Complejidad de Problemas y Cotas Inferiores

IIC2283 – Diseño y Análisis de Algoritmos

Diego Arroyuelo
diego.arroyuelo@uc.cl

Departamento de Ciencia de la Computación
Pontificia Universidad Católica de Chile

2025-2

Complejidad de Problemas y Cotas Inferiores

En la mayoría de las clases del curso, estaremos preocupados en encontrar algoritmos eficientes para resolver un problema en particular

También analizar y comparar dichos algoritmos

Sin embargo, en la algoritmia es también importante decir que ciertos problemas no pueden ser resueltos tan rápidamente como uno quisiera

En muchos casos permite entender que ciertos algoritmos son la solución más eficiente que se puede tener para el problema

Complejidad de Problemas y Cotas Inferiores

Pregunta

Sobre un modelo de computación arbitrario, ¿Es posible resolver cualquier problema tan eficientemente como quisiéramos?

- La respuesta es, obviamente, **no**
- Para cada modelo de computación dado, cada problema tiene un complejidad inherente asociada

Complejidad de un Problema $\mathcal{P}$

Es una medida de la cantidad de trabajo computacional (o pasos discretos) **necesarios** para resolver $\mathcal{P}$ (usando un algoritmo).

Complejidad de Problemas y Cotas Inferiores

Inicialmente, cuando se comienza a estudiar, la complejidad de un problema $\mathcal{P}$ es desconocida

A medida que se descubren nuevos algoritmos para $\mathcal{P}$ que mejoran progresivamente el tiempo de ejecución de otros algoritmos:

- Estos se acercan (por arriba) a la complejidad del problema
- Descubrimos que el problema es cada vez más fácil de resolver de lo que se pensaba

Pero, ¿Cómo sabemos si, en un punto dado, el tiempo de ejecución de un algoritmo coincide con la complejidad de problema, o no?

Saber esto es importante, porque implica que es imposible diseñar algoritmos más eficientes para ese problema

El concepto clave aquí serás las cotas inferiores

Complejidad de Problemas y Cotas Inferiores

Complejidad de Peor Caso de un Problema

Definimos la complejidad de peor caso de un problema $\mathcal{P}$ sobre instancias de tamaño n como el tiempo de ejecución de peor caso del algoritmo más eficiente que lo resuelve:

$$T_{\mathcal{P}}(n) \equiv \min_{\mathcal{A} \text{ resuelve } \mathcal{P}} \{T_{\mathcal{A}}(n)\}$$

Aunque esta definición es clara y precisa, implica conocer **todos** los algoritmos que resuelven a $\mathcal{P}$, lo cual es imposible en la mayoría de los casos interesantes

Complejidad de Problemas y Cotas Inferiores

Mientras la complejidad de un problema $\mathcal{P}$ es todavía desconocida, en general nos conformamos con poder acotar dicha complejidad

Hablamos de cotas inferiores y superiores de la complejidad

Estas cotas nos dan **un rango** dentro del cual se encuentra la verdadera complejidad del problema

Es común que esas cotas mejoren con el tiempo, como resultado de investigaciones asociadas a los problemas

Complejidad de Problemas y Cotas Inferiores

Descubrir un nuevo algoritmo $\mathcal{A}$ que resuelve un problema $\mathcal{P}$ de forma más eficiente que las conocidas, inmediatamente implica una mejor cota superior para $T_{\mathcal{P}}(n)$

Sin embargo, el estudio de cotas inferiores es, en general, más complicado que el de cotas superiores

Para probar que $T_{\mathcal{P}}(n) \in \Omega(f(n))$ debemos:

- Demostrar que todo posible algoritmo para $\mathcal{P}$ no puede hacer menos que una cierta cantidad de operaciones en el peor caso
- Aún cuando muchos de estos algoritmos no han sido descubiertos todavía

Complejidad de Problemas y Cotas Inferiores

La complejidad de un problema $\mathcal{P}$ se mantiene desconocida hasta que ambas cotas convergen

Eso significa hasta el momento en que el tiempo de ejecución $O(f(n))$ de un algoritmo $\mathcal{A}$ (en el peor caso) coincide con una cota inferior $\Omega(f(n))$ demostrada para $\mathcal{P}$

En ese caso, decimos:

- Que la complejidad del problema $\mathcal{P}$ es $\Theta(f(n))$
- Que el algoritmo $\mathcal{A}$ es óptimo

Demostrando cotas inferiores: Árboles de decisión

Desafortunadamente, no es fácil ni práctico considerar todos los posibles algoritmos para resolver un problema dado

Al probar cotas inferiores, primero debemos especificar precisamente qué clase de algoritmos consideraremos y precisamente qué tipo de operaciones se van a considerar al medir el tiempo de ejecución

Esta especificación es llamada el modelo de computación

En particular, estudiaremos el modelo de *árboles de decisión*

Demostrando cotas inferiores: Árboles de decisión

Un modelo asociado a los árboles de decisión es el modelo de comparaciones.

- El computo principal de un algoritmo puede ser realizado solo mediante comparaciones entre elementos de la entrada
- El tiempo de ejecución de este modelo se mide en cantidad de comparaciones hechas por el algoritmo

Muchos problemas importantes pueden resolverse comparando elementos de la entrada:

- Búsqueda en un conjunto ordenado
- Ordenamiento
- Selección

Demostrando cotas inferiores: Árboles de decisión

Un árbol de decisión binario es un modelo de computación en el cual un algoritmo particular es representado mediante un árbol con las siguientes características:

- **Nodos internos**: representan las comparaciones de elementos hechas por el algoritmo.
- **Subárbol izquierdo de un nodo interno**: representa el cómputo a realizar si la comparación del nodo actual es verdadera
- **Subárbol derecho de un nodo interno**: representa el cómputo a realizar si la comparaci´on del nodo actual es falsa
- **Nodos externos**: representan las posibles salidas del algoritmo

Demostrando cotas inferiores: Árboles de decisión

El modelo de árboles de decisión no permite iterar

La idea es implementar las iteraciones mediante un árbol de **if**s anidados

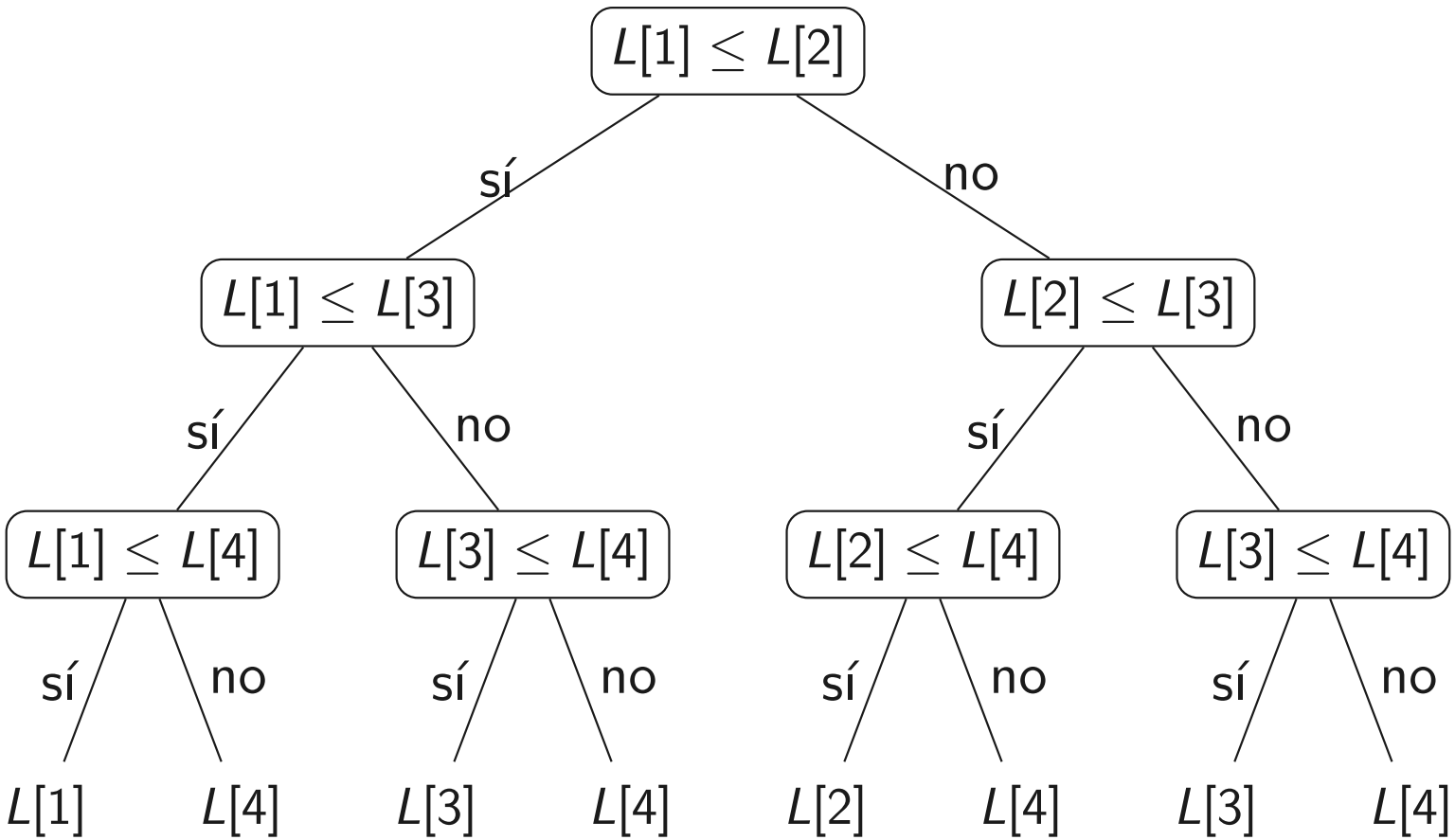
Las hojas del árbol son los **return** del algoritmo

Para cada $n \in \mathbb{N}$ definimos un árbol de decisión $\mathcal{T}_{\mathcal{A},n}$ que describe el funcionamiento del algoritmo $\mathcal{A}$ con entradas de largo n

- Usamos el mismo árbol para todas las entradas de largo n
  - El árbol de decisión tiene un registro de las comparaciones hechas por el algoritmo $\mathcal{A}$
- Para cada valor de n se debe tener un árbol que procese entradas de ese tamaño

Demostrando cotas inferiores: Árboles de decisión

Por ejemplo, el siguiente árbol de decisión permite encontrar el mínimo elemento de un arreglo $L[1..4]$:



Demostrando cotas inferiores: Árboles de decisión

Sea $\mathcal{P}$ un problema abstracto

Para **todo algoritmo** $\mathcal{A}$ que resuelve a $\mathcal{P}$ y su correspondiente árbol de decisión $\mathcal{T}_{\mathcal{A},n}$ (para un n dado) hay que considerar:

- Cada posible salida debe tener al menos un nodo externo en $\mathcal{T}_{\mathcal{A},n}$ que la represente
  - De otra manera el algoritmo sería incorrecto
  - La cantidad de nodos externos del árbol debe ser al menos el número de posibles salidas del algoritmo
- La ejecución del algoritmo sobre una entrada en particular corresponde a un camino desde la raíz del árbol hasta el nodo externo correspondiente
- La cantidad de comparaciones realizadas para una entrada dada corresponde al largo del camino seguido en la ejecución
- El tiempo de ejecución de peor caso es la altura del árbol de decisión

Demostrando cotas inferiores: Árboles de decisión

La idea central detrás del modelo de árboles de decisión es la siguiente:

- Determinar el número de nodos externos ℓ que debe tener cualquier árbol que resuelve el problema $\mathcal{P}$ en cuestión
- Un árbol con ℓ nodos externos (posibles salidas) tiene que ser lo suficientemente alto para tener ese número de nodos externos
- En particular, se cumple que para cualquier árbol binario con ℓ nodos externos y altura h,

$$h \geq \lceil \log_2 \ell \rceil$$

- Esto es, el número de nodos externos define una cota inferior respecto a la altura del árbol
- Esto define una cota inferior para el número de comparaciones en peor caso para cualquier algoritmo para el problema en cuestión
- La complejidad de peor caso del problema es $\Omega(\lceil \log_2 \ell \rceil)$

Ordenando un arreglo

Estudiemos ahora el problema de ordenar un arreglo $L[1..n]$ de n elementos

Los algoritmos más eficientes conocidos tienen un tiempo de ejecución de peor caso $O(n \lg n)$

- Mergesort
- Heapsort
- Quicksort (en caso promedio)

¿Existen maneras más rápidas (en el modelo de comparaciones) para ordenar un arreglo?

Vamos a demostrar que no existe tal algoritmo usando árboles de decisión

- Mergesort y Heapsort son óptimos en el peor caso para el modelo de comparaciones

Ordenando un arreglo

Para estudiar mejor el problema de ordenamiento, vamos a redefinirlo de la siguiente manera

Dado un arreglo $L[1..n]$ de n elementos de cierto tipo sobre el que se ha definido una relación de orden total $\leq$ (i.e., los elementos se pueden ordenar)

Queremos obtener una permutación $i_1, i_2, \ldots, i_n$ de $\{1, 2, \ldots, n\}$ tal que:

$$L[i_1] \leq L[i_2] \leq \cdots \leq L[i_n]$$

En la práctica, los elementos del arreglo original deben ser reorganizados de manera que:

$$L[1] \leq L[2] \leq \cdots \leq L[n]$$

lo cual también puede hacerse a partir de nuestra definición

Ordenando un arreglo

Por ejemplo, para el arreglo

1	2	3	4	5	6	7	8	9	10
56	4	58	90	55	30	85	80	50	35

el resultado es la permutación:

$$2, 6, 10, 9, 5, 1, 3, 8, 7, 4,$$

ya que

$$L[2] \leq L[6] \leq L[10] \leq L[9] \leq L[5] \leq L[1] \leq L[3] \leq L[8] \leq L[7] \leq L[4]$$

Ordenando un arreglo

A partir de esta definición del problema queda claro que todo algoritmo de ordenamiento:

- Debe producir una permutación de $\{1, 2, \ldots, n\}$ correspondiente al ordenamiento de la entrada
- Es más, debe ser capaz de producir cualquiera de las posibles permutaciones de $\{1, 2, \ldots, n\}$
- El algoritmo podría ser visto como que es capaz de distinguir la permutación que corresponde a cada entrada
- Hay $n!$ permutaciones distintas de $\{1, 2, \ldots, n\}$

Ordenando un arreglo

A continuación representamos cualquier algoritmo $\mathcal{A}$ de ordenamiento como un árbol de decisión

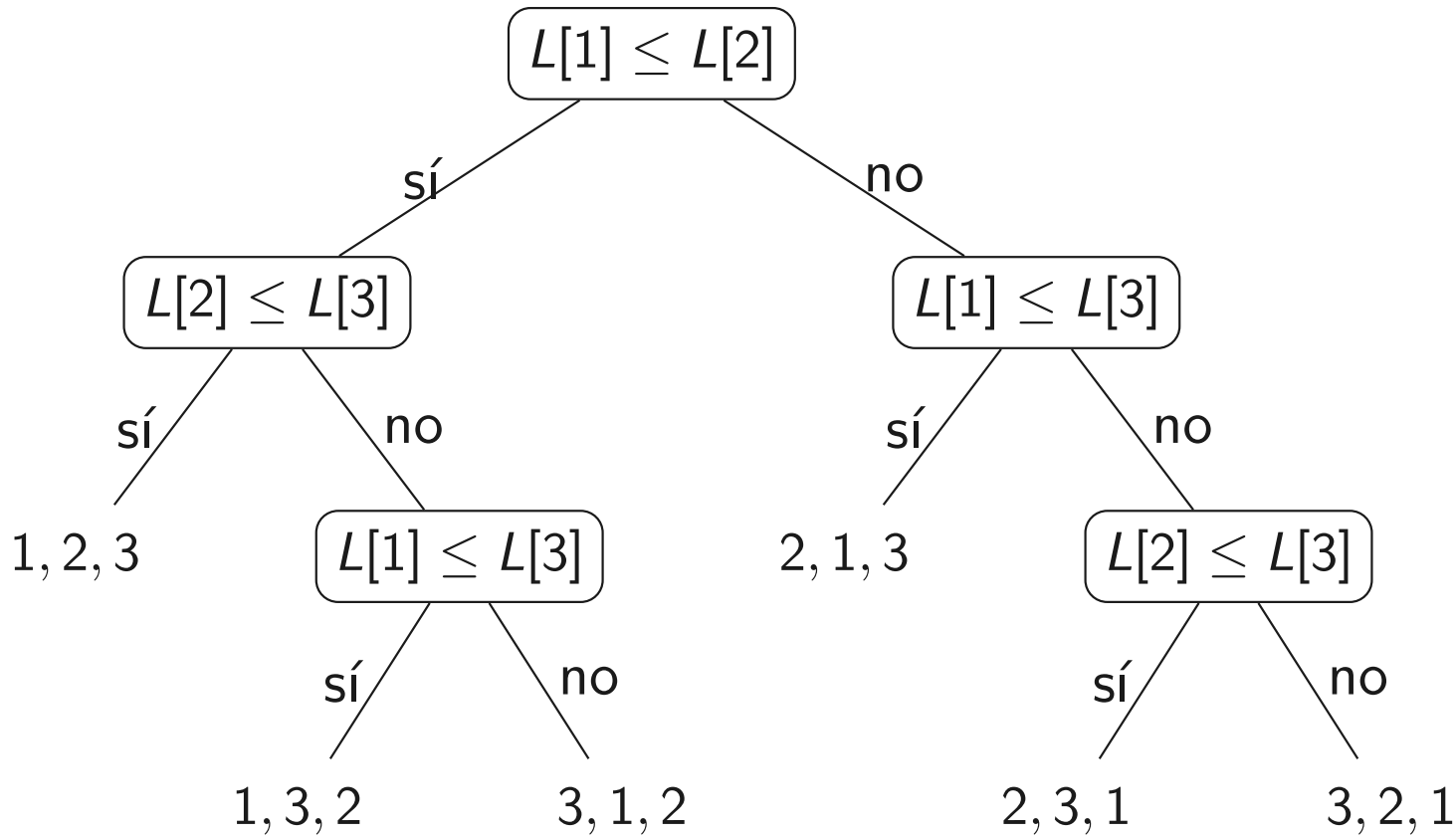
Los nodos internos del árbol son las comparaciones entre elementos del arreglo

Las ramas del árbol corresponden a las decisiones tomadas por el algoritmo al hacer las comparaciones

Los nodos externos del árbol corresponden a las permutaciones de salida

Ordenando un arreglo

Un posible algoritmo para ordenar un arreglo $L[1..3]$ de 3 elementos podría ser:



Ordenando un arreglo

Dado que para n elementos hay $n!$ posibles permutaciones, para todo árbol de decisión $\mathcal{T}_{\mathcal{A},n}$ que represente un algoritmo de ordenamiento se debe cumplir

$$n! \leq \text{hojas}(\mathcal{T}_{\mathcal{A},n})$$

por lo que

$$\lceil \log_2(n!) \rceil \leq \lceil \log_2(\text{hojas}(\mathcal{T}_{\mathcal{A},n})) \rceil$$

Para la altura h de cualquiera de esos árboles también se cumple

$$\lceil \log_2(\text{hojas}(\mathcal{T}_{\mathcal{A},n})) \rceil \leq h,$$

por lo que en total podemos acotar h inferiormente así:

$$\lceil \log_2(n!) \rceil \leq h.$$

Ordenando un arreglo

$\lceil \log_2(n!) \rceil$ es una cota inferior para la altura de todo árbol de decisión que represente un algoritmo de ordenamiento

Ningún algoritmo de ordenamiento puede realizar menos de $\lceil \log_2(n!) \rceil$ comparaciones en el peor caso para ordenar un arreglo

- Una menor altura no le permitiría tener los al menos $n!$ nodos externos necesarios para ordenar correctamente

Encontramos a continuación una forma más intuitiva para $\lceil \log_2(n!) \rceil$

Ordenando un arreglo

La aproximación de Stirling para $n!$ es la siguiente:

$$n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$$

por lo que

$$\begin{aligned}
\log_2(n!) &\approx \left\lceil \log_2\left(\sqrt{2\pi n}\left(\frac{n}{e}\right)^n\right)\right\rceil \\
&\geq \log_2\left(\sqrt{2\pi n}\left(\frac{n}{e}\right)^n\right) \\
&= n\log_2(n) - n\log_2(e) + \frac{\log_2(n)}{2} + \frac{\log_2(2\pi)}{2} \\
&\approx n\lg\left(\frac{n}{e}\right)
\end{aligned}$$

Ordenando un arreglo

En conclusión, todo algoritmo de ordenamiento (en el modelo de comparaciones) debe hacer $\Omega(n \log_2 n)$ comparaciones en el peor caso

Por otro lado, Mergesort y Heapsort hacen $O(n \log_2 n)$ comparaciones en el peor caso

Eso significa:

- Mergesort y Heapsort son algoritmos de ordenamiento óptimos
- La complejidad del problema de ordenamiento es $\Theta(n \log_2 n)$ comparaciones

El problema de mezclar dos arreglos

Sean $A[1..n]$ y $B[1..m]$ dos arreglos ordenados

Se quiere producir un arreglo ordenado $C[1..n+m]$ que contiene los elementos de A y B

- Es la subrutina principal sobre la que se construye Mergesort

Por ejemplo:

1	2	3	4	5
4	55	56	58	90

1	2	3	4	5
30	35	50	80	85

1	2	3	4	5	6	7	8	9	10
4	30	35	50	55	56	58	80	85	90

El problema de mezclar dos arreglos

Un posible algoritmo para este problema es el siguiente

Algoritmo 1: MERGE($A[1..n], B[1..m]$)

```
1  i1 ← 1, i2 ← 1, i ← 1
2  while i1 ≤ n and i2 ≤ m do
3  |   if A[i1] ≤ B[i2] then
4  |   |   C[i] ← A[i1]
5  |   |   i1 ← i1 + 1
6  |   else
7  |   |   C[i] ← B[i2]
8  |   |   i2 ← i2 + 1
9  |   end
10 |   i ← i + 1
11 end
12 while i1 ≤ n do C[i++] ← A[i1++]
13 while i2 ≤ m do C[i++] ← B[i2++]
14 return C
```

El problema de mezclar dos arreglos

En el peor caso, el algoritmo anterior hace $m + n - 1$ comparaciones entre elementos de A y B (línea 3 del algoritmo)

¿Existe una manera más eficiente (en el peor caso) de resolver este problema?

Demostraremos a continuación una cota inferior para el problema que nos permitirá entender mejor esa pregunta

- Usaremos árboles de decisión $\rightarrow$ hay que estudiar cuántas salidas posibles tenemos

El problema de mezclar dos arreglos

Intento 1 (Fallido)

Modificamos brevemente la definición del problema, para poder estudiarlo de mejor manera

Sean $A[1..n]$ y $B[n+1..n+m]$ dos arreglos ordenados

Se quiere producir un arreglo ordenado $C[1..n+m]$ que contiene los elementos de A y B

Por ejemplo:

1	2	3	4	5
4	55	56	58	90

6	7	8	9	10
30	35	50	80	85

1	2	3	4	5	6	7	8	9	10
4	30	35	50	55	56	58	80	85	90

1 6 7 8 2 3 4 9 10 5

El problema de mezclar dos arreglos

Intento 1 (Fallido)

De esta manera, todo algoritmo que resuelve este problema debe producir una permutación de $\{1, 2, \ldots, n + m\}$ como salida

Eso podría significar que hay $(n + m)!$ posibles, por lo que la complejidad de mezclar dos arreglos ordenados sería

$$\Omega(\log_2 (n + m)!) = \Omega((n + m) \log_2 (n + m))$$

Eso significa que todo algoritmo que resuelve el problema debe hacer $\Omega((n + m) \log_2 (n + m))$ comparaciones en el peor caso

Sin embargo, el algoritmo estudiado anteriormente hace $n + m - 1$ comparaciones en el peor caso, lo que indica que algo no está bien

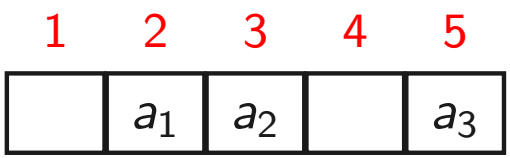
El error viene de que no todas las $(n + m)!$ permutaciones son válidas, ya que el resultado debe estar ordenado (lo cual invalida algunas)

El problema de mezclar dos arreglos

Intento 2 (Válido)

Entendamos ese error a continuación, asumiendo que tenemos arreglos $A[1..3] = \langle a_1, a_2, a_3 \rangle$ y $B[1..2] = \langle b_1, b_2 \rangle$

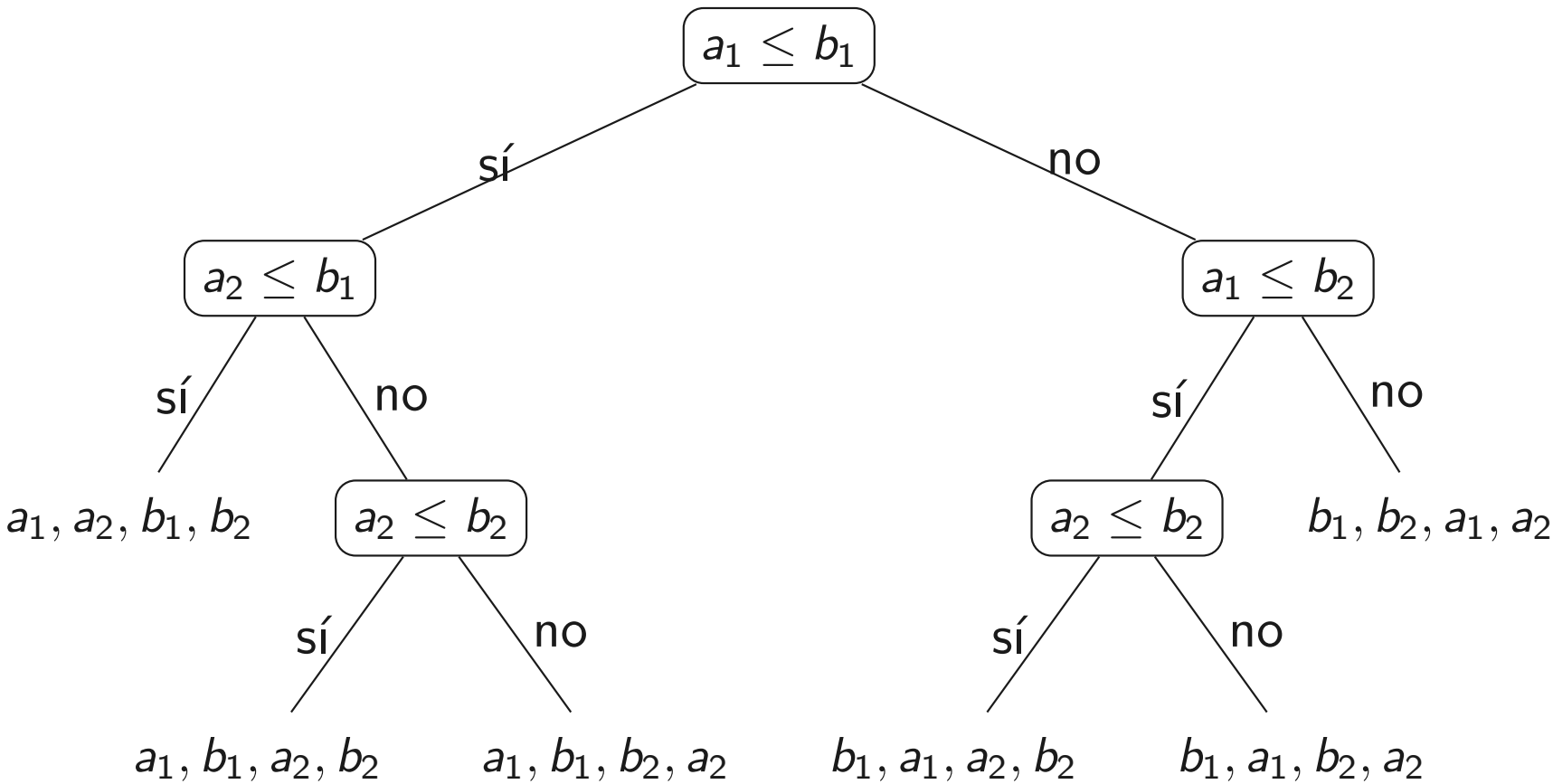
Note que si ubicamos los elementos de A en una cierta ubicación de la permutación de salida, no tenemos elección sobre dónde colocar los elementos de B:



b_1 sólo puede usar la posición 1, mientras que b_2 la 4

El problema de mezclar dos arreglos

El siguiente árbol de decisión permite mezclar los arreglos $A[1..2] = \langle a_1, a_2 \rangle$ y $B[1..2] = \langle b_1, b_2 \rangle$



El problema de mezclar dos arreglos

Intento 2 (Válido)

La cantidad distinta de salidas que debemos producir es igual a la cantidad de formas en que podemos ubicar los elementos de A en la salida.

Hay $\binom{n+m}{n}$ formas distintas de ubicar los elementos de A en la salida

En consecuencia, todo algoritmo que mezcle dos arreglos ordenados debe ser capaz de producir esas $\binom{n+m}{n}$ salidas

Note que

$$\binom{n+m}{n} = \binom{n+m}{m},$$

por lo que el análisis anterior es simétrico con los elementos de B

El problema de mezclar dos arreglos

Intento 2 (Válido)

Todo algoritmo que mezcle dos arreglos ordenados $A[1..n]$ y $B[1..m]$ debe realizar

$$\geq \left\lceil \log_2 \binom{n+m}{n} \right\rceil$$

comparaciones entre elementos de A y B en el peor caso

Pero... ¿ Cuántas comparaciones son $\left\lceil \log_2 \binom{n+m}{n} \right\rceil$?

El problema de mezclar dos arreglos

Intento 2 (Válido)

Analicemos el caso $n = m$, que es más simple:

$$\begin{aligned}\left\lceil \log_2 \binom{2n}{n} \right\rceil &\geq \log_2 \binom{2n}{n} \\ &= \log_2 \left(\frac{(2n)!}{n!n!} \right)\end{aligned}$$

La aproximación de Stirling dice que $n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$, por lo que

$$\begin{aligned}\log_2 \left(\frac{(2n)!}{n!n!} \right) &= \log_2 \left(\frac{\sqrt{4\pi n}\left(\frac{2n}{e}\right)^{2n}}{\left(\sqrt{2\pi n}\left(\frac{n}{e}\right)^n\right)^2} \right) \\ &= \log_2 \left(2^{2n}/\sqrt{\pi n}\right) \\ &= 2n - \frac{1}{2}\log_2 n - \frac{1}{2}\log_2 \pi \\ &= 2n - \frac{1}{2}\log_2 n - 0{,}826.\end{aligned}$$

El problema de mezclar dos arreglos

Intento 2 (Válido)

Conclusión: todo algoritmo que mezcla dos arreglos ordenados de tamaño n (cada uno) debe hacer

$$\geq 2n - \frac{1}{2}\log_2 n - 0{,}826$$

comparaciones

Note que el algoritmo estudiado anteriormente hace $2n-1$ comparaciones en el peor caso, por lo que hay una (pequeña) brecha todavía

El problema de búsqueda en un arreglo ordenado

Sea $A[1..n]$ un arreglo ordenado de elementos de algún universo U. Dado $x \in U$, se quiere determinar si $x \in A$ o no.

Un algoritmo eficiente para resolver este problema es la búsqueda binaria, que necesita $O(\lg n)$ comparaciones en el peor caso.

Ejercicio

1. Use árboles de decisión para demostrar que todo algoritmo que resuelve el problema de búsqueda en un arreglo ordenado de n elementos necesita al menos $\lceil \log_2 n \rceil + 1$ comparaciones en el peor caso.

2. Demuestre que $\lceil \log_2 n \rceil + 1$ comparaciones son suficientes en el peor caso para resolver el problema (lo cual, junto al punto anterior, demostraría la complejidad del problema).