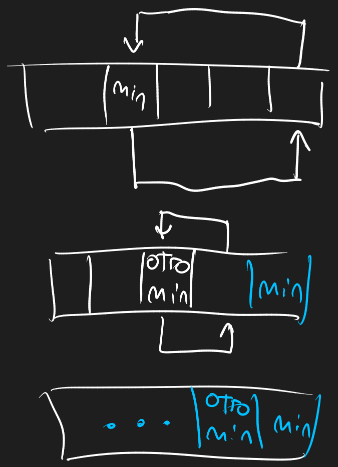
Ordenamiento (Sorting)

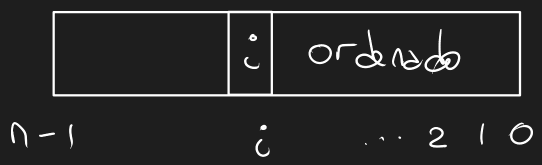
Nov 17

Selection Sort

Busco el mín → lo ordeno



Invariante



Complejidad:

Caso: Peor

$$O\left(\frac{n.(n-1)}{2}\right) \equiv O(n^2)$$

Caso: Mejor

$$O\left(\frac{n.(n-1)}{2}\right) \equiv O(n^2)$$

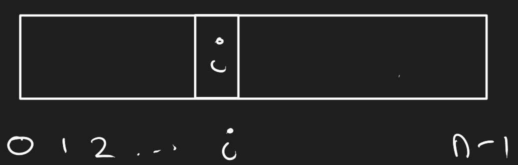
Caso Promedio

$$O(n^2)$$

Insertion Sort

Los veo en orden → los acomodo en su lugar

Invariante



Costo *

$$\sum_{i=1}^{n-1} i-1 = \frac{(n-1)(n-2)}{2}$$

* Pero caso promedio suele ser mejor

Pensar el caso ya ordenado.

Estabilidad.

Dos elementos iguales mantienen el orden original

Inestable

↳ No hay garantía.

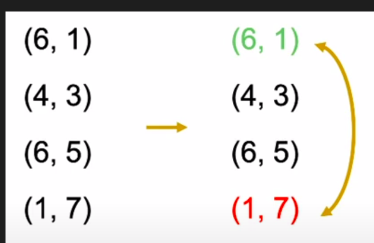
Casos relevantes

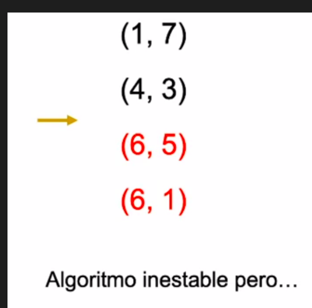
- Pares de naturales
- Pares de string

} Ordenar alguna de las claves no desordena la otra.

Selection Sort

↳ Inestable





Se puede modificar para que sea estable

↳ Estable si me fijo que hay un repetido
y lo pongo "detrás" de él.

Heap Sort

Selection Sort se puede implementar sobre un Heap.

De n^2 a $n \cdot \log n$

Uso max-heap para ordenar de menor a mayor.

↳ desencolo máx (raíz)

↳ lo meto al final

Costo:

$$O(n) + O(n \cdot \log n)$$

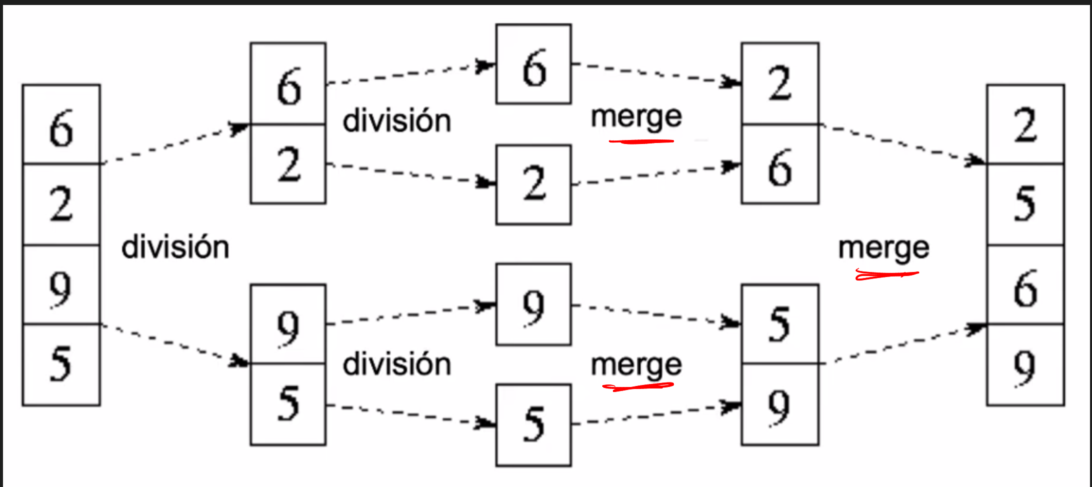
No requiere espacio adicional

Pregunta de examen final:

¿Cómo puedo hacerlo estable?

Merge Sort

- Clásico ejemplo de la metodología "Divide & Conquer" (o "Divide y Reinarás")
- La metodología consiste en
  - dividir un problema en problemas similares....pero más chicos
  - resolver los problemas menores
  - Combinar las soluciones de los problemas menores para obtener la solución del problema original.



- **Análisis**

$$T(n) = 2T\left(\frac{n}{2}\right) + (n-1) =$$

$$= 2^i T\left(\frac{n}{2^i}\right) + i\,n - 2^{i-1} - \dots - 2 - 1 =$$ Hasta $2^i = n$ o sea $i = \log n$

$$= 2^{\log n} T\left(\frac{n}{2^{\log n}}\right) + \log n\, n - 2^{\log n - 1} - 2^{\log n - 2} - \dots - 2 - 1 =$$

$$= 2^{\log n} T(1) + \log n\, n - 2^{\log n - 1} - 2^{\log n - 2} - \dots - 2 - 1 =$$

$$= 2^{\log n} + \log n\, n - 2^{\log n - 1} - 2^{\log n - 2} - \dots - 2 - 1 =$$

$$= O(n \log n)$$

Esto suponiendo que $n = 2^k$, pero si no fuera exactamente así?

Quick Sort

- También D & Q
- Supongo que tengo el mediano

↳ Pero como no lo sé, agarro el del medio como pivote ⊛

Preg: Puedo elegir al azar ?

1° Pasada: Busco el max y lo pongo al final.

$O(n)$ Me sirve para marcar el final del array

(Ahorro 1 pregunta POR CICLO!)

2°: Tomo el pivote (medio) y lo pongo al comienzo.

3°: Recorro con dos punteros comparando contra pivot.

⋮

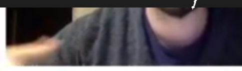
Costo = $O($ Nro de comparaciones $)$

- Peor: $O(n^2)$ (si elijo siempre el menor/mayor)
- Mejor y Promedio $(n \log n)$

↳ si elijo siempre el mediano.

Asume distribución uniforme

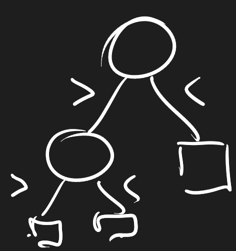
- Si elijo al azar el pivote, no dependo de la distr. uniforme.
- Puedo elegir "S" elementos y usar el mediano como pivote.



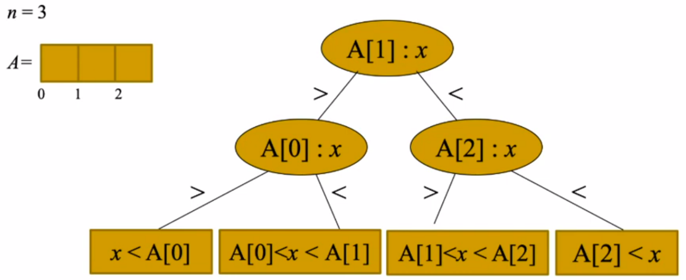
- Merge Sort (y Heap Sort): O(n log n)
- Quick Sort, Selection Sort, Insertion Sort: $O(n^2)$
  - Quick Sort: O(n log n) en el caso mejor
  - Selection Sort: $O(n^2)$ en todos los casos
  - Insertion Sort: O(n) en el caso mejor
- <u>Pregunta</u>: ¿cuál es la eficiencia máxima (complejidad mínima) obtenible en el caso peor? -> Lower bound

Rta: $\Omega(n \cdot \log n)$

Árboles de Decisión



Búsqueda binaria de elemento *x* en un arreglo:

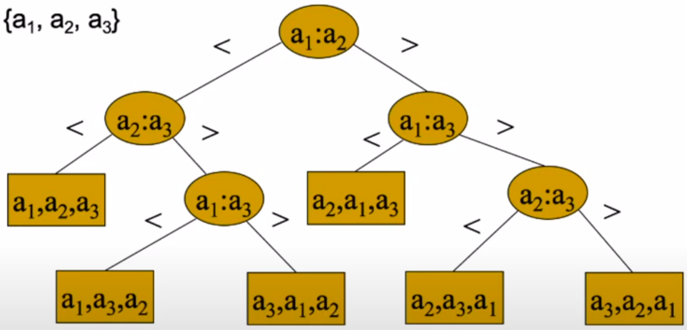


Un árbol de decisión representa las comparaciones ejecutadas por un algoritmo sobre un input dado

Cada hoja corresponde a una de las posibles outputs del algoritmo.

Hojas : n!

árbol de decisión sobre el conjunto $\{a_1, a_2, a_3\}$



- Hay n! posibles permutaciones -> el árbol debe contener n! hojas
- La ejecución de un algoritmo corresponde a un camino en el árbol de decisión correspondiente al input considerado

El camino más largo de la raíz a una hoja (altura) representa el número de comparaciones que el algoritmo tiene que realizar en el caso peor

Teorema: cualquier árbol de decisión que ordena n elementos tiene altura $\Omega(n \log n)$

Demostración:

- Árbol de decisión es binario
- Con n! hojas
- Altura mínima $\rightarrow \Omega(\log(n!)) = \Omega(n \log n)$

Aprox. de Stirling

- Corolario: ningún algoritmo de ordenamiento tiene complejidad mejor que $\Omega(n \log n)$
- Corolario: los algoritmos Merge Sort y Heap Sort tienen complejidad asintótica óptima

- Nota: existen algoritmos de ordenamiento con complejidad más baja, pero requieren ciertas hipótesis extra sobre el input