los padres π para cada vértice del grafo. Da un algoritmo de tiempo $O(V+E)$ para verificar el resultado del programa de tu compañero. Supón que todas las aristas tienen costos no negativos. En particular,

i) ¿qué debes verificar con respecto al vértice de parida s?

ii) ¿qué debes verificar con respecto a los otros vértices?

iii) ¿cómo te puedes asegurar que el programa de tu compañero efectivamente encontró las rutas más cortas desde s?

Resp.

i) Para s, hay que verificar $s.d = 0$ y $s.\pi = nil$.

ii) Para los otros vértices v, hay que verificar $v.d = v.\pi.d + w(v.\pi, v)$; o bien $v.d = \infty$ si y solo si $v.\pi = nil$.

Si cualquiera de las verificaciones anteriores, **i)** o **ii)**, falla, entonces el resultado del programa del compañero es incorrecto.

iii) De lo contrario, todavía hay que asegurarse que haya encontrado las rutas más cortas desde s. Para esto, basta con aplicar `reduce` a cada arista una vez: si algún $v.d$ cambia, entonces el resultado es incorrecto; de lo contrario, es correcto.

2013-1-Ex-C–Árbol de cobertura mínimo, Kruskal (1/1)

C) Con respecto a los árboles de cobertura de costo mínimo (MST),

9) Si todos los costos de las aristas son números enteros en el rango 1 a $|V|$, ¿qué tan rápido, en notación O( ), se puede hacer que corra el algoritmo de Kruskal? Toma en cuenta que el algoritmo incluye una inicialización, una ordenación, y finalmente la ejecución del algoritmo propiamente tal.

Kruskal toma O(V) para inicialización, O(E log E) para ordenar las aristas, y O(E α(V)) para las operaciones de conjuntos disjuntos (la ejecución del algoritmo propiamente tal); por lo tanto, Kruskal es O(E log E). Ahora, bajo el supuesto de arriba y usando countingSort, podemos ordenar las aristas en O(V + E) = O(E) —**ya que V = O(E)**. De modo que ahora Kruskal es O(E α(V)).

10) Si todos los costos de las aristas son números enteros en el rango 1 a W, en que W es una constante, ¿qué tan rápido, en notación O( ), se puede hacer que corra el algoritmo de Kruskal? Toma en cuenta que el algoritmo incluye una inicialización, una ordenación, y finalmente la ejecución del algoritmo propiamente tal.

Kruskal toma O(V) para inicialización, O(E log E) para ordenar las aristas, y O(E α(V)) para las operaciones de conjuntos disjuntos (la ejecución del algoritmo propiamente tal); por lo tanto, Kruskal es O(E log E). Ahora, bajo el supuesto de arriba y usando countingSort, podemos ordenar las aristas en O(W + E) = O(E) —**ya que W es constante**. De modo que ahora Kruskal es O(E α(V)).

11) Si todos los costos de las aristas son distintos, muestra que el MST es único.

Como todos los costos de las aristas son distintos, para cada corte hay una única arista liviana. A partir de las diapositivas #47 a 49, deducimos que el MST es único.

Suponga que el MST no es único, es decir, existen dos MSTs T y T' con T distinto de T'. Considere "e" la arista de menor peso de T que no aparece en T'. A su vez, considere e' la arista de menor peso de T' que no aparece en T. Sin pérdida de generalidad, suponga que "e" tiene menor peso que e'. Entonces, necesariamente se tiene que T' unido con {e} forma un ciclo C. Considere S=C-E(T) el conjunto de todas las aristas del ciclo que no aparecen en T. Notar que todas las aristas de S tienen estrictamente mayor peso que "e", pues todas las aristas son distintas y "e" es la arista con menor peso que no aparecen en ambos árboles T y T'. Sea r una arista de S y se tendrá que T" = T' U {e} – {r} forma un árbol de cobertura. Dado que se intercambió una arista más pesada por otra de menor peso, se tiene que T" tiene menor peso que T', lo que contradice con el hecho de que T' es MST. Luego, debe ocurrir que si todos los costos de las aristas son distintos, el MST es único.

12) Si todos los costos de las aristas son distintos, muestra que el ***segundo mejor*** MST puede no ser único.

Basta con dar un ejemplo.

D) Considera el problema de determinar las **rutas más cortas entre todos los pares de vértices** de un grafo direccional G = (V, E) con costos en las aristas.

13) Enuncia las dos propiedades características de los problemas de optimización que pueden ser resueltos mediante programación dinámica.

Las dos propiedades características son: Propiedad de Subestructura óptima y Propiedad de subproblemas traslapados.

- Subestructura óptima: La solución óptima al problema original puede ser construida con soluciones óptimas de subproblemas.
- Subproblemas traslapados: Si uno utiliza un algoritmo recursivo para resolver el problema, entonces existirá al menos un subproblema que será resuelto varias veces.

2012-1-I3-P2-a–Identificación de solución codiciosa (1/2)

IIC2133 - Estructuras de datos y algoritmos I-2012 – Interrogación 3

Pregunta 2 – Técnicas de programación (20 pts)

A continuación se presentan 4 problemas, cada uno con una solución propuesta. Para cada caso, escribe qué tipo de solución es y por qué, entre: Dividir y conquistar (DyC), Algoritmo codicioso, Programación Dinámica o ninguno de los anteriores. Nota que las soluciones propuestas podrían ser incorrectas, pero este aspecto no es relevante para la evaluación que debes hacer.

a) (5 pts) El gato enojado >(

Tienes un gatito que está durmiendo plácidamente en medio de la sala. Tu quieres cruzar la sala, pasando frente al gatito. Pero ¡cuidado! Si despiertas al gato, este estará muy enojado, así que debes ser muy silencioso. Para mala suerte, el piso de la sala rechina y cada vez que das un paso el gato despierta ligeramente. Pero si esperas un poco, el gato vuelve a dormir profundamente.

Para ser exactos, para cruzar la sala necesitas dar a pasos ($0 < a < 100$). El gato duerme inicialmente con profundidad 1 y si en algún momento la profundidad de sueño es inferior a w ($0 < w < 1$), el gato despertará. Dependiendo de donde estés, cada vez que das un paso, la profundidad del sueño decae en s_i ($0 < s_i < 1$), para s_0, s_1, … s_{a-1}. Finalmente, por cada tiempo k ($0 < k < 10$) que esperes, la profundidad del sueño subirá en 0.1 .

¿Cuál es el menor tiempo de espera en que puedes cruzar la sala sin despertar al gatito?

El input consiste en una primera línea con los valores a y w. Luego una línea con los a valores s (s_0, s_1, … s_{a-1}). Por último hay una línea con el valor k. Los valores a y k son números enteros, mientras los demás números son decimales con 1 dígito de precisión.

El output consiste en un número entero con el menor tiempo de espera o -1 si no es posible cruzar la sala sin despertar al gato.

Ejemplo:

Input

```
10 0.3
0.2 0.4 0.3 0.5 0.6 0.4 0.3 0.1 0.2 0.1
1
```

Output

```
24
```

Solución propuesta

En una variable vamos a mantener el nivel de sueño del gato (p). Luego, para cada paso entre 0 y a -1 primero evaluamos si con el nivel de sueño actual podemos dar el paso sin despertar al gato, damos el paso y actualizamos el nivel de sueño como $p' = p - s_i$. La evaluación de si podemos dar el gato en el fondo es evaluar si $p - s_i >= w$. Si no es posible, entonces esperaremos el mínimo posible para poder dar el paso, es decir j*k unidades de

2012-1-I3-P2-a–Identificación de solución codiciosa (2/2)

tiempo con el menor j tal que p + (0.1*j*k) – s_i >= w. Esta condición de mínimo se cumple en la igualdad, es decir, esperaremos j*k = (w + s_i – p)/0.1. Lo que si, antes es necesario un chequeo adicional: si para poder dar el paso hubiera que hacer que la profundidad de sueño sea superior a 1 (es decir, si p + (0.1*j*k) > 1), entonces no podemos dar el paso y la respuesta será un -1. Si logramos dar el paso, agregar j*k a la suma acumulada de tiempos esperados.

Finalmente, si logramos llegar hasta el final, entregar la suma acumulada de tiempos esperados.

Este algoritmo es greedy que supone que encontrará el óptimo al tratar siempre de esperar lo menos posible, justo lo suficiente para dar el siguiente paso. Si uno compara con el método general, considera que la solución al problema consiste en el conjunto de tiempos de espera, la selección de cada turno es el tiempo mínimo de espera, la decisión de factibilidad es ver si no se supera el nivel de sueño "1" para poder dar el paso y agregarlo a la solución es la suma (este último análisis de comparación con el método general no es necesario, pero da una evidencia irrefutable de que es greedy).

2012-1-I3-P2-b–Identificación de solución codiciosa (1/1)

b) (5 pts) Elegir las frutas más pesadas del canasto*

Se tiene un canasto de m frutas con n tipos distintos (m > n). Calcula la suma de los pesos de las frutas, al elegir una fruta de cada tipo (siempre la más pesada de su tipo).

El input consiste en n (tipos de frutas), luego m (total de frutas), luego una lista de m pesos de las frutas y finalmente una lista con los m tipos de cada fruta (números entre 0 y n-1).

Ejemplo.

Input:

```
3 5
1 8 5 20 2
0 1 0 1 2
```

Output:

```
27
```

Solución propuesta

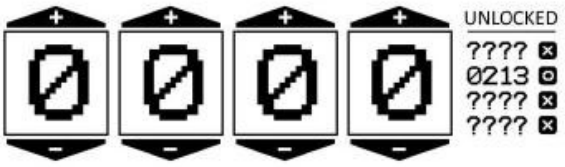
Se divide la lista de frutas en n listas, una por cada tipo. Luego en cada lista se elige la fruta de mayor peso y con las soluciones parciales para cada tipo, se suman los pesos máximos de cada tipo, obteniendo el peso total.

Este algoritmo parece ser dividir y conquistar porque al principio se dividen las frutas en n listas, pero en realidad no lo es. Aquí alguien podría confundirse y pensar que si ordena en la segunda etapa con mergesort (que si es DyC) o con quicksort, entonces eso hace que todo el algoritmo sea DyC, pero se equivoca. DyC requiere que el problema se reduzca a instancias más pequeñas del mismo problema, lo cual no ocurre en este caso.

2012-1-I3-P4–Planteamiento de problema, Árbol de cobertura mínimo (1/2)

Pregunta 4 – Bloqueo anti fuerza bruta* (20 pts)

Recientemente ha habido un serio problema con las cajas fuertes Panda: ¡varias cajas fuertes han sido robadas! Estas cajas fuertes están usando el viejo sistema de cerraduras de combinaciones de 4 dígitos (sólo necesitas girar el dígito, ya sea para arriba o para abajo hasta que los cuatro coinciden con la clave). Cada dígito está hecho para girar de 0 a 9. Girar hacia arriba en el 9 hace que el dígito se convierta en 0, y girar hacia abajo el 0 hace que el dígito se convierta en 0. De momento que sólo hay 10.000 claves posibles, desde 0000 hasta 9999, cualquiera puede probar todas las combinaciones hasta que la caja se haya desbloqueado.



Lo hecho, hecho está. Pero con el fin de retrasar futuros ataques de ladrones, la Agencia de Seguridad Panda (ASP) ha diseñado una cerradura más segura con múltiples claves. En lugar de usar sólo una combinación como clave, la cerradura ahora tiene hasta N claves las cuales deben ser todas desbloqueadas antes que la caja fuerte puede abrirse. Esta cerradura funciona así:

- Inicialmente los dígitos están en 0000
- Las claves pueden ser desbloqueadas en cualquier orden, poniendo los dígitos en la cerradura para coincidir con la clave deseada y luego presionando el botón DESBLOQUEAR.
- Un botón mágico, SALTAR, puede convertir los dígitos en cualquier clave ya desbloqueada sin girar ningún dígito.
- La caja fuerte se desbloqueará si y sólo si todas las claves son desbloqueadas en un total mínimo de giros de dígitos, excluyendo cualquier cambio hecho con el botón SALTAR (si, esta característica es una de las más geniales).

2012-1-I3-P4–Planteamiento de problema, Árbol de cobertura mínimo (2/2)

- Si el número de giros de dígitos es excedido, los dígitos serán regresados a 0000 y todas las claves se bloquearán nuevamente. En otras palabras, el estado de la cerradura se reiniciará si el desbloqueo falla.

ASP está bien confiado en que este nuevo sistema retrasará un intento de violación por fuerza bruta, dándoles suficiente tiempo como para identificar y capturar los ladrones. Con el fin de determinar el número mínimo de giros de dígitos requeridos, ASP quiere que escribas un programa. Dadas todas las claves, calcula el mínimo número de giros necesarios para desbloquear la caja fuerte.

Input

La primera línea de input contiene un entero T, el número de casos que siguen. Cada caso comienza con un entero N (1 <= N <= 500), el número de claves. La siguiente línea contiene N números de exactamente 4 dígitos (con ceros a la izquierda) representando las claves del desbloqueo.

Output

Para cada caso, imprimir en una única línea el mínimo número de giros requeridos para desbloquear todas las claves.

Input de ejemplo	Output de ejemplo
`4`	`16`
`2 1155 2211`	`20`
`3 1111 1155 5511`	`26`
`3 1234 5678 9090`	`17`
`4 2145 0213 9113 8113`	

Explicación del segundo caso

- Convierte `0000` en `1111`, giros: 4
- Convierte `1111` en `1155`, giros: 8
- Saltar `1155` en `1111`, podemos hacer esto porque `1111` había sido desbloqueado previamente.
- Convertir `1111` en `5511`, giros: 8

Giros totales = 4 + 8 + 8 = 20

Primero hay que representar el problema como un grafo, en que cada vértice es una de las claves y la distancia entre vértices está dada por el número de giros necesarios.

Luego, hay que calcular el número de giros entre el "0000" y las demás claves, y anexar "0000" como un vértice más, que está únicamente conectado a la clave de distancia mínima.

2011-2-I2-P4–Kruskal (1/1)

4. Considera el algoritmo de Kruskal para encontrar un árbol de extensión de costo mínimo (MST) para un grafo direccional $G = (V, E)$ en que cada arista tiene asociado un costo. Si los costos de todas las aristas de G son números enteros en el rango de 1 a $|V|$, ¿qué tan rápido se puede hacer la ejecución del algoritmo de Kruskal? Recuerda que el algoritmo de Kruskal toma tiempo $O(V)$ para inicialización, $O(E \log E)$ para ordenar las aristas, y $O(E\ \alpha(V))$ para las operaciones de conjuntos disjuntos.

Respuesta. Primero, el tiempo total del algoritmo de Kruskal 'básico' es $O(V) + O(E \log E) + O(E\ \alpha(V)) = O(E \log E)$. Si ahora los costos de todas las aristas son números enteros en el rango de 1 a $|V|$, entonces podemos ordenarlas en tiempo $O(V + E)$ —en lugar de $O(E \log E)$— usando *countingSort*; además, como G es conexo, $V = O(E)$, y el tiempo de ordenación se reduce a $O(E)$. Esto da un tiempo total para el algoritmo de $O(V) + O(E) + O(\mathrm{E}\ \alpha(V)) = \boldsymbol{O(E\ \alpha(V))}$ —de nuevo, ya que $V = O(E)$. (Es decir, el término dominante cambia de ser el tiempo que toma ordenar las aristas al tiempo que toma procesarlas una vez ordenadas. Además, sólo la ordenación de las aristas se hace más rápido, porque ninguna otra parte del algoritmo usa los costos de las aristas.)

2011-2-I3-P1–Subestructura óptima, solución recursiva (1/1)

Estructuras de Datos y Algoritmos – IIC2133

I3 - pauta

2 noviembre 2011

1. Tú quieres conducir un auto desde Santiago hasta Pueto Montt. Cuando el estanque de bencina del auto está lleno, te permite viajar n kilómetros. Tú tienes un mapa con las distancias entre estaciones de bencina en la ruta. Tu propósito es hacer el menor número posible de detenciones en el viaje.

a) Muestra que este problema tiene subestructura óptima.

(Suponemos que partimos con el estanque lleno y que las distancias entre pares consecutivos de estaciones de bencina son todas $\leq n$.)

Supongamos que tenemos la solución óptima, y que en esta solución, la primera detención es a los m kilómetros ($m \leq n$); en este punto, que llamaremos R, llenamos el estanque. El resto del recorrido tiene que ser una solución óptima (minimiza el número de detenciones) al problema de viajar desde R (m kilómetros al sur de Santiago) hasta Puerto Montt. La razón es que si no lo fuera, entonces podríamos encontrar otra secuencia de detenciones para viajar desde R hasta Puerto Montt que tuviera menos detenciones, y podríamos usar esta secuencia en el viaje desde Santiago hasta Puerto Montt, después de parar en R, reduciendo el número total de detenciones para este viaje, contradiciendo así la suposición de que la solución que tenemos es óptima.

b) Plantea una solución recursiva.

Dada la propiedad de subestructura óptima, podemos plantear la solución como sigue: Elegir de la mejor manera la próxima estación, y, de allí, encontrar una solución óptima al problema de viajar desde esa estación hasta Puerto Montt.

c) Justifica que en cualquier etapa de la recursión, una de las elecciones óptimas es la elección codiciosa.

La elección codiciosa es recorrer la mayor distancia que podamos antes de parar en una estación de bencina (y allí llenar nuevamente el estanque).

Supongamos que acabamos de reanudar el viaje depués de detenernos para llenar el estanque. Supongamos también que las siguientes estaciones son S, …, T, …, U, en orden de cercanía a nuestra posición actual, y que T es la más lejana que no está a más de n kilómetros de distancia. Y supongamos que en una solución óptima la próxima detención es en la estación S (más cercana que T).

Es fácil ver que en esta solución óptima podemos cambiar la elección de detenernos en S por la de detenernos en T, sin aumentar el número total de detenciones: si en la solución óptima, la próxima detención más allá de T es U, a la cual pudimos llegar desde S, entonces también podemos llegar a U desde T, ya que T está más cerca de U que S; y podemos llegar desde nuestra posición actual a T sin detenernos antes, ya que la distancia a T no es mayor que n kilómetros.

2011-2-I3-P2–Subestructura óptima, solución recursiva, programación dinámica (1/1)

2. Siguiendo con el viaje de Santiago a Puerto Montt, supón que tienes autorización para poner letreros con propaganda de tu negocio en el camino. Los puntos en los que puedes poner los letreros son x_1, x_2, …, x_n, que representan las distancias desde Santiago, y tu modelo de predicción te dice que si pusieras un letrero en el punto x_k ganarías $r_k > 0$. Por otro lado, la autoridad vial exige que no puede haber dos letreros de un mismo negocio a menos de 5 km entre ellos.

Tu problema es determinar donde poner letreros, de manera de maximizar tu ganancia. P.ej., si los puntos son $\{ x_1, x_2, x_3, x_4 \} = \{ 6, 7, 12, 14 \}$ y las ganancias son $\{ r_1, r_2, r_3, r_4 \} = \{ 5, 6, 5, 1 \}$, entonces te conviene poner los letreros en x_1 y x_3 para ganar 10.

a) Muestra que este problema tiene subestructura óptima.

Consideremos una solución óptima. Ésta puede incluir poner un letrero en x_n o no. Si no lo incluye, entonces la solución óptima es la misma que para los puntos x_1, …, x_{n-1}. Si lo incluye, entonces, si sacamos x_n de la solución óptima, los puntos que quedan son una solución óptima para el camino de Santiago hasta 5 km antes de x_n.

b) Plantea una solución recursiva.

Definamos $s(j)$ como el punto más al sur que está a más de 5 km (al norte) de x_j. Es decir, si decidimos que x_j está en la solución, entonces x_1, x_2, …, $x_{s(j)}$ son aún posibilidades válidas, pero $x_{s(j)+1}$, …, x_{j-1} no. Si opt(j) es la ganancia del subconjunto óptimo de sitios entre x_1, …, x_j, entonces, de a):

$\text{opt}(j) = \max\{ r_j + \text{opt}(s(j)), \text{opt}(j-1) \}$

c) Justifica que una buena forma de resolver el problema es mediante programación dinámica.

Al tratar de resolver el problema recursivamente, aplicando la fórmula recursiva anterior de manera "top-down", vamos a tener que resolver un mismo subproblema opt(j) muchas veces. En cambio, podemos resolver la fórmula recursiva de manera "bottom-up," para $j = 2, 3, \ldots n$, sabiendo que opt(0) = 0 y opt(1) = 1.

2010-2-I2-P3–Algoritmo Prim (1/1)

3. Considera el algoritmo de Prim, estudiado en clase, para encontrar un árbol de cobertura de costo mínimo para un grafo no direccional $G = (V, E)$, a partir del vértice r en E.

```
for ( cada vértice u en V ) u.key = ∞
r.key = 0
formar una cola Q con todos los vértices en V, priorizada según el campo key de cada vértice
π[r] = null
while ( !Q.empty( ) )
{
    u = Q.extractMin( ) —esta operación modifica la cola Q
    for ( cada vértice v en listadeAdyacencias[u] )
        if ( v ∈ Q ∧ costo(u,v) < v.key )
        {
            π[v] = u
            v.key = costo(u,v) —esta operación modifica la cola Q
        }
}
```

Recuerda que el desempeño de Prim depende de cómo se implementa la cola Q; p.ej., si Q es un heap binario, entonces Prim toma tiempo $O(E \log V)$: el *for* dentro del *while* mira cada arista de G y para cada una realiza una operación *decreaseKey* sobre un vértice (cuando actualiza *v.key*, en la última línea).

Si los costos de todas las aristas de G son números enteros entre 0 y una constante W (tal que es factible declarar un arreglo de tamaño W+1), describe una forma de implementar la cola Q, tal que Prim corra en tiempo $O(E)$, es decir, que cada operación *descreaseKey* tome tiempo O(1). [Recuerda que si W es constante, es decir, no depende ni de E ni de V, entonces recorrer un arreglo de tamaño W+1 toma tiempo $O(1)$.]

Respuesta:

[**2 pts.**] Implementamos Q como un arreglo Q[0], Q[1], …, Q[W], Q[W+1]. Cada elemento del arreglo es una lista doblemente ligada de vértices, en que Q[k] contiene todos los vértices cuyo *key* vale k; Q[W+1] contiene los vértices cuyo *key* vale infinito (∞).

[**2 pts.**] Entonces, *extractMin* consiste simplemente en recorrer Q buscando el primer elemento no vacío; por lo tanto, toma $O(W) = O(1)$.

[**2 pts.**] También, *decreaseKey* toma tiempo $O(1)$: basta sacar el vértice de la lista en que está (la lista tiene que ser doblemente ligada para que sacar un vértice tome $O(1)$) y agregarlo al comienzo de la nueva lista que le corresponda según su nuevo valor de *key*.

2010-2-I3-P1–Dijkstra (1/1)

Estructuras de Datos y Algoritmos – IIC2133
I3
15 de noviembre, 2010

1) Se tiene un grafo direccional en que las aristas que salen desde el vértice de partida s pueden tener costos negativos, todas las otras aristas tienen costos no negativos, y no hay ciclos con costo acumulado negativo. Prueba que, a pesar de lo que dijimos en clase, el ***algoritmo de Dijkstra*** encuentra correctamente las rutas más cortas desde s en este grafo.

Respuesta:

Dijkstra resuelve el problema porque lo costos negativos, de las aristas que salen de s, son los primeros que ve, y, por lo tanto, los toma en cuenta antes de tomar en cuenta cualquier otro costo no negativo.

2010-2-I3-P3–Creación de algoritmo codicioso (1/1)

3) Se tiene que programar un conjunto de n charlas en varias salas. Cada charla ch_i tiene una hora de inicio s_i y una hora de término f_i , y puede darse en cualquier sala. Queremos programar las charlas de manera de usar el menor número posible de salas.

a) [**1/3**] Da un ejemplo que muestre que la solución "obvia" de usar repetidamente el algoritmo codicioso `seleccion` visto en clase ***no funciona***.

b) [**2/3**] Da un ***algoritmo codicioso*** de tiempo $O(n + n \log n)$ para determinar cuál charla debería usar cuál sala: parte por ordenar todas las horas de inicio y todas las horas de término en una misma lista, y luego procesa esta lista en orden.

Respuesta:

a) [**1/3**] Si las charlas son [1, 5), [3, 7), [6, 13) y [9, 12), `seleccion` toma primero [1, 5) y luego [9, 12), para una misma sala. Como [3, 7) y [6, 13) no son compatibles entre ellas, se necesitan dos salas más, para un total de 3 salas.

Sin embargo, se podría haber asignado [1, 5) y [6, 13) a una misma sala, y así quedarían [3, 7) y [9, 12) en otra, para un total de 2 salas.

b) [**2/3**] Supongamos que las salas son A, B, C, …, y están en una lista ligada ls de salas disponibles. Primero, ordenamos todas las horas en tiempo $O(n \log n)$; en el caso del ejemplo anterior, las horas ordenadas quedan {1, 3, 5, 6, 7, 9, 12, 13}. Luego, procesamos estas horas así [solo para ayudar a entender el algoritmo + estructura de datos que aparece más abajo]:

– Tomamos la hora 1, inicio de la charla [1, 5), y asignamos la sala A a esta charla, sacándola de ls.
– Tomamos la hora 3, inicio de la charla [3, 7), y asignamos la sala B a esta charla, sacándola de ls.
– Tomamos la hora 5, término de la charla [1, 5), y devolvemos la sala A al comienzo de ls.
– Tomamos la hora 6, inicio de la charla [6, 13), y asignamos la sala A a esta charla, sacándola de ls.
– Tomamos la hora 7, término de la charla [3, 7), y devolvemos la sala B al comienzo de ls.
– Tomamos la hora 9, inicio de la charla [9, 12), y asignamos la sala B a esta charla, sacándola de ls.
– Tomamos la hora 12, término de la charla [9, 12), y devolvemos la sala B al comienzo de ls.
– Tomamos la hora 13, término de la charla [6, 13), y devolvemos la sala A al comienzo de ls.

Es decir, para cada hora, si la hora corresponde al inicio de una charla, tomamos la sala que está al ***comienzo*** de la lista ls y se la asignamos a la charla; si la hora corresponde al término de una charla, devolvemos la sala asignada a esa charla al ***comienzo*** de la lista ls. Esta iteración toma tiempo $O(n)$.

Este algoritmo es, evidentemente, codicioso. Además, produce una solución óptima, porque, cuando asigna una sala a una charla, asigna una que nunca ha sido ocupada ***solo si todas las que han sido ocupadas aún están ocupadas***: las salas se sacan del comienzo de la lista y se devuelven al comienzo de la lista.

2010-2-I3-P4–Subestructura óptima (1/1)

4) Se tiene que programar un conjunto de n actividades en una máquina. Cada actividad a_i tiene una hora de inicio s_i , una hora de término f_i , y un valor v_i . El objetivo es maximizar el valor total de las actividades programadas (y no necesariamente el número de actividades programadas).

Sea C_{ij} el conjunto de actividades que empiezan después que la actividad a_i termina y que terminan antes que empiece la actividad a_j . Sea A_{ij} una solución óptima a C_{ij} , es decir, A_{ij} es un subconjunto de actividades mutuamente compatibles de C_{ij} que tiene valor máximo.

a) Supongamos que A_{ij} incluye la actividad a_k . Demuestra que este problema tiene la propiedad de ***subestructura óptima***: una solución óptima al problema contiene soluciones óptimas a subproblemas.

b) Sea $val[i, j]$ el valor de una solución óptima para el conjunto C_{ij}. Demuestra que este problema tiene la propiedad de ***subproblemas traslapados***: si usamos un algoritmo recursivo para resolver el problema, este tiene que resolver los mismos subproblemas repetidamente.

Respuesta:

a) Podemos descomponer la solución óptima A_{ij} como $A_{ik} \cup \{a_k\} \cup A_{kj}$; es decir, la actividades que, estando en A_{ij} , terminan antes que empiece a_k , junto a las actividades que, estando en A_{ij} , empiezan después que termina a_k , junto a a_k . Así, el valor de la solución óptima A_{ij} es igual al valor de A_{ik} más el valor de A_{kj} más v_k . Siguiendo un razonamiento similar a los vistos en clase para otros problemas, concluimos que A_{ik} debe ser una solución óptima al problema definido por C_{ik} , y análogamente para A_{kj} y C_{kj} .

b) De acuerdo con a), $val[i, j] = val[i, k] + val[k, j] + v_k$. Pero en realidad no sabemos que la solución óptima al problema definido por C_{ij} incluye la actividad a_k , por lo que debemos mirar todas las actividades en C_{ij} para averiguar cuál elegir; es decir,

$$val[i, j] = 0, \qquad \text{si } C_{ij} = \varnothing$$
$$val[i, j] = \max_{a_k \in S_{ij}} \{ val[i, k] + val[k, j] + v_k \}, \qquad \text{si } C_{ij} \neq \varnothing$$

Aquí aparecen los subproblemas traslapados.

2010-2-Ex-P3–Subestructura óptima (misma pregunta que 2010-2-I3-P4) (1/1)

3) Se tiene que programar un conjunto de n actividades en una máquina. Cada actividad a_i tiene una hora de inicio s_i , una hora de término f_i , y un valor v_i . El objetivo es maximizar *el valor total de las actividades programadas* (y no necesariamente el número de actividades programadas).

Sea C_{ij} el conjunto de actividades que empiezan después que la actividad a_i termina y que terminan antes que empiece la actividad a_j . Sea A_{ij} una solución óptima a C_{ij} , es decir, A_{ij} es un subconjunto de actividades mutuamente compatibles de C_{ij} que tiene valor máximo.

a) Supongamos que A_{ij} incluye la actividad a_k . Demuestra que este problema tiene la propiedad de ***subestructura óptima***: una solución óptima al problema contiene soluciones óptimas a subproblemas.

b) Sea $val[i, j]$ el valor de una solución óptima para el conjunto C_{ij}. Demuestra que este problema tiene la propiedad de ***subproblemas traslapados***: si usamos un algoritmo recursivo para resolver el problema, este tiene que resolver los mismos subproblemas repetidamente.

7 Programación Dinámica

En esta sección se encuentran los siguientes contenidos:

- Problemas de Programación Dinámica
- Función de recurrencia
- Floyd-Warshall
- Bellman-Ford

2020-2-I3-P3–Modificaciones a Bellman-Ford (1/3)

Pregunta 3

Considere el siguiente algoritmo de Bellman-Ford para un grafo dirigido, con costos y sin ciclos de costo acumulado negativo:

```
bellman − ford(s):
    for each u ∈ V:
        d[u] ← ∞
        π[u] ← null
    d[s] ← 0
    for k = 1..|V| − 1:
        for each (u,v) ∈ E:
            if d[v] > d[u] + w(u,v):
                d[v] ← d[u] + w(u,v)
                π[v] ← u
```

Este algoritmo calcula las rutas de menor costo desde el vértice s a todos los demás vértices del grafo en tiempo $\Theta(V \cdot E)$. Queremos mejorar su rendimiento.

Sea $f(v)$ el número de aristas de la ruta de menor costo de s a v.

a) Definimos L como el máximo $f(v)$ entre todos los posibles v. Modifuica el algoritmo de Bellman-Ford para que su cimplejidad sea $\Theta(L \cdot E)$ para cualquier grafo.

b) Sea $g(v)$ la cantidad de aristas que llegan a v. Modifica el algoritmo de Bellman-Ford para que el tiempo que tome esté dado por la siguiente expresión:

$$T(V,E) = \sum_{v \in V} g(v) \cdot f(v)$$

Solución Pregunta 3a)

Recordemos que luego de la i-ésima iteración del algoritmo, todas las rutas tienen a lo más i aristas. Si $f(v) < i$, entonces las siguientes iteraciones no modificarán la ruta del vértice v.

En particular, la cantidad de iteraciones que será necesario hacer está dada por el mayor $f(v)$, L.

Podemos modificar el algoritmo para que detecte si hubo cambios en la ruta más corta de cualquier vértice. Si en una iteración no hubo cambios, significa que en todas las siguientes tampoco habrá cambios.

2020-2-I3-P3–Modificaciones a Bellman-Ford (2/3)

```
bellman − ford(s):
    for each u ∈ V:
        d[u] ← ∞
        π[u] ← null
    d[s] ← 0
    for k = 1..|V| − 1:
        changed ← false
        for each (u,v) ∈ E:
            if d[v] > d[u] + w(u,v):
                d[v] ← d[u] + w(u,v)
                π[v] ← u
                changed ← true
        if not changed, break
```

Y este punto se produce cuando hacemos la L-ésima iteración, ya que la ruta de ese vértice es la última en terminar de calcularse: de ahí en adelante las iteraciones no cambian las rutas calculadas.

Es decir, el for termina luego de la L-ésima iteración, y dentro del for se recorren todas las aristas cada vez, para una complejidad de $O(L \cdot E)$ en total (ignorando el $O(V)$ de la preparación del grafo)

Aclaración importante: Responder con SPFA es incorrecto porque, si bien este está acotado por arriba por $\mathcal{O}(L * E)$, no lo está por abajo (no se acota por $\Omega(L * E)$), y la notación pedida era en Big Theta, por lo que ambas condiciones debían cumplirse.

Solución Pregunta 3b)

Aclaración importante: A continuación se muestra una opción de respuesta. Esta, sin embargo, fue analizada y no sirve para todos los casos. Por lo tanto, se tomará como correcta cualquier propuesta de algoritmo que cumpla con los tiempos, aunque no sirva para todos los casos. También, se aceptará cualquier mejora sobre el algoritmo de la parte (a) de la pauta (que tenga mejor caso con complejidad menor a $\Omega(L * E)$ y sea correcto siempre), aunque no cumpla específicamente con el tiempo de ejecución pedido en todos los casos (como *Shortest Path Faster Algorithm* (SPFA)).

Como se dijo antes, luego de la i-ésima iteración del algoritmo, todas las rutas tienen a lo más i aristas. Si $f(v) < i$, entonces las siguientes iteraciones no modificarán la ruta del vértice v.

Para un vértice v, las únicas aristas que pueden modificar su ruta son las de la forma (u, v). Si en algún minuto del algoritmo la ruta de v ya es la más corta, no tiene sentido revisar ninguna de estas aristas.

Tal como $\alpha[v]$ son las aristas que salen de v, definimos $\beta[v]$ como las aristas que llegan a v.

En primer lugar, reformulamos el algoritmo para definirlo en función de β

2020-2-I3-P3–Modificaciones a Bellman-Ford (3/3)

```
bellman − ford(s):
    for each u ∈ V:
        d[u] ← ∞
        π[u] ← null
    d[s] ← 0
    for k = 1..|V| − 1:
        for each v ∈ V:
            for each u ∈ β[v]:
                if d[v] > d[u] + w(u, v):
                    d[v] ← d[u] + w(u, v)
                    π[v] ← u
```

Y ahora agregamos la parte de no revisar las aristas que llegan a vértices que ya están listos.

```
bellman − ford(s):
    for each u ∈ V:
        d[u] ← ∞
        π[u] ← null
    d[s] ← 0
    C ← V                                   // Vértices que no están listos
    while C ≠ ∅:
        C′ ← ∅                              // Aquí guardaremos los que no están listos
        for each v ∈ C:
            for each u ∈ β[v]:
                if d[v] > d[u] + w(u, v):
                    d[v] ← d[u] + w(u, v)
                    π[v] ← u
                    C′ ← C′ ∪ {v}           // Si no entra al if ni una vez es porque está listo
        C ← C′
```

Por lo tanto, para cada vértice se repite su $for\ u \in \beta[v]$ unas $f(v)$ veces. Como $g(v) = |\beta[v]|$, entonces cada vértice aporta $f(v) \cdot g(v)$ pasos al algoritmo.

2020-2-I3-P4–Problema de Progamación Dinámica (1/1)

Pregunta 4

Se acercan las fiestas, y el generoso Krampus te presenta n regalos, de los cuales debes escoger k. Cada regalo trae la etiqueta con su precio, y como eres una persona materialista, quieres maximizar la suma de los precios de los regalos que elijas. Los regalos están dispuestos en la forma de un árbol binario de navidad (no de búsqueda), donde cada nodo corresponde a un regalo. Hay una sola restricción para los regalos que puedes escoger: si escoges el regalo u, necesariamente debes escoger el padre de u en el árbol, y así sucesivamente.

Escribe un algoritmo de programación dinámica que indique los k regalos que debes escoger de manera de maximizar el precio total.

Solución Pregunta 4)

A continuación se propone un algoritmo que resuelve el problema. Este algoritmo no es el único que lo resuelve. Se evalúa que el algoritmo que propongan cumpla los criterios establecidos en el desglose de puntaje.

```
function ProgramacionDinamica(u, k):
    Aux = diccionario inicialmente vacío
    maximum, set = C(u, k)
    return set

function C(u, k):
    if Aux[u, k] not null:
        return Aux[u, k]
    if k == 0:
        Aux[u, k] = (0, {})
    else if k == 1:
        Aux[u, k] = (u.precio, {u})
    else if left(u) is null and right(u) is null and k > 1:
        Aux[u, k] = (-Inf, {})
    else:
        max = 0, set = {u}
        for i in 0..k-1:
            price_left, set_left = C(left(u), i)
            price_right, set_right = C(right(u), k - 1 - i)
            if price_left + price_right > max:
                set = {u} union set_left union set_right
                max = price_left + price_right
        Aux[u, k] = (max + u.price, set)
    return Aux[u, k]
```

- 2pt Por definir la función recursiva que resuelva el problema en base a la resolución de subproblemas del mismo índole (Tipo **DFS**) y que recorra solo hasta k. 1pt si no recorre hasta k.
- 1pt Por utilizar programación dinámica, implementando la estructura de datos auxiliar (o bien creándola en el mismo nodo) que almacene resultados anteriores (puede ser tabla de hash, diccionario, arreglo, etc).
- 1pt Por utilizar la estructura auxiliar adecuadamente.

2020-1-Ex-P4–Función de recurrencia (1/5)

4. Se tiene una colección de $\boldsymbol{k}$ jarros de volúmenes $\{\boldsymbol{v_1}, \cdots, \boldsymbol{v_k}\}$ todos **distintos** entre sí, donde el volumen de cada jarro es un número primo de litros.

   Se quiere usar estos jarros para sumar un volumen arbitrario $\boldsymbol{x}$ mediante las acciones de costo 1:

   - $\infty \rightarrow \boldsymbol{v_i}$: llenar el jarro $\boldsymbol{i}$ con $\boldsymbol{v_i}$ litros.
   - $\boldsymbol{v_i} \rightarrow \boldsymbol{v_j}$: llenar el jarro $\boldsymbol{j}$ con el contenido $\boldsymbol{v} \leq \boldsymbol{v_i}$ del jarro $\boldsymbol{i}$**.** Así, el jarro $\boldsymbol{i}$ queda con $\boldsymbol{v} - \boldsymbol{v_j}$ litros.

   Y las acciones de costo 0:

   - $\boldsymbol{v_i} \rightarrow \emptyset$ **:** desechar el contenido del jarro $\boldsymbol{i}$
   - $\boldsymbol{v_i} \rightarrow \boldsymbol{F}$ **:** traspasar el contenido del jarro $\boldsymbol{i}$ a un recipiente final $\boldsymbol{F}$

   Considera el siguiente ejemplo de costo 4 con los jarros $\{\mathbf{5}, \mathbf{7}\}$ y $\boldsymbol{x} = \mathbf{4}$:

   - $\infty \rightarrow \mathbf{7}$
   - $\mathbf{7} \rightarrow \mathbf{5}$
   - $\mathbf{7} \rightarrow \boldsymbol{F}$
   - $\mathbf{5} \rightarrow \emptyset$
   - $\infty \rightarrow \mathbf{7}$
   - $\mathbf{7} \rightarrow \mathbf{5}$
   - $\mathbf{7} \rightarrow \boldsymbol{F}$

   a) Escribe la ecuación de recurrencia que calcule el costo **mínimo** de resolver este problema.
   b) Mediante diagramas, justifica la utilidad de aplicar programación dinámica a este problema.

2020-1-Ex-P4–Función de recurrencia (2/5)

Problema 4

Primero, llamaremos J al conjunto de jarros. Definimos la funcion $V(J, x)$ que es el costo mínimo de sumar x con los jarros de J.

Parte A

En primer lugar, por definición, si queremos sumar un número x y hay un jarro de ese volumen, entonces el costo es 1 ya que es llegar y hacer $\infty \to x$. Es decir:

$$V(J,x) = \begin{cases} 1, & x \in J \\ \dots \end{cases}$$

[1pt] por el caso base.

Luego, los números de la forma $v - v_j$ se pueden construir haciendo $v_i \to v_j$, donde v_i es el jarro que contiene el volumen v. El costo de esto será (el costo de construir v) + 1. Llamaremos a este costo $T(J, x)$

$$T(J,x) = \begin{cases} V(J,v) + 1, & \exists\ v_j \in J \mid x = v - v_j \\ \dots \end{cases}$$

Ya que no sabemos cual es el más barato, debemos revisar cada uno. Considerando que $v = x + v_j$:

$$T(J,x) = \begin{cases} \min\limits_{v_j\ \in\ J} [V(J, x + v_j) + 1] \\ \dots \end{cases}$$

Cambiando v_j de nombre a j para que sea más legible:

$$T(J,x) = \begin{cases} \min\limits_{j\ \in\ J} [V(J, x + j) + 1] \\ \dots \end{cases}$$

[1pt] Por establecer el caso del traspaso de un jarro a otro.

Ojo que así como está, nada impide que se llame a $V(J, x + j)$ con números cada vez más grandes y que la recursión jamás termine. Y lo cierto es que para poder hacer el traspaso de un jarro a otro, este volumen tiene estar contenido en algun jarro. Eso significa que no sirve sumar volumenes más grandes que el jarro más grande (llamémoslo $J_{\max}$) para luego traspasarlo a otro jarro, así que lo dejamos fuera del MIN. Si el MIN queda vacío, es decir, x más el mínimo de J ya supera al máximo, entonces retornamos infinito.

$$T(J,x) = \begin{cases} \infty, & x + J_{\min} > J_{\max} \\ \min\limits_{\substack{j\ \in\ J \\ x+j\ \leq\ J_{\max}}} [V(J, x + j) + 1], & else \end{cases}$$

2020-1-Ex-P4–Función de recurrencia (3/5)

Además, todos los números mayores a 1 pueden ser expresados como la suma de otros dos números, por lo que para los números que no cumplan con ninguna de las dos anteriores, debemos descomponerlos y buscar la forma más barata de sumarlos. Llamaremos a esto $S(J, x)$

$$S(J,x) = \begin{cases} \infty, & x = 1 \\ \min\limits_{y \in [1,\cdots,\lfloor \frac{x}{2} \rfloor]} [V(J,y) + V(J, x-y)], & else \end{cases}$$

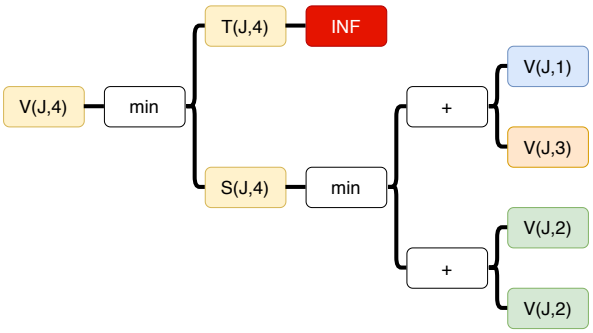
[1pt] Por establecer el caso de la descomposición

Uniendo todo, la manera más barata de crear un volumen va a ser o la más barata traspasando, o la más barata sumando.

$$V(J,x) = \begin{cases} 1, & x \in J \\ \min\left[T(J,x), S(J,x)\right], & else \end{cases}$$

Parte B

Tomemos el ejemplo del enunciado: $J = \{5, 7\}$, $x = 4$. Haremos un diagrama para mostrar la recursión.



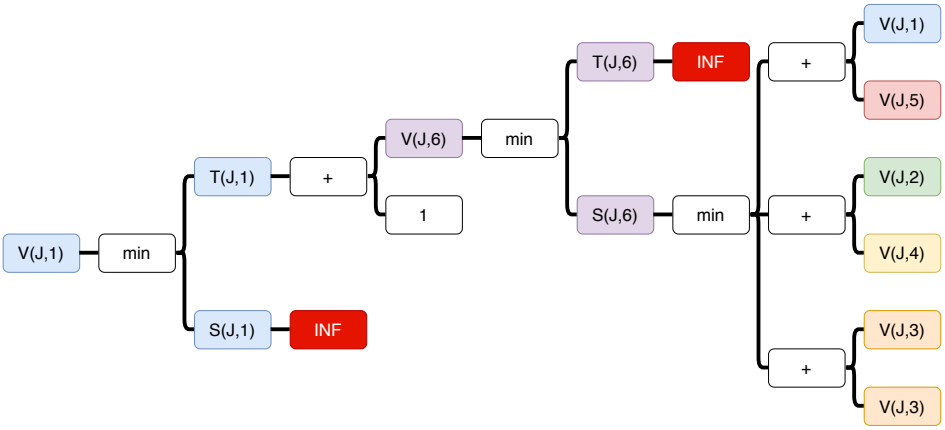
Podemos ver que ya en la primera llamada se nos repite el $V(J, 2)$: para que calcularlo dos veces si podemos calcularlo una vez y guardar su resultado? Para esto sirve programación dinámica: para no tener que calcular dos veces un resultado que ya conocemos.

[1pt] Por indicar casos en que se repiten las llamadas a la funcion con los mismos parámetros.
[1pt] Por explicarlo mostrando un diagrama (se pedía explicítamente en el enunciado)
[1pt] Por explicar por qué es util usar programación dinámica en este caso.

2020-1-Ex-P4–Función de recurrencia (4/5)

Lo siguiente se incluye solo para que la respuesta esté completa.

Veamos qué pasa en $V(J,1)$:



Podemos ver que DENTRO de la llamada a $V(J,1)$ hay nuevamente una llamada a $V(J,1)$. Esto significa que el algoritmo jamás va a terminar. Considerando que no tiene sentido argumentar circularmente el mínimo costo de de sumar un número, entonces deberíamos dejar fuera esos casos, al igual que el $V(J,4)$ (recordemos que seguimos dentro de la llamada original a $V(J,4)$).

Para solucionar esto podemos agregar un "contexto" a la ecuación de recurrencia, para ignorar llamadas a $V(J,x)$ si ya estamos dentro de una llamada a $V(J,x)$.

Esto lo podemos hacer con un conjunto C que indica todos los x dentro de los cuales estamos, partiendo por el x original, en nuestro caso $= 4$.

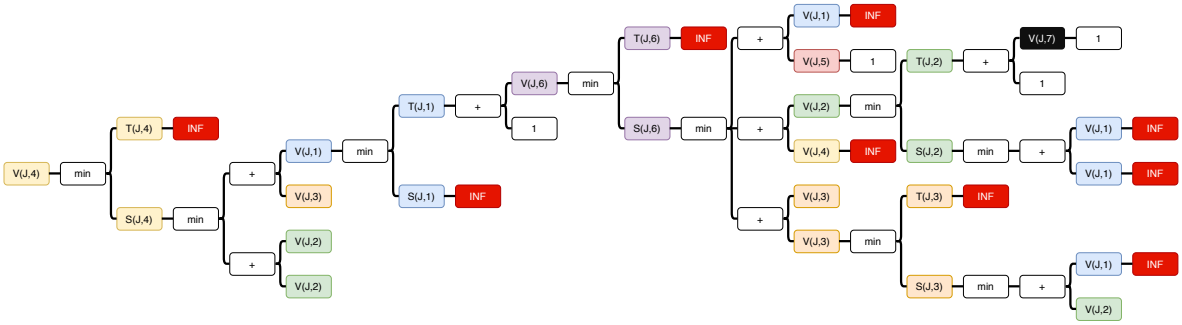
$$V(J,C,x) = \begin{cases} \infty, & x \in C \\ 1, & x \in J \\ \min\left[T(J,C,x), S(J,C,x)\right], & else \end{cases}$$

$$T(J,C,x) = \begin{cases} \infty, & x + J_{\min} > J_{\max} \\ \min\limits_{\substack{j \in J \\ x+j \leq J_{\max}}} [V(J, C \cup \{x+j\}, x+j) + 1], & else \end{cases}$$

$$S(J,C,x) = \begin{cases} \infty, & x = 1 \\ \min\limits_{y \in [1,\cdots,\lfloor \frac{x}{2} \rfloor]} [V(J, C \cup \{y\}, y) + V(J, C \cup \{x-y\}, x-y)], & else \end{cases}$$

2020-1-Ex-P4–Función de recurrencia (5/5)

Con contexto y programación dinámica el árbol de llamadas queda como sigue (se obvia el contexto en los parámetros).



Así, $V(J,4) = 4$

Nota del ayudante: elegí el ejemplo del enunciado por simplicidad, no se me ocurrió que fuera el caso más complejo para ese J :c

2019-1-C6–Función de recurrencia, diagrama de recursión (1/5)

Estructuras de Datos y Algoritmos - IIC2133
Pauta Control 6

1. Dado un string $S = a_1 \cdots a_n$ se define k como el mínimo valor tal que $S = P_1 \cdots P_k$, donde cada P_i es un palíndromo.

a) Escribe la función de recurrencia $K(s, i, j)$ que dado un subtring $s[i{:}j]$ determine el valor del k antes descrito, justificando su correctitud.

Solución:

$$K(s,i,j) = \begin{cases} 1 & \text{si } s[i:j] \text{ es un palíndromo} \\ \min\limits_{x=i}^{j-1}(K(s,i,x) + K(s,x+1,j)) & \text{en otro caso} \end{cases}$$

[0.5pt] por el caso base de la recurrencia

[1pt] por el caso recursivo

[0.5pts] por que los límites del caso recursivo sean correctos. Se perdona hasta 1 error en los límites.

Por ejemplo

Min de x=i … j, o K(s,i,x) + K(s,x,j) son errores en los límites del caso recursivo.

[1pt] Por la justificación de correctitud.

Justificación:

La correctitud del caso base es trivial. **[0pts]**

Lema: existe **alguna** manera de cortar S en dos strings S_1 y S_2 tal que $S = S_1 S_2$ y $k(S) = k(S_1) + k(S_2)$. **[0.25pts]** (Esta propiedad es la que hace que la recurrencia funcione)

2019-1-C6–Función de recurrencia, diagrama de recursión (2/5)

No tenemos cómo saber a priori cual es la manera óptima de cortar S, pero por el lema, **alguno** de los cortes entregará el k mínimo. Así que para encontrarlo, probamos **todos** los posibles cortes de S y elegimos el que nos entregue el menor k **[0.25pts]**

[Demostración del lema] **[0.5pts]**

(Para demostrar esto basta con probar que existe al menos un corte que cumple la propiedad del lema)

Posible demostración:

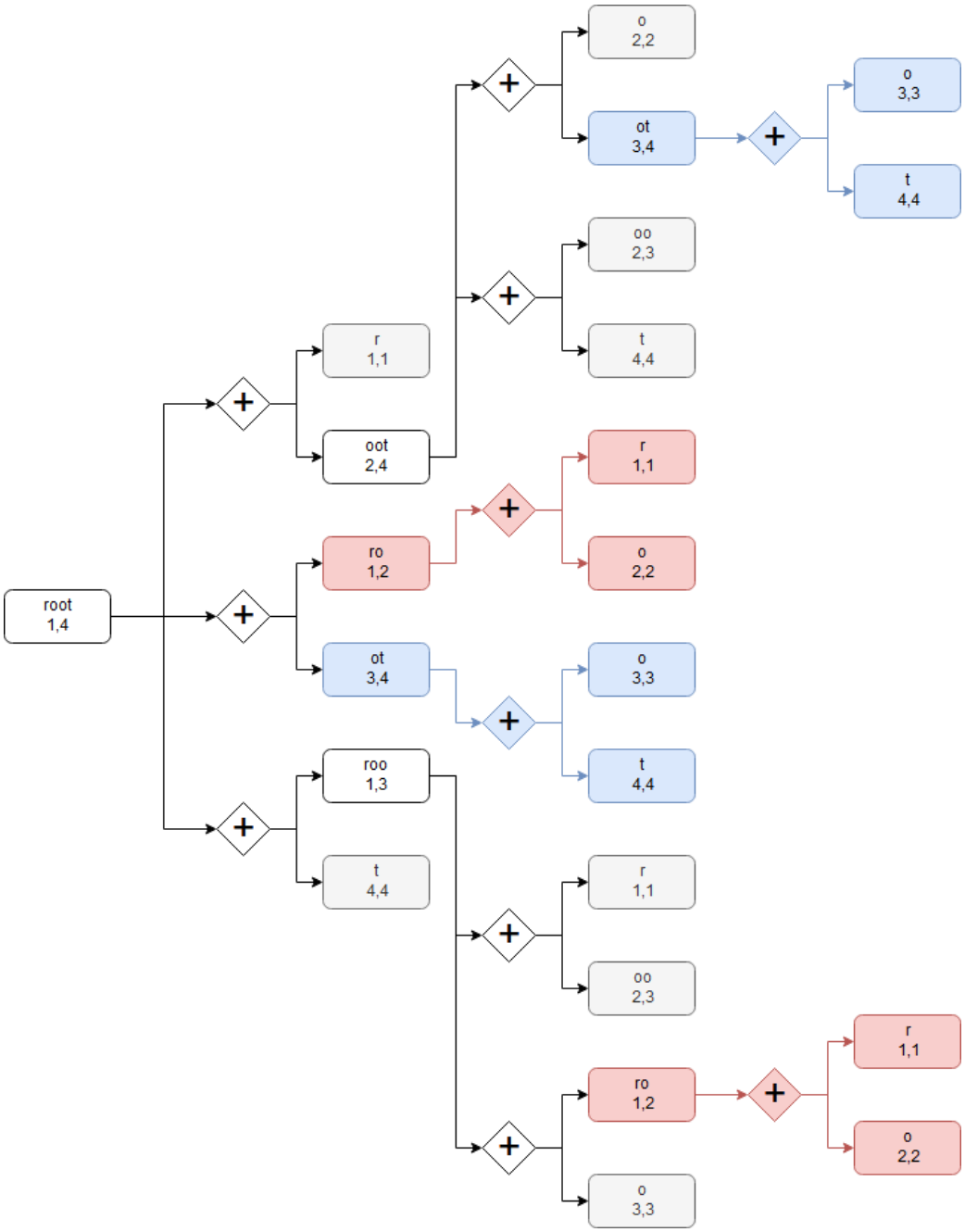
Para un string S dado siempre existe una forma de expresarlo como $S = P_1 \cdots P_k$, aunque sea con $k = n$. Si elegimos un y cualquiera entre 1 y $k - 1$, y lo usamos para cortar S en 2 de la siguiente manera: $S_1 = P_1 \cdots P_y$, $S_2 = P_{y+1} \cdots P_k$ tenemos que el k de S_1 es y, y el k de S_2 es $k - y$. Con esto tenemos al menos una forma de cortarlo.

[No es necesario que la justificación sea tan formal, mientras mencionen y justifiquen correctamente los puntos descritos.]

b) Explica y justifica el uso de programación dinámica para este problema. Haz un diagrama de la recursión para respaldar tus argumentos.

Usemos como ejemplo la palabra *root*. (pueden usar la palabra que quieran)

2019-1-C6–Función de recurrencia, diagrama de recursión (3/5)



2019-1-C6–Función de recurrencia, diagrama de recursión (4/5)

[1pt] por mostrar que la recurrencia puede generar un mismo subproblema más de una vez:

Ejemplo: como se ve en el diagrama, las llamadas K(s,1,2) y K(s,3,4) se repiten, y generan un árbol entero repetido. [No es necesario dibujar el diagrama **completo** de la recursión mientras quede claro que se generan subárboles repetidos y los indiquen correctamente]

Sólo si está justificado correctamente, [1pt] por explicar cómo aplicar programación dinámica:

Ejemplo: Podemos calcular el valor de cada llamada una sola vez, guardar su resultado, y las siguientes veces que se haga esa llamada simplemente retornar el valor previamente calculado, así ahorramos generar árboles de llamadas repetidos.

c) **[1pt]** por complejidad con PD:

Con programación dinámica se calculará a lo más una vez cada llamada distinta por lo que la complejidad es el costo de cada llamada una sola vez.

Sabemos que se tiene que verificar si la palabra es un palíndromo por lo que cada llamada toma O(largo sub palabra).

Por lo tanto el tiempo total es la suma de los largos de todas las sub palabras distintas:

$$T(n) = 1 \cdot n + 2 \cdot (n-1) + 3 \cdot (n-2) + \ldots + n \cdot 1$$
$$T(n) \in O(n^3)$$

[0.5pts de bonus] por la complejidad sin programación dinámica:

El peor caso es cuando $k = n$.

Planteamos T(n) como el tiempo de procesar una palabra de largo n

$$T(n) = n + \sum_{x=1}^{n-1} T(x) + T(n-x)$$

Por simetría, eso es equivalente a

$$T(n) = n + 2\sum_{x=1}^{n-1} T(x)$$

2019-1-C6–Función de recurrencia, diagrama de recursión (5/5)

Esta función es estrictamente mayor a

$$f(n) = \sum_{x=1}^{n-1} f(x)$$

Desarrollamos:

$$f(n) = f(n-1) + f(n-2) + f(n-3) + \ldots + f(1)$$
$$f(n) = f(n-1) + f(n-1)$$
$$f(n) = 2 \cdot f(n-1)$$

Y esto es $f(n) = 2^{n-1}$

Por lo tanto $T(n) \in \Omega(2^n)$. En otras palabras es al menos exponencial.

2018-2-I3-P4-I–Función de recurrencia (1/2)

4. Rutas más cortas / programación dinámica

I. **(3pts)** El canal DCC TV tiene un nuevo programa llamado EDD donde los participantes tienen la chance de concursar y ganar dinero en premios. El juego consiste en un triángulo de pelotas, donde cada pelota tiene impreso un número íntegro, tal como se muestra en la figura.

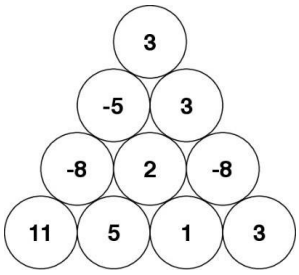


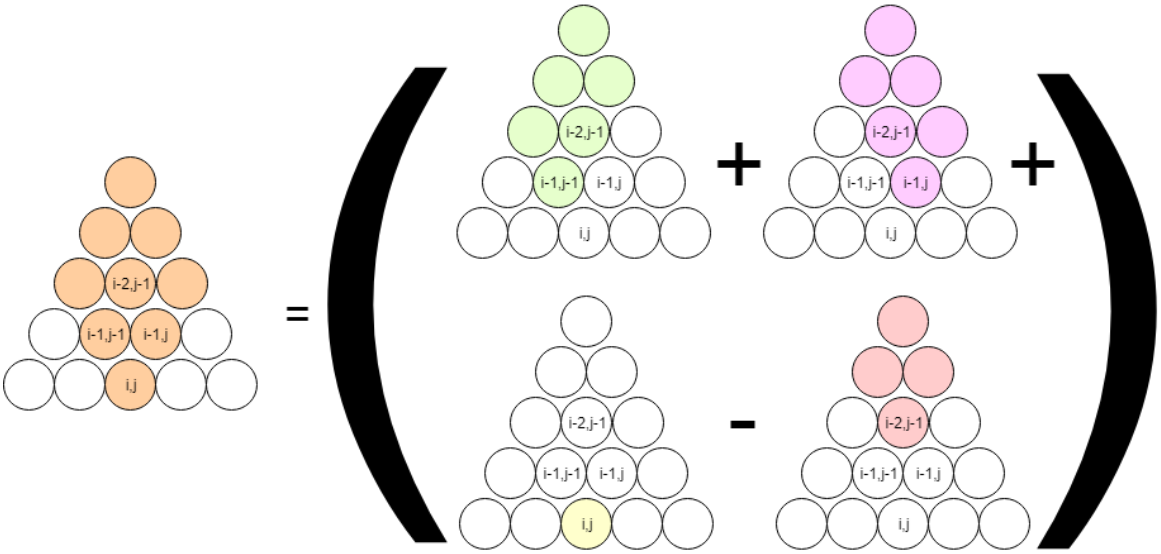
Figura 1

El jugador puede elegir una pelota de la pirámide y sacarla o no sacar ninguna. En caso de que saque una pelota, se quedará con los puntos de esa pelota y **todas las que estén arriba de ella**. En caso de que saque ninguna, se quedará con 0 puntos. En la figura 1, para sacar la pelota 1, se necesitan sacar también las pelotas 2, -8, -5, 3 y 3 por lo que el valor de sacar la pelota 1 sería -4. El presentador está preocupado por el máximo premio que se pueda llevar un concursante, por lo que se te pide ayuda a ti para resolver este problema.

Esta pirámide se representa como arreglo de arreglos M donde M[0] es el nivel superior y M[n] es el nivel de más abajo. Cada nivel M[i] tiene i+1 elementos.

a) Haz la ecuación de recurrencia que permite calcular el premio obtenido de sacar una pelota específica: *P(i, j, M)* donde la pelota elegida es M[i][j].

Respuesta: [1.5 pts] Se acepta como respuesta correcta una función de recurrencia de forma matemática o un pseudocódigo que exprese la ecuación siguiente:



2018-2-I3-P4-I–Función de recurrencia (2/2)

Esto es:

$$p(i,j,M) = \begin{cases} 0 & , \quad i,j < 0 \; o \; j > i \\ M[i][j] + p(i-1,j-1,M) + p(i-1,j,M) - p(i-2,j-1,M), & else \end{cases}$$

Si falta 1 término de la ecuación (normalmente falta el *-p(i-2, j-1, M)*), se da 1 punto de 1.5

b) Crea una solución iterativa de este problema en la cual se calcula el premio para toda pelota de la pirámide (Versión bottom-up).

Respuesta: [1.5 pts] Para hacer una respuesta iterativa para toda la pirámide debo crear una tabla R que contenga las respuestas e implementarlo de la siguiente forma:

```
premios(M):
        R = Tabla de las mismas dimensiones de M
        for i = 0..n:
                for j = 0..i:
                        R[i][j] = M[i][j]
                        if i > 0:
                                if j > 0:
                                        R[i][j] += R[i-1][j-1]
                                if j < i:
                                        R[i][j] +=R[i-1][j]
                                if i - 2 >= 0 and j > 0:
                                        R[i][j] -=R[i-2][j-1]
        return R
```

Se acepta como respuesta correcta la implementación iterativa del problema modelado en 4a) independiente de si es correcto para el problema.

2018-2-I3-P4-II–Floyd-Warshall, Dijkstra (1/1)

II. **(3pts)** La gran ventaja del algoritmo de Floyd-Warshall frente a Dijkstra es que permite tener las rutas precalculadas y almacenadas. El tiempo que toma en construir la matriz de costos mínimos entre todos los pares de nodos es O(|$\mathbb{V}$|^3). Digamos el algoritmo de Floyd-Warshall es muy complicado de entender por lo que queremos utilizar el algoritmo de Dijkstra para calcular esta matriz de costos mínimos. Calcule la complejidad de este método.
Considere que el algoritmo de Dijkstra toma tiempo O(|E| log(|V|)) en su implementación con un min heap como cola de prioridad.

Respuesta: [3 pts] Para rellenar la matriz de distancias mínimas hace falta obtener los valores de min_dist(i, j) para todos los pares de nodos i, j. Dijkstra permite obtener el valor de min_dist desde un nodo i hasta todos los nodos del grafo, por lo que con una llamada a Dijkstra se rellena una fila de la matriz. Por lo tanto solo es necesario usar Dijkstra |V| veces. Esto tiene un costo de O(|V||E|log(|V|)).

Se dio 0 puntos por usar Dijkstra por cada par de nodos.

Se dio 2 puntos por dar mal la complejidad a pesar de haber explicado bien el procedimiento.

2018-2-Ex-P4–Análisis de implementación de PD en un problema (1/2)

4. Programación dinámica

En clases estudiamos el problema de programar el mayor número de tareas en una misma máquina, en que cada tarea k tiene una hora de inicio s_k y una hora de finalización f_k, y la máquina solo puede ejecutar una tarea a la vez. Vimos que el problema se puede resolver usando un algoritmo codicioso en tiempo O(n), si hay n tareas en total y vienen ordenadas por sus horas de finalización.

Considera ahora que cada tarea tiene, además, un valor v_k, y queremos maximizar la suma de los valores de las tareas realizadas (y no el número de tareas realizadas).

a) Muestra que el algoritmo codicioso anterior no resuelve esta versión más general del problema.

Bssta un ejemplo:

k: s_k–f_k, v_k

1: 3–10, 1
2: 5–15, 3
3: 12–20, 1

En este caso, el algoritmo codicioso elige las tareas 1 y 3, con un valor total de 1+1 = 2; pero la elección de la tarea 2 habría producido un valor de 3.

Muestra que el problema se puede resolver mediante programación dinámica. En particular, suponemos nuevamente que las tareas vienen ordenadas por sus horas de finalización: $f_1 \leq f_2 \leq \ldots \leq f_n$. Entonces, para cada tarea j, definimos b[j] como la tarea g que termina más tarde antes del inicio de j; b[j] = 0 si ninguna tarea satisface esta condición.

Sea T es una solución óptima al problema. Revisamos las tareas "de atrás hacia adelante": obviamente, la tarea n pertenece a T, o bien la tarea n no pertenece a T.

b) Argumenta clara y precisamente que en cada una de estas dos situaciones el problema puede resolverse a partir de encontrar la solución óptima a un problema del mismo tipo pero más pequeño.

Si la tarea n no pertenece a T, entonces T es igual a la solución óptima para las tareas 1, …, n–1; es decir, un problema del mismo tipo pero más pequeño.

En cambio, si la tarea n pertenece a T, entonces ninguna tarea q, b[n] < q < n, puede pertenecer a T, por lo que T debe incluir, además de la tarea n, una solución óptima para las tareas 1, …, b[n]; es decir, nuevamente, un problema del mismo tipo pero más pequeño.

c) Generalizando, podemos decir que si T_j es la solución óptima al problema de las tareas 1, …, j, y su valor es OPT(j), entonces el problema original consiste en buscar T_n y encontrar su valor OPT(n). Escribe una ecuación recursiva para encontrar OPT(j), en función de v_j, b[j] y j–1.

La generalización se traduce a

si j pertenece a T_j , entonces OPT(j) = v_j + OPT(b[j])

si j no pertenece a T_j , entonces OPT(j) = OPT(j–1)

Por lo tanto, **OPT(j) = max{ v_j + OPT(b[j]) , OPT(j–1) }** … (falta agregar condición de borde)

2018-2-Ex-P4–Análisis de implementación de PD en un problema (2/2)

d) Explica claramente cuál sería la dificultad práctica de tratar de resolver el problema aplicando directamente la ecuación recursiva anterior.

La ecuación recursiva hace dos llamadas recursivas, pero (a diferencia de *quicksort* o *mergesort*, en que las llamadas recursivas son sobre conjuntos disjuntos de datos) estas pueden ser llamadas para resolver el mismo problema, o para resolver problemas que ya han sido resueltos; es decir, las llamadas son sobre conjuntos de datos no necesariamente disjuntos y, por lo tanto, pueden terminar resolviendo un mismo problema muchas veces. Así, **el número total de problemas efectivamente resueltos puede ser mucho mayor que el número total de problemas diferentes**.

e) Explica precisamente cómo la programación dinámica ayuda a resolver la dificultad de d).

La P.D. parte resolviendo los problemas más pequeños (problemas que no dan origen a llamadas recursivas) primero y va almacenando estos problemas (convenientemente codificados) y sus resultados en una tabla. Así, cuando aparece un problema para resolver, primero se busca en la tabla a ver si ese problema ya está resuelto, en cuyo caso se usa el resultado que está en la tabla y no se vuelve a resolver el problema.

2018-1-Ex-P2–Análisis de implementación de PD en un problema (1/2)

2. Programación dinámica

Queremos dar vuelto de S pesos usando el menor número posible de monedas. Si los valores de las monedas, en pesos, ordenados de mayor a menor son $\{v_1, v_2, \ldots, v_n\}$ (es decir, $v_1 > v_2 > \ldots > v_n = 1$), y tenemos una cantidad suficientemente grande de monedas de cada valor, entonces:

a) Muestra que la estrategia codiciosa de dar tantas monedas como sea posible de valor v_1, seguido de dar tantas monedas como sea posible de valor v_2, y así sucesivamente con v_3, etc., no siempre usa el menor número posible de monedas para totalizar S pesos. [**1 pto**]

b) Demuestra, en cambio, que el problema siempre puede resolverse usando programación dinámica. En particular, sea $z(S,n)$ el problema de encontrar el menor número de monedas necesarias para totalizar la cantidad S, con monedas de valor $\{v_1, v_2, \ldots, v_n\}$; entonces:

b1) Dada una solución a $z(S,n)$, identifica subpartes de la solución que sean soluciones óptimas para algunos subproblemas (del mismo tipo, pero más pequeños); o bien, identifica subproblemas cuyas soluciones óptimas puedan ser usadas para construir una solución a $z(S,n)$. [**1.5 pts.**]

b2) Escribe la recurrencia que relaciona la solución a $z(S,n)$ con las soluciones óptimas a los subproblemas; luego, generaliza esta recurrencia de modo que sea aplicable para resolver los subproblemas; y, finalmente, escribe los casos iniciales (es decir, los casos cuyos valores se determinan sin aplicar la recurrencia). [**2 pts.**]

b3) Escribe el algoritmo de programación dinámica, típicamente un algoritmo iterativo. (Si, en cambio, escribes un algoritmo recursivo, asegúrate de incluir los tests necesarios para evitar hacer cálculos redundantes). [**1.5 pts.**]

Respuesta:

a) Basta con dar un contraejemplo donde usar la estrategia greedy no llega al óptimo. Esto se puede hacer cuando los valores de las monedas no son divisores de las monedas más grandes **[1pto]**.

b1) Digamos que la moneda más grande usada en nuestra solución a *z(S,n)* es v_k. Entonces el problema *z(S-*v_k*,n)* fue resuelto de manera óptima para que *z(S,n)* sea óptimo **[1.5pts]**.

b2) Recurrencia **[2pts]**:

$$z(S,n) = \begin{cases} \infty \; si \; S < 0 \\ 0 \; si \; S = 0 \\ \min\limits_{i=1..n} (z(S - v_i, n) + 1) \end{cases}$$

OJO: Hay muchas maneras de escribir esta recurrencia. Esta es solo una manera de hacerla.

2018-1-Ex-P2–Análisis de implementación de PD en un problema (2/2)

b3)

Modo recursivo:

```
int z(S, n, T):
        si T[S] está en la tabla:
                return T[S]
        si S < 0:
                return infinity
        si S = 0:
                return 0

        minimo = infinity

        for i = 1…n:
                result = z(S-v_i, n, T) +1
                si result < minimo:
                        minimo = result
        T[S] = minimo
        return minimo
```

Modo iterativo:

```
int z(S, n):
        T = [-1 for i = 0…S]
        T[0] = 0
        for i = 1...S:
                si tengo una moneda de costo S:
                        T[i] = 1
                        Break
                minimo = infinity
                for j = 1…n:
                        si v_j ≤ S:
                                result = T[S-v_j] +1
                                si result < minimo:
                                        minimo = result
                T[S] = minimo
```

OJO: Hay muchas maneras de escribir estos algoritmos.

2018-1-Ex-P4–Bellman-Ford (1/1)

4. Algoritmo de Bellman-Ford

a) El algoritmo revisa todas las aristas en cada iteración para ver si es necesario actualizar el costo de llegar a un nodo. Sin embargo, es posible saber cuáles son las aristas que realmente vale la pena revisar en cada iteración (en vez de revisarlas siempre todas). Describe una versión del algoritmo que incorpore este cambio y justifica por qué mejoraría la eficiencia del algoritmo.

b) ¿Cómo se puede hacer para detectar si el grafo contiene un ciclo cuyo costo total es negativo? Juatifica.

Respuesta:

a) Al hacer una iteración actualizando los pesos de algunos nodos solo es posible actualizar los pesos de los nodos vecinos a los recién actualizados **[1pto]**, por lo que se pueden guardar los vecinos de los recién actualizados en una cola y solo se actualizan estos en la iteración siguiente **[2ptos]**.

b) Luego de hacer $|V|$ iteraciones ya no debería haber más actualizaciones de los pesos de los nodos si se sigue iterando a no ser que exista un ciclo negativo. Por lo que si se hace una iteración número $|V|+1$ y se actualiza algún peso entonces existe un ciclo de costo negativo **[3ptos]**.

2017-1-Ex-P4–Bellman-Ford (1/2)

4. Dado un grafo $G = (V, E)$, una función de pesos $w : E \to \mathbb{R}^+$, y un nodo $s \in V$, nos interesa el problema de encontrar el costo del camino simple (sin ciclos) de *mayor* costo entre s y cada nodo del grafo.

   Un ex-alumno de IIC2133 argumenta que una modificación del algoritmo de Bellman-Ford (BF) puede resolver este problema. Su razonamiento es el siguiente:

   > *"Como el camino que buscamos no tiene ciclos, el camino más largo entre dos nodos tiene a lo más* $|V| - 1$ *aristas. De esta forma podemos primero buscar los caminos más largos de 1 arista, luego los de dos aristas, y así sucesivamente, tal como lo hace BF."*

   El algoritmo que propone es este:

```
1  procedure Init()
2      for each u ∈ V[G] do
3          d[u] ← −∞
4          π[u] ← nil
5  procedure CaminosMasCaros(G,s)
6      Init()
7      d[s] ← 0
8      for i ← 1 to |V| − 1 do
9          for each (u, v) ∈ E do
10             costo ← d[u] + w(u, v)
11             if costo > d[v] then
12                 d[v] ← costo
13                 π[v] ← u
```

2017-1-Ex-P4–Bellman-Ford (2/2)

Note que el único cambio respecto de la rutina de *CaminosMasCortos*, discutida en clases, son la línea 3 (cambio de ∞ por $-\infty$) y la línea 11 (cambio de $<$ por $>$).

En el resto de esta pregunta usted debe dar un argumento magistral que muestre por qué este algoritmo es **incorrecto**. Específicamente:

a) Entregue un contraejemplo que muestre que el algoritmo es incorrecto. **Respuesta:** El contrajemplo se construye con un grafo con un ciclo en el camino hacia un nodo.

b) Identifique una condición que debe cumplir el grafo G para que el algoritmo del alumno sea correcto. Demuéstrelo. (Ayuda: reduzca el problema de encontrar caminos más caros al de encontrar caminos más cortos.) **Respuesta:** Observamos primero que el algoritmo del alumno está basado en una subrutina de BF. Segundo, observamos que encontrar el camino más caro sin ciclos en un grafo $G = (V, E)$ con w es equivalente a encontrar el camino más corto sin ciclos en un grafo $G = (V, E)$ con $-w$. BF, ejecutado con $(G, -w)$, calculará el camino más corto sin ciclos exactamente cuando retorna "true". Y BF retorna true ssi no hay un ciclo negativo. $(G, -w)$ tiene un ciclo negativo ssi G tiene un ciclo, y por lo tanto esta es la propiedad que debe cumplir G para que el algoritmo del alumno sea correcta.

c) Hay parte del argumento del alumno que efectivamente está correcto: "los caminos más largos no pueden tener más de $|V| - 1$ aristas". Diga cómo, entonces, es posible modificar la idea de BF para encontrar caminos más largos. Analice el algoritmo resultante. **Respuesta:** La modificación no es sencilla ni eficiente. Esto es porque el camino más largo hasta U podría pasar por V, mientras que el camino más largo hasta V podría pasar por U.

La modificación de BF, cada vez que encuentra un nuevo camino hasta un nodo, lo guarda explícitamente, junto a su costo.

En la práctica esto significa que tenemos una matriz $d[v][i]$ que almacena el costo del i-ésimo camino sin ciclos hasta v que se ha encontrado. El camino, as su vez, se almacena en $\pi[v][i]$, por ejemplo, como una lista ligada.

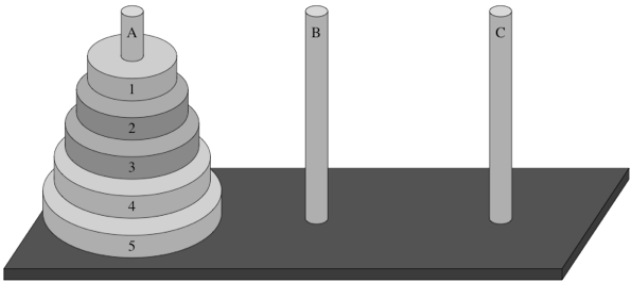
Cuando en la línea 9 miramos (u, v) debemos usar esta arista completando cada camino hasta u para generar un nuevo camino hasta v. El costo de computar esto depende del tamaño de $d[v][]$, que, lamentablemente, en el peor caso debe llevar la cuenta de todos los caminos posibles desde la fuente hasta v, que es exponencial en $|V|$ ($O(|V|!)$, de hecho)

El algoritmo principal, entonces, realiza $|V| - 1$ iteraciones, pero cada iteración es $O(V!)$. El algoritmo es $O((|V| + 1)!)$.

2016-1-I2-P4–Función recursiva, complejidad (1/1)

4. En el problema conocido las *torres de Hanōi*, hay 3 agujas verticales, llamadas A, B y C (ver figura). En la aguja A hay una torre de n discos, todos de distintos tamaños, ordenados hacia arriba de más grande a más pequeño (en la figura, $n = 5$). El problema consiste en mover la torre entera de discos de la aguja A a la aguja B, usando la aguja C como auxiliar, siguiendo las siguientes reglas:

- Sólo se puede mover un disco a la vez.
- Sólo se puede mover el disco de más arriba de una de las torres y depositarlo encima de otra torre.
- Ningún disco, en ningún movimiento, puede ser depositado sobre otro más pequeño.



a) Escribe la ecuación de recurrencia que resuelve el problema de mover la torre de n discos de la aguja A a la aguja B [0.5]; calcula la complejidad de resolver esta ecuación recursivamente [0.5].

b) Explica cómo usar programación dinámica para mejorar la solución en **a**) [2.5]; calcula la complejidad usando este método [0.5].

c) Si pudiéramos llevar a cabo una secuencia arbitraria de movimientos de manera instantánea, ¿cómo se vería afectada la complejidad de la solución en **a**)? [0.5] ¿Y la de la solución en **b**)? [0.5].

En tus respuestas, considera tanto las complejidades de M(n), el costo de mover los discos, y H(n), el costo de calcular la solución. La complejidad del problema es la de H + M.

2015-2-Ex-P2–Bellman-Ford (1/1)

2. En clase estudiamos la siguiente versión del algoritmo de Bellman–Ford para encontrar las rutas más cortas desde un vértice de partida s, en un grafo $G = (V, E)$ direccional y con costos:

```
Bellman-Ford(s):
   init(s)
   for k = 1 ... |V|-1:
      for each (u,v) ∈ E:
         reduce(u,v)
```

```
init(s):
   for each v ∈ V:
      d[v] = ∞
      π[v] = null
   d[s] = 0
```

```
reduce(u, v):
   if d[v] > d[u]+ω(u,v):
      d[v] = d[u]+ω(u,v)
      π[v] = u
```

Básicamente, el algoritmo hace $|V|-1$ pasadas por todas las aristas del grafo, tratando de reducirlas (o relajarlas). Sin embargo, las únicas aristas que podrían producir un cambio en `d[]` son aquellas que salen de un vértice cuyo `d[]` cambió en la pasada anterior.

Modifica este algoritmo de manera que intente reducir aristas sólo cuando tenga sentido hacerlo. Para ello, emplea **una cola de vértices** y **un arreglo booleano** de $|V|$ casilleros; además, en lugar de hacer $|V|-1$ pasadas, tu algoritmo debe detenerse cuando la cola quede vacía.

La cola `Q` se usa para guardar los vértices cuyos `d[]` acaban de cambiar.

El arreglo booleano `enQ` es para saber si un vértice está en `Q` y, en ese caso, no guardarlo nuevamente allí

```
Bellman-Ford'(s):
    d[s] = 0 —suponemos que todos los otros d[] son inicialmente ∞
    Q.enqueue(s) —suponemos que Q está inicialmente vacía
    enQ[s] = True —suponemos que todos los otros casilleros de enQ son inicialmente False
    while !Q.empty(): —en lugar de iterar |V|−1 veces, iteramos mientras Q no esté vacía
        u = Q.dequeue()
        enQ[u] = False
        for each v in adj[u]: —reducimos (o tratamos de reducir) cada arista que sale de u
            if d[v] > d[u] + ω(u,v):
                d[v] = d[u] + ω(u,v)
                π[v] = u
                if !enQ[v]: —si pudimos reducir (u,v), entonces guardamos v en Q (si v no está ya en Q)
                    Q.enqueue(v)
                    enQ[v] = True
```

La ventaja de esta versión del algoritmo de Bellman-Ford, que es la que se usa en la práctica, es que es más rápida.

2015-2-Ex-P5–Subestructura óptima, función recursiva (1/2)

5a. Un ladrón entra a una tienda llevando una mochila con capacidad de 10 kg. En la tienda, el ladrón encuentra tres tipos de objetos (aunque hay innumerables objetos de cada tipo): los objetos de tipo 1 pesan 4 kg y tienen un valor de 11; los de tipo 2, pesan 3 kg y valen 7; y los de tipo 3, pesan 5 kg y valen 12. ¿Con cuáles objetos debe llenar la mochila el ladrón para maximizar su valor sin exceder su capacidad? Resuelve este problema empleando **programación dinámica**; en particular:

a) Demuestra que el problema exhibe la propiedad de subestructura óptima.

Subestructura óptima significa que la solución óptima al problema original contiene soluciones óptimas a problemas más pequeños del mismo tipo; en este caso, ya sea mochilas de menor capacidad, o bien sólo uno o dos tipos de objetos, o bien mochilas de menor capacidad y sólo uno o dos tipos de objetos.

La mochila óptima puede contener o no objetos de tipo 1. Hay que analizar ambos casos y quedarse con el que produce el mejor resultado. Supongamos que la mochila óptima contiene al menos un objeto de tipo 1. Esto significa que su valor es 11 más el valor del resto de la mochina. Pero este valor debe corresponder a una mochila óptima de capacidad 6 kg (= 10 kg – 4 kg) con objetos de los tipos 1, 2 y 3: un problema similar al original, pero más pequeño.

Supongamos ahora que la mochila óptima no contiene objetos de tipo 1. Esto significa que es equivalente a una mochila óptima de capacidad 10 kg, pero que sólo contiene objetos de los tipos 2 y 3: nuevamente, un problema similar al original, pero más pequeño.

b) Plantea la solución recursivamente.

Si representamos la solución óptima al problema de una mochila con capacidad C y objetos de los tipos t_1, t_2 y t_3 por $[C, \{t_1,t_2,t_3\}]$, entonces, del análisis de **a**), tenemos que

$[10, \{1,2,3\}] = \max\{ 11 + [6, \{1,2,3\}], [10, \{2,3\}] \}$.

Y para una instancia intermedia cualquiera, gracias a la propiedad de subestructura óptima, suponiendo objetos de tipos $t_j, \ldots, t_k$ con valores $v(t_i)$ y pesos $w(t_i)$,

$[c, \{t_j, \ldots, t_k\}] = \max\{ v(t_j) + [c - w(t_j), \{t_{j+1}, \ldots, t_k\}], [c, \{t_{j+1}, \ldots, t_k\}] \}$

c) Desarrolla la formulación recursiva de la solución, de modo de responder la pregunta anterior.

Si se desarrolla la formulación anterior a partir de la primera ecuación y aplicando la segunda en los pasos intermedios, finalmente obtenemos que la solución óptima es un objeto de tipo 1 y dos de tipo 2: llenan la mochila exactamente y su valor total es 25.

2015-2-Ex-P5–Subestructura óptima, función recursiva (2/2)

5b. Supongamos que tienes los votos emitidos por los estudiantes de la universidad para elegir al presidente de la federación; cada voto contiene simplemente el RUT del candidato. Hay n votos emitidos y sabemos que hay k candidatos, pero no sabemos quiénes son.

a) [**2/3**] Describe completamente un algoritmo y las estructuras de datos correspondientes que permitan determinar al ganador de la elección en tiempo $O(n \log k)$; tus estructuras de datos pueden usar a lo más $O(k)$ memoria.

b) [**1/3**] ¿Qué suposición razonable debes hacer para que el tiempo sea efectivamente $O(n \log k)$?

2015-2-Ex-P7–Floyd-Warshall (1/1)

7. Encuentra los costos de las rutas más cortas entre todos los pares de vértices para el siguiente grafo direccional representado por su matriz de adyacencias, **empleando el algoritmo de Floyd–Warshall**. P.ej., el costo de la arista que va del vértice 1 al vértice 2 es 51; y no hay arista del vértice 3 al vértice 4. En particular, muestra cada una de las siguientes **tres** matrices que produce el algoritmo.

	0	**1**	**2**	**3**	**4**	**5**
0	0	41				29
1		0	51		32	
2			0	50		
3	45			0		38
4			32	36	0	
5		29			21	0

	0	**1**	**2**	**3**	**4**	**5**
0	0	41				29
1		0	51		32	
2			0	50		
3	45	86		0		38
4			32	36	0	
5		29			21	0

	0	**1**	**2**	**3**	**4**	**5**
0	0	41	92		73	29
1		0	51		32	
2			0	50		
3	45	86	137	0	118	38
4			32	36	0	
5		29	80		21	0

	0	**1**	**2**	**3**	**4**	**5**
0	0	41	92	142	73	29
1		0	51	101	32	
2			0	50		
3	45	86	137	0	118	38
4			32	36	0	
5		29	80	130	21	0

2015-1-Ex-P4–Floyd-Warshall (1/1)

4. [**Este problema vale doble**] Considera el siguiente grafo direccional con costos, con vértices *a*, *b*, *c*, *d* y *e*, representado mediante sus listas de adyacencias:

[*a*]: [*b*, 3] – [*c*, 8] – [*e*, –4] [*b*]: [*d*, 1] – [*e*, 7] [*c*]: [*b*, 4] [*d*]: [*a*, 2] – [*c*, –5] [*e*]: [*d*, 6]

El algoritmo de Floyd-Warshall para determinar las rutas más cortas entre todos los pares de vértices en un grafo direccional con costos es el siguiente:

```
D = matriz de adyacencias
for k = 1 … n —n es el número de vértices
    for i = 1 … n —para cada vértice i
        for j = 1 … n —para cada vértice j
            dij = min( dij, dik+dkj )
return D
```

a) Ejecuta el algoritmo de Floyd-Warshall sobre el grafo anterior; muestra el contenido de la matriz D después de cada iteración del índice k, y encuentra tanto las longitudes de las rutas más cortas **como las rutas propiamente tales**.

Ver p. 696 de Cormen et al. [2009].

b) ¿Cómo podemos usar el resultado del algoritmo de Floyd-Warshall para detectar la presencia de un **ciclo con costo acumulado negativo**? Justifica.

Una posibilidad es ejecutar una iteración adicional, con $k = n+1$, y ver si alguno de los valores de la matriz D cambia. Si hay CCANs, entonces el costo de alguna ruta más corta deberá disminuir más allá del "mínimo" entontrado después de n iteraciones.

Lo otro es mirar los valores de la diagonal de D: hay un CCAN si y sólo si alguno de esos valores es negativo. **Demostración**. … (si la necesitan, díganme).

Este algoritmo es un algoritmo de programación dinámica; **muestra que es aplicable al problema**:

c) Demuestra que el problema tiene subestructura óptima.

Trivial. Lo hicimos varias veces en clases.

d) Demuestra que una formulación recursiva de la solución (~~p.ej., a partir de **b***~~[este "hint" no tenía *nada que ver*]) presenta la característica de *subproblemas traslapados*.

La formulación recursiva útil (que también da origen al algoritmo) es la siguiente: Consideremos rutas más cortas que sólo usan los vértices 1, 2, …, k como vértices intermedios. Una ruta más corta de i a j (la *ruta*) puede incluir o no al vértice k; como no sabemos, debemos calcular ambas posibilidades y quedarnos con la mejor (similarmente a como lo hicimos en otros problemas):

- si no lo incluye, entonces la *ruta* es idéntica a la ruta más corta de i a j que sólo usa los vértices 1 a k–1 como vértices intermedios;

- si lo incluye, entonces (por subestructura óptima) la *ruta* es la concatenación de la ruta más corta de i a k con la ruta más corta de k a j, ambas usando sólo los vértices 1 a k–1 como vértices intermedios.

2014-1-Ex-P4–Problema de PD, principio de optimalidad, función recursiva (1/2)

4) Tenemos dos strings, $X = x_1x_2\ldots x_m$ e $Y = y_1y_2\ldots y_n$. Consideremos los conjuntos {1, 2, …, m} y {1, 2, …, n} que representan las posiciones (de los símbolos) en los strings X e Y, y consideremos un *emparejamiento* de estos conjuntos, es decir, un conjunto de pares ordenados tal que cada item aparece en a lo más un par. P.ej., si X = `hola` e Y = `ollas`, entonces los conjuntos son {1, 2, 3, 4} y {1, 2, 3, 4, 5}, un emparejamiento podría ser { (2,1), (3,2), (4,4) } (que empareja letras iguales entre ellas), y otro { (1,5), (4,1) } (que empareja letras extremas entre ellas).

Decimos que un emparejamiento M es una *alineación* si no hay pares que "se crucen": si (j, k), (j', k') $\in M$, y si $j < j'$, entonces $k < k'$; p.ej., el primer emparejamiento anterior es una alineación, pero el segundo no. Dado M, un *gap* (brecha) es una posición de X o Y que no está en M; p.ej., para la alineación anterior, la posición 1 de X, y las posiciones 3 y 5 de Y son gaps.

Finalmente, podemos asociar un costo a una alineación: cada gap incurre un costo fijo $\delta > 0$; además, si para cualquier par de símbolos u, v del alfabeto del que provienen X e Y hay un costo $\alpha(u, v)$ de emparejamiento, entonces cada par (i, j) de M tiene un costo $\alpha(x_i, y_j)$. El *costo de la alineación* es la suma de los costos debidos a sus gaps más los costos debidos a sus emparejamientos. P.ej., el costo de la alineación anterior es $3\delta + \alpha($`o`, `o`$) + \alpha($`l`, `l`$) + \alpha($`a`, `a`$)$.

En estas condiciones, dados dos strings, X e Y, queremos encontrar la *alineación de costo mínimo*. Muestra que este problema puede ser resuelto mediante programación dinámica. En particular,

a) Muestra que —explica por qué o explica cómo— el problema puede ser visto como el resultado de una secuencia de decisiones.

En una alineación de costo mínimo M, ya sea $(m, n) \in M$ o $(m, n) \notin M$; en este último caso, ya sea la posición m de X o bien la posición n de Y no aparecen emparejadas en M. Esta última afirmación se demuestra por contradicción: de lo contrario, habría pares "cruzados" y no tendríamos propiamente una *alineación*, según la definición de más arriba.

Si $(m, n) \in M$, entonces pagamos el costo $\alpha(x_m, y_n)$ y alineamos lo mejor posible $x_1\ldots x_{m-1}$ con $y_1\ldots y_{n-1}$; es decir, $\text{opt}(m,n) = \alpha(x_m, y_n) + \text{opt}(m-1, n-1)$.

Si, en cambio, la posición m de X no está emparejada, pagamos el costo δ del gap y alineamos lo mejor posible $x_1\ldots x_{m-1}$ con $y_1\ldots y_n$; es decir, $\text{opt}(m,n) = \delta + \text{opt}(m-1, n)$. Y similarmente si la posición n de Y no está emparejada, obteniendo $\text{opt}(m,n) = \delta + \text{opt}(m,n-1)$.

b) Muestra que —explica por qué o explica cómo— se verifica el *principio de optimalidad*, cuando se aplica al estado del problema que resulta al tomar una decisión.

De la argumentación de **a**), cuando decimos "y alineamos lo mejor posible".

Si M es una alineación óptima y sacamos el par que contiene las posiciones m y/o n, entonces el resto de los pares, M', es una alineación óptima para los demás símbolos de X e Y. Si no fuera así y hubiera una alineación M'' mejor que M' para el resto de los símbolos, entonces podríamos cambiar M' por M'' en M y obtener una alineación mejor que M, contradiciendo que M sea óptima.

2014-1-Ex-P4–Problema de PD, principio de optimalidad, función recursiva (2/2)

c) A partir de **a**) y **b**), plantea una ecuación recursiva para calcular el costo de la alineación de costo mínimo (la ecuación debe estar planteada, entre otros, en términos de los costos de subalineaciones óptimas); incluye las condiciones de borde —es decir, cuando la recursión ya no aplica más— que permiten resolver la ecuación.

Para $i \geq 1$ y $k \geq 1$,

$\text{opt}(i, k) = \min\{\ \alpha(x_i, y_k) + \text{opt}(i–1, k–1),\ \delta + \text{opt}(i–1, k),\ \delta + \text{opt}(i, k–1)\ \}$

con $\text{opt}(i, 0) = \text{opt}(0, i) = i\delta$, para todo i (ya que la única forma de alinear una palabra de i letras con una de 0 letras es usar i gaps)

2013-2-I3-P4–Bellman-Ford (1/1)

4) Considera el ***algoritmo de Bellman-Ford***, que determina las rutas más cortas desde s a cada uno de los vértices v de un grafo direccional $G = (V, E)$:

```
boolean BellmanFord( Vértice s )
    Init(s)
    for (k = 1; k < |V|; k++)
       for (cada arista (u,v) ∈ E) Reduce(u,v)
    for (cada arista (u,v) ∈ E)
       if (d[v] > d[u] + ω(u,v)) return false
    return true
```

Aplícalo al grafo representado por la siguiente matriz de adyacencias, para determinar las longitudes de las rutas más cortas desde el vértice 1 (las casillas vacías representan ∞):

	1	2	3	4	5	6	7
1	0	6	5	5			
2		0			–1		
3		–2	0		1		
4			–2	0		–1	
5					0		3
6						0	3
7							0

Muestra los valores del vector **d** después de cada iteración.

Respuesta:

$k = 1$:	0	6	5	5	∞	∞	∞
$k = 2$:	0	3	3	5	5	4	∞
$k = 3$:	0	1	3	5	2	4	7
$k = 4$:	0	1	3	5	0	4	5
$k = 5$:	0	1	3	5	0	4	3
$k = 6$:	0	1	3	5	0	4	3

Si bien las respuestas podrían diferir en cuanto a la progresión de los valores de **d**, las respuestas correctas deben cumplir lo siguiente:

- Para $k = 1$, **d** debe corresponder a los costos de las aristas (directas) desde el vértice 1 a los otros vértices.
- El algoritmo ha encontrado todas las distancias mínimas cuando $k = 5$ (ya que la ruta más corta que tiene más aristas tiene 4 aristas) y, por lo tanto, **d** no cambia de $k = 5$ a $k = 6$.
- Los valores de **d** no pueden aumentar a lo largo de las iteraciones.

2013-2-Ex-P5–Subestructura óptima, función recursiva, algoritmo PD (1/1)

5) Tenemos una máquina que puede procesar tareas, de a una a la vez, y tenemos un conjunto de n tareas que procesar. Además, tenemos un tiempo total T para usar la máquina y sabemos que la tarea i ocupa un tiempo t_i para ser procesada, $i = 1, \ldots, n$. Queremos encontrar un subconjunto de tareas tal que todas puedan ser procesadas por la máquina en el tiempo T y al mismo tiempo maximicen el tiempo efectivo de uso de la máquina.

a) [**1 pt.**] Muestra (basta con un ejemplo) que este problema no puede ser resuelto en general usando un algoritmo codicioso basado en procesar las tareas en orden decreciente de tiempo t_i (tampoco se podría en orden creciente).

Si T es par y tienes tres tareas que toman tiempos T/2, T/2 y T/2 + 1. Al ordenarlas de mayor a menor tiempo, quedan T/2 + 1, T/2, T/2; y al elegirlas en ese orden sólo alcanzas a procesar la primera, ocupando la máquina durante un tiempo T/2 + 1. Pero la solución óptima es procesar las otras dos tareas, cada una de las cuales toma tiempo T/2 y por lo tanto ocupan la máquina durante un tiempo total T/2 + T/2 = T.

b) [**2 pts.**] Justifica que este problema cumple la propiedad de subestructura óptima.

La propiedad de subestructura óptima significa que la solución óptima al problema incluye soluciones óptimas a problemas del mismo tipo pero más pequeños; "más pequeños" en este caso significa menos tareas y/o menos tiempo total T. Esto lo podemos ver así:

La solución óptima puede incluir o no la tarea 1.

Si la incluye, entonces observamos que si sacamos esta tarea de la solución, el resto de las tareas que pertenecen a la solución son una solución óptima al problema (más pequeño) de elegir las tareas que maximizan el tiempo efectivo de uso de la máquina de entre las tareas 2 a la n, cuando el tiempo total posible es $T - t1$.

Por el contrario, si no la incluye, entonces la solución óptima es igual a la que habríamos obtenido si las tareas a progra-mar en el tiempo total T fueran sólo las tareas 2 a la n, es decir, sin incluir la tarea 1.

c) [**2 pts.**] Plantea una formulación recursiva del valor de la solución óptima, incluyendo condiciones de borde.

Definamos como opt(p, q, t) el problema de encontrar el subconjunto óptimos de tareas de entre las tareas p, p+1, …, q, cuando el tiempo total disponible en la máquina es t (así, el problema original es opt(1, n, T)). Entonces, generalizando el argumento de b),

$$opt(k, n, t) = max\{\, opt(k+1, n, t),\ opt(k+1, n, t - t_{k+1}) + t_{k+1} \,\},$$

sabiendo que $opt(n+1, n, t) = 0$, *si* $t \geq 0$, *y* $opt(n+1, n, t) = -\infty$, *si* $t < 0$.

d) [**1 pt.**] Escribe un algoritmo de programación dinámica que aproveche la formulación recursiva anterior para resolver el problema.

El algoritmo debe ir llenando una tabla M con los valores de opt(k, n, t):

```
M[0..n][0..T]
for each t = 0,1,…,T: M[0][t] = 0
for k = 1, 2, …, n
    for t = 0,…, T
        —usar la recurrencia de c) para calcular M[k][t], que corresponde a opt(k,n,t)
return M[n][T]
```

2013-1-I3-P4-b–Bellman-Ford (1/1)

b) Considera el *algoritmo de Bellman-Ford* y, en particular, el número de aristas en cada una de las rutas más cortas. Sea m el número de aristas de la ruta más corta con más aristas. Modifica el algoritmo, de manera que termine después de m+1 pasadas (en lugar de $|V|$ ~~+1~~ pasadas), *aún cuando m no se conozca por adelantado*.

```
boolean BellmanFord( Vértice s )
{
  Init(s)
  for (k = 1; k < |V|; k++)
    for (cada arista (u,v) ∈ E) Reduce(u,v)
  for (cada arista (u,v) ∈ E)
    if (d[v] > d[u] + ω(u,v)) return false
  return true
}
```

Resp. *Bellman-Ford* itera $|V|-1$ veces (primer `for` exterior), porque la ruta más corta con más aristas tiene a lo más $|V|-1$ aristas (si la ruta tuviera más aristas, contendría un ciclo). El segundo `for` exterior hace una última pasada por todas las aristas, para verificar que en G no haya ciclos con costo acumulado negativo. (Este sería el caso si esta última pasda pudiera "mejorar" alguna ruta más corta.)
Por lo tanto, bajo el supuesto de que la ruta más corta con más aristas tiene m aristas, la modificación al algoritmo consiste en cambiar el primer `for` exterior por un `while` que itere mientras sea posible mejorar alguna ruta más corta, es decir, mientras la ejecución de algún `Reduce` haya producido una mejora. (El segundo `for` exterior desaparece, porque si G contuviera ciclos con costo acumulado negativo, el `while` no se detendría jamás.)

```
BellmanFord-m( Vértice s )
{
  Init(s)
  mejora = true
  while ( mejora )
  {
    mejora = false
    for ( cada arista (u,v) ∈ E )
          if ( d[v] > d[u]+ω(u,v) )
          {
                d[v] = d[u]+ ω(u,v)
                π[v] = u
                mejora = true
          }
  }
}
```

Tiempo: 120 minutos

2013-1-Ex-D–Propiedades de PD, Floyd-Warshall (1/2)

D) Considera el problema de determinar las **rutas más cortas entre todos los pares de vértices** de un grafo direccional $G = (V, E)$ con costos en las aristas.

13) Enuncia las dos propiedades características de los problemas de optimización que pueden ser resueltos mediante programación dinámica.

Las dos propiedades características son: Propiedad de Subestructura óptima y Propiedad de subproblemas traslapados.

- Subestructura óptima: La solución óptima al problema original puede ser construida con soluciones óptimas de subproblemas.
- Subproblemas traslapados: Si uno utiliza un algoritmo recursivo para resolver el problema, entonces existirá al menos un subproblema que será resuelto varias veces.

2013-1-Ex-D–Propiedades de PD, Floyd-Warshall (2/2)

14) Muestra que este problema cumple ambas propiedades.

Para la propiedad de subestructura óptima, suponga que p(u,v) es el camino de menor costo desde u hasta v. Sea x el nodo inmediatamente anterior a v en el camino p (por lo que (x,v) es una arista). Entonces suponga que p(u,x) no es el camino más corto desde u hasta x. Entonces existe otro camino p'(u,x) que sí es óptimo. Luego, agregue la arista (x,v) a este nuevo camino. Y por lo tanto, tenemos que:

$$w(p'(u,x))+w(x,v) < w(p(u,x))+w(x,v)$$

Por lo que el camino p' seguido por (x,v) es mucho mejor que el camino p(u,v) original. Luego, p(u,v) no es óptimo y hay una contradicción por suponer que no tiene subestructura óptima. Por lo tanto, este problema exhibe subestructura óptima.

Así, para ir desde u hasta v, considere la siguiente recursión, donde d(x,y) es el costo del camino más corto desde x hasta y:

d(x,x) = 0
d(x,y) = w(x,y) si (x,y) es una arista
d(x,y) = +infinito si y no es alcanzable desde x
d(x,y) = min{d(x,u)+d(u,y) | u está en V}

Para la propiedad de subproblemas traslapados, suponga que quiere encontrar el valor de d(x,y) y para ello, necesita encontrar el valor de d(x,u) y d(u,y). También necesita encontrar el valor de d(x,a) y d(a,y). Para encontrar el valor de d(x,u) necesita de d(x,a) y d(a,u). Luego tiene que calcular al menos dos veces el valor de d(x,a) y este problema tiene subproblemas traslapados.

15) Plantea un algoritmo eficiente para resolverlo; justifica que tu algoritmo es eficiente.

(No basta con decir que el algoritmo corresponde al de Floyd-Warshall, debe plantearse el algoritmo)

Suponga que tenemos una matriz W donde W(i,j) indica el costo de ir desde el nodo i hasta el nodo j. Sea D otra matriz, inicialmente vale D=W y que tenemos N nodos. El algoritmo es:

```
D=W;
for k=1..N{
  for i=1..N{
    for j=1..N{
      if(D(i,j)>D(i,k)+D(k,j))
        D(i,j) = D(i,k)+D(k,j)
    }
  }
}
```

El algoritmo toma tiempo O(n^3). Es más eficiente que ejecutar N veces Dijkstra (que toma tiempo O((V+E)log V)) y más eficiente que ejecutar N veces Bellman-Ford (que toma tiempo O(VE))

16) El ***diámetro*** de *G* es la más larga de las rutas más cortas en *G*. A partir de tu algoritmo para 15), da un algoritmo eficiente que imprima la *secuencia de vértices* del diámetro de *G*.

Si uno simplemente ejecuta el algoritmo anterior, obtiene una matriz *D* con las distancias más cortas entre todos los pares de vértices; el mayor valor en esta matriz es la magnitud del diámetro del grafo; pero falta la secuencia de vértices.

Para esto, entre otras posibilidades, [Cormen et al., 2001] sugiere construir una matriz *P* iterativamente, similarmente y en paralelo a la de las distancias más cortas, pero que represente las rutas propiamente tales. Si en la iteración *k* me conviene incluir el vértice *k* para ir de *u* a *v*, entonces *P*[*u*, *v*] = *P*[*k*, *v*] de la iteración *k*–1; de lo contrario, *P*[*u*, *v*] = *P*[*u*, *v*] de la iteración *k*–1. Así, al finalizar Floyd-Warshall, *P*[*u*, *v*] contiene el vértice anterior a *v* en la ruta más corta de *u* a *v*.

2012-1-I3-P3–Análisis de implementación de PD en un problema (1/2)

Pregunta 3 – Equilibrio perfecto (20 pts)

El asombroso Equilibrini, un famoso malavarista, está preparando su nuevo espectáculo llamado “Equilibrio perfecto”. En este espectáculo, hay n elementos a su alrededor, cada uno con pesos p_0, p_1, … p_{n-1}. Mientras el asombroso Equilibrini se mantiene en su monociclo, le pide al público que elija algún elemento y uno de sus asistentes lo toma y lo ubica en una de sus manos (o sobre el elemento que tenga en esa mano). De esta forma, va formando dos torres en perfecto equilibrio sobre el pequeño monociclo.

2012-1-I3-P3–Análisis de implementación de PD en un problema (2/2)

Bueno, no todo es tan mágico como parece: en realidad es Equilibrini quien elige los elementos, pues el público normalmente pide casi todo lo que está a la vista, y él simplemente remarca el elemento que previamente había decidido. Esto lo hace por una buena razón, Equilibrini puede mantener las dos torres sin problema siempre y cuando, la diferencia entre el peso de ambas no sea mayor a 4 en ningún momento. Además, desea maximizar el asombro del público, el cual se puede medir como la cantidad de elementos que Equilibrini llega a equilibrar.

Ayuda al asombroso Equilibrini, creando un algoritmo que determine qué elementos debe seleccionar y en qué orden, de modo de maximizar lo asombroso del espectáculo.

El input consiste en una línea con n (el número de pesos) y luego una línea con los pesos p_0, p_1, … p_{n-1}. El output consiste en el nivel de asombro alcanzado.

Ejemplo Input	Output:
10 2 5 9 4 4 1 3 2 7 4	8

Este problema posiblemente se puede resolver con algún algoritmo greedy, pero convertir esta posibilidad en una afirmación requeriría acompañar el algoritmo de una demostración de correctitud.

Como ya lo he dicho en múltiples ocasiones, prefiero un algoritmo de programación dinámica que sería así:

Considerar el siguiente método:

int probar(int elementos, bool* agregados, int n, int* pesosManos, int elementosManos);*

el cual recibe la lista con los elementos que hay que agregar (peso y si está agregado o no), y los pesos en las mano derecha e izquierda (puntero de largo 2).

a) Entonces, si sólo falta 1 elemento por agregar ver si es posible agregarlo a la mano derecha, sino, a la izquierda, sino descartar el agregar este elemento y actualizar el pesosManos. Retornar el nuevo número de elementos (elementosManos +1 o +0, según el caso).

b) Si faltan más elementos por agregar, entonces tomar el primer elemento, y si es posible agregarlo a la mano izquierda (actualizando las variables respectivas) y llamar a "probar" sin este elemento. Repetir de manera análoga agregando en la mano derecha. Luego elegir el caso de probar mayor y retornarlo, actualizando las variables respectivas.

Para enmazcarar el algoritmo recursivo, crear un método "fachada" que reciba los elementos, inicialice el arreglo de agregados y el arreglo de pesos manos, llame a probar con estas variables y elementosManos=0, y una vez obtenido el máximo, elimine la memoria solicitada y retorne el máximo.

2011-2-I3-P3–Ciclo en grafo direccional, Bellman-Ford (1/1)

3. Podemos usar las discrepancias en las tasas de cambio de monedas para transformar una unidad de una moneda en más de una unidad de la misma moneda. P.ej., si un euro compra 1.38 dólares, un dólar compra 500 pesos, y un peso compra 0.0015 euros, entonces, a partir de un euro podemos obtener $1.38 \times 500 \times 0.0015 = 1.035$ euros.

Supón que tienes n monedas $\langle c_1, c_2, \ldots, c_n \rangle$ y una tabla R de $n \times n$ tasas de cambio: una unidad de la moneda c_j compra $R[j][k]$ unidades de la moneda c_k. Queremos determinar si existe o no una secuencia de monedas $\langle c_a, c_b, \ldots, c_m \rangle$ tal que

$R[a][b] \times R[b][c] \times ... \times R[m][a] > 1$.

a) Muestra que este problema puede plantearse como un problema de determinar si en un grafo direccional existe un ciclo con costo total negativo. [Ayuda: Recuerda que $x > 1 \Rightarrow 1/x < 1$ y que $\log(ab) = \log(a) + \log(b)$.]

Construimos el siguiente grafo:

- cada vértice corresponde a una moneda;
- el costo de la arista dirigida (c_j,c_k) es $-\log(R[j][k])$.

¿Por qué? Si aplicamos la ayuda a la fórmula de arriba, queda, primero

$R[a][b] \times R[b][c] \times ... \times R[m][a] > 1 \Rightarrow 1/(R[a][b] \times R[b][c] \times ... \times R[m][a]) < 1.$

y luego,

$\log(1/(R[a][b] \times R[b][c] \times ... \times R[m][a])) < 0 \Rightarrow \log(1/ R[a][b]) + \log(1/ R[b][c]) + \ldots + \log(1/ R[m][a]) < 0$

$\Rightarrow -\log R[a][b] - \log R[b][c] - \ldots -\log R[m][a] < 0$

Es decir, si hay una secuencia de monedas como la que buscamos, entonces este grafo tiene un ciclo con costo total negativo.

b) Da un algoritmo para imprimir la secuencia, si existe.

Primero, hay que detectar el ciclo, para lo cual podemos aplicar el algoritmo de Bellman-Ford. Pero hay que asegurarse que partamos de un vértice desde el cual se pueda llegar al ciclo. Una posibilidad es agregar un nuevo vértice v al grafo con aristas direccionales con costo 0 a todos los otros vértices (las monedas). Ahora, en el **`for`** final del algoritmo, simplemente detectamos el primer vértice u cuyo valor d mejore; y de ahí seguimos los valores π hasta regresar a u.

2010-2-I3-P2–Bellman-Ford (1/1)

2) Considera el ***algoritmo de Bellman-Ford***, que determina las rutas más cortas desde s a cada uno de los vértices v de un grafo direccional $G = (V, E)$, y considera el número de aristas en cada una de estas rutas más cortas. Sea m el número de aristas de la ruta más corta con más aristas. Modifica el algoritmo, de manera que termine después de m+1 pasadas (en lugar de $|V| + 1$ pasadas), aún cuando m no se conozca por adelantado.

```
boolean BellmanFord( Vértice s )
{
   Init(s)
   for (k = 1; k < |V|; k++)
      for (cada arista (u,v) ∈ E) Reduce(u,v)
   for (cada arista (u,v) ∈ E)
      if (d[v] > d[u] + ω(u,v)) return false
   return true
}
```

Respuesta:

Bellman-Ford itera $|V|-1$ veces (primer **for** exterior), porque la ruta más corta con más aristas tiene a lo más $|V|-1$ aristas (si tuviera más aristas, contendría un ciclo). El segundo **for** exterior hace una última pasada por todas las aristas, pero solo para verificar que en G no haya ciclos con costo acumulado negativo. (Este sería el caso si esta última pasda pudiera "mejorar" alguna ruta más corta.)

Por lo tanto, bajo el supuesto de que la ruta más corta con más aristas tiene m aristas, la modificación al algoritmo consiste en cambiar el primer **for** exterior por un **while** que itere mientras sea posible mejorrar alguna ruta más corta, es decir, mientras la ejecución de algún **Reduce** haya producido una mejora. (El segundo **for** exterior desaparece, porque si G contuviera ciclos con costo acumulado negativo, el **while** no se detendría jamás.)

```
BellmanFord-m( Vértice s )
{
   Init(s)
   mejora = true
   while ( mejora =  true )
   {
      mejora = false
      for ( cada arista (u,v) ∈ E )
         if ( d[v] > d[u]+ω(u,v) )
         {
            d[v] = d[u]+ ω(u,v)
            π[v] = u
            mejora = true
         }
   }
}
```