

Pontificia Universidad Católica de Chile
Departamento de Ciencia de la Computación
IIC2283 - Diseño y Análisis de Algoritmos

Ayudantía 6 - Repaso I1

Septiembre de 2025

Caetano Borges, Isabella Cherubini

---

1. Programación Dinámica

Diego dice que un arreglo de enteros positivos es *bonito* si tiene la forma de una serie de bloques, cada uno empezando con su largo, es decir, primero viene el largo del bloque, y luego sus elementos. Por ejemplo, los arreglos `[3, 3, 4, 5, 2, 6, 1]` y `[1, 8, 4, 5, 2, 6, 1]` son *bonitos*, mientras que `[1], [1, 4, 3], [3, 2, 1]` no lo son.

En una operación, Diego puede eliminar cualquier elemento del arreglo.

Desarrolle y escriba el pseudocódigo de un algoritmo eficiente que utilice programación dinámica para determinar el mínimo número de operaciones que Diego debe realizar para transformar cualquier arreglo en un arreglo *bonito*.

Solución
Una técnica que suele ser útil para resolver problemas con programación dinámica es pensar inductivamente: asumir que tenemos la solución para algún subproblema de menor tamaño, y utilizar esta solución para resolver el problema de mayor tamaño. Luego, necesitamos encontrar un caso base para solucionar el subproblema más básico, pero si elegimos bien el subproblema, esto suele ser más fácil.
En este caso, podemos ponernos en el siguiente escenario:
Sea $A[1 \dots n]$ nuestro arreglo. Planteemos el siguiente subproblema: encontrar el mínimo de operaciones que necesitamos para que el arreglo $A[i \dots n]$ sea *bonito*, con $1 \leq i \leq n$. Nuestro objetivo final es computar esta respuesta con $i = 1$. Sin embargo, por ejemplo, es muy fácil computar la respuesta cuando $i = n$: siempre es 1, ya que nuestro arreglo es de enteros positivos, es decir, no puede haber un 0 para que haya un bloque de largo 0. Esto nos da un caso base que podría ser útil.
Buscamos la respuesta para un i. Asumamos que tenemos la respuesta para $A[j \dots n]$ con $i < j \leq n$. Hay dos posibilidades: (1) sacamos el elemento i, o (2) no lo sacamos.

- Si sacamos el elemento i, entonces la solución para $A[i \dots n]$ será 1 (por la operación de eliminación que acabamos de hacer) más la solución para $A[i+1 \dots n]$.
- Si no sacamos el elemento i, como este es el primer elemento del arreglo $A[i \dots n]$, debe ser el largo de un bloque. Luego, la solución es 0 (no eliminar no tiene costo) más la solución para $A[i+a_i+1 \dots n]$.

Con esto tenemos una recurrencia con un número constante (2) de recursiones hacia subproblemas más pequeños. Lo que haremos para resolver para cada i será tomar el mejor de los dos casos, es decir, computar ambas soluciones (en $O(1)$ gracias a programación dinámica!) y tomar el mínimo.
Ahora tenemos que encontrar los casos base. Ya encontramos el caso para $i = n$. Pero es posible que $i + a_i + 1 > n$ en la segunda recurrencia, por lo que debemos definir la respuesta para $i > n$ también. Notemos que si $i + a_i = n$, entonces el bloque termina justo en el final del arreglo, con lo que la solución al subproblema respectivo (cuando $i + a_i + 1 = n + 1$) debería ser 0. Además, no podemos "pasarnos" con el largo del bloque, es decir, no podemos tener una solución en que $i + a_i > n$, con lo que los casos en que $i + a_i + 1 > n + 1$ deberían estar "prohibidos". Una forma elegante de hacerlo es asignarles una respuesta que siempre será la peor, y así nunca los tomaremos al tomar el mínimo; en este caso podemos usar ∞. Con ello, obtenemos la siguiente función solución:

$$f(i) = \begin{cases} 1 & i = n \\ 0 & i = n+1 \\ \infty & i > n+1 \\ \text{mín}\{1 + f(i+1), f(i+a_i+1)\} & 1 \leq i < n \end{cases}$$

2. Programación Dinámica

Los dragones han capturado al príncipe en la casilla inferior derecha de un calabozo. El calabozo consiste en $m \times n$ cuartos que se encuentras dispuestos en una grilla 2D. Te encuentras en la casilla superior izquierda y debes avanzar por el calabozo para rescatarlo.
Tienes una vida inicial representada por un número positivo h, si es que tu vida llega a ser igual o menor a 0 no podrás seguir avanzando.
Cada cuarto tiene un valor $n_{i,j}$. Algunos de los cuartos están protegidos por dragones (representados por números negativos), por lo que pierdes $n_{i,j}$ vida al pasar por ellos. Otros cuartos contienen pociones para sanarte (representados por números positivos), por lo que ganas $n_{i,j}$ al pasar por ellos.
Para rescatar al príncipe solo puedes moverte a la casilla derecha o inferior.
Define cual es el menor valor inicial de h tal que puedas rescatar al príncipe dadas estas condiciones.

Solución La primera intuición para resolver el problema es intentar resolverlo al revés (es decir, iniciando desde la celda donde se encuentra el príncipe), ya que podemos ir determinando cual es el mínimo de vida necesaria para poder llegar a esa celda. Por ejemplo, si la celda final tiene un valor de -5, sabemos que debemos llegar a ella con al menos 6 puntos de vida.
Con eso, podemos expadir la idea a una celda (i, j), sabemos que podemos movernos tanto hacia la derecha $(i, j+1)$ como hacia abajo $(i+1, j)$. Dado esto, debemos tomar en consideración que siempre se elegirá el camino que requiera tener menor vida inicial. Con esto, podemos plantear la siguiente solución:

$$dp(i,j) = \begin{cases} max(1, 1 - A[i][j]) & i = n, j = m \\ max \begin{cases} 1 \\ -A[i][j] + min \begin{cases} dp(i+1, j) \\ dp(i, j+1) \end{cases} \end{cases} & e.o.c. \end{cases}$$

3. Cotas Inferiores

Un arreglo $A[1..n]$ está k-ordenado si puede dividirse en k bloques, cada uno de tamaño n/k (asumiendo que n es múltiplo de k), tal que los elementos de cada bloque son menores que los del bloque siguiente. Los elementos dentro de cada bloque no necesitan estar ordenados. Por ejemplo, el arreglo de 20 elementos:

$$< 2, 1, 3, 4, 5, |7, 6, 8, 10, 9, |14, 15, 12, 13, 11, |16, 17, 18, 20, 19 >$$

está 4-ordenado (se han agregado las barras verticales para demarcar los bloques de manera más clara), ya que puede dividirse en 4 arreglos de tamaño $20/4 = 5$ cada uno, y los elementos de un bloque son menores que los del bloque siguiente.

a) Demuestre formalmente (usando árboles de decisión) que cualquier algoritmo basado en comparaciones que reciba un arreglo de n elementos como entrada y produzca como salida un arreglo k-ordenado (para un k indicado) requiere $\Omega(nlog_2(k/e))$ comparaciones en el peor caso (en donde e = 2,71828... es la base del logaritmo natural)

b) ¿Qué pasos seguiría para demostrar que la complejidad del problema de k-ordenar un arreglo es $\Theta(nlog_2(k/e))$ comparaciones?

Solución

a) Cualquier algoritmo que resuleva el problema de producir un arreglo k-ordenado puede ser pensado como un árbol de decisión tal que:
   - Los nodos internos del árbol corresponden a las comparaciones entre elementos del arreglo hechas por el algoritmo.

- Las ramas del árbol corresponden a las decisiones tomadas en base a las comparaciones hechas por el algoritmo
- Estando en un nodo del árbol, el estado del algoritmo hasta ese punto de la ejecución se mantiene de forma implícita por el camino que se ha seguido.
- Los nodos externos del árbol corresponden a las distintas salidas del algoritmo. El árbol debe una cantidad de nodos externos $\geq$ a la cantidad de posibles salidas distintas.

La clave para calcular la cota inferior está en el último punto, por lo que vamos a determinar la cantidad de salidas distintas que cualquier algoritmo que resuelva el problema debería ser capaz de producir. Dado que la cota inferior que se pide demostrar es $\Omega(nlog_2(k/e))$, se sospecha que hay que usar el coeficiente multinomial.

Aquí es importante entender que cualquier algoritmo que k-ordene un arreglo debe ser capaz de indicar en cuál de los k bloques deber ser ubicado cada uno de los n elementos del arreglo A. Más formalmente, debemos producir como salida un arreglo $P[1..n]$ tal que $P[i] \in \{1, ..., k\}$ indica en cuál de los k bloques debe ser ubicado el elemento $A[i]$, para $1 \leq i \leq n$. Dado que cada bloque debe tener n/k elementos, en P habrá n/k entradas con valor 1, n/k entradas con valor 2, y así siguiendo, hasta n/k entradas con valor k. La cantidad de arreglos distintos de este tipo está dado por el coeficiente multinomial: Para $\Sigma_{i=1}^{l} v_i = m$ se tiene

$$\binom{n}{\frac{n}{k}, \frac{n}{k}, ..., \frac{n}{k}} = \frac{n!}{\frac{n}{k}!, \frac{n}{k}!, ..., \frac{n}{k}!} = \frac{n!}{(\frac{n}{k}!)^k}$$

Esa es la cantidad de distintas salidas que cualquier algoritmo que resuelve el problema debe ser capaz de producir. Por lo tanto, cualquier árbol de decisión que resuelve el problema debe tener al menos esa cantidad de hojas. Como consecuencia, cualquiera de dichos árboles tiene altura

$$h \geq \lceil log_2 \frac{n!}{(\frac{n}{k}!)^k} \rceil$$

Manipulando esta expresión y usando la aproximación de Stirling para $n!$

tenemos:

$$
\begin{aligned}
h &\geq \lceil log_2 \frac{n!}{(\frac{n}{k}!)^k} \rceil && (1)\\
&\geq log_2 \frac{n!}{(\frac{n}{k}!)^k} && (2)\\
&= log(n!) - log(((n/k)!)^k) && (3)\\
&\geq log(n!) - log(((n/k)^{n/k})^k) && (4)\\
&= log_2(n!) - nlog_2 \frac{n}{k} && (5)\\
&= log_2(n!) - nlog_2 n + nlog_2 k && (6)\\
&\geq log_2(n/e)^n - nlog_2 n + nlog_2 k && (7)\\
&= nlog_2 n - nlog_2 e - nlog_2 n + nlog_2 k && (8)\\
&= nlog_2 k - nlog_2 e && (9)\\
&= nlog_2(\frac{k}{e}) && (10)
\end{aligned}
$$

Dado que la altura del árbol de decisión corresponde a la cantidad de comparaciones realizadas por un algoritmo en el peor caso, tenemos que $\Omega(nlog_2(k/e))$ es una cota inferior para el problema de k-ordenar un arreglo.

b) Para demostrar que la complejidad del problema es $\Theta(nlog_2(k/e))$ comparaciones hay que:
   (I) Demostrar que $\Omega(nlog_2(k/e))$ comparaciones es una cota inferior del problema (ya hecho en la parte anterior)
   (II) Demostrar que $O(nlog_2(k/e))$ comparaciones es una cota superior del problema. Para esto, se debe diseñar un algoritmo cuyo tiempo de ejecución de peor caso es $O(nlog_2(k/e))$ comparaciones.