Ayudantía N° 5

MergeSort y QuickSort

Repaso

Merge

Pseudocódigo

Version not Inplace

Input: secuencias A y B ordenadas

Output: secuencia C ordenada

Memoria adicional: O(n)
Complejidad tiempo: O(n)

Merge(A,B):

1. Nueva secuencia vacia C
2. Sea a y b los primeros elementos de A y B respectivamente
3. Extraemos menor entre a y b de su secuencia
4. Si A y B no vacíos volvemos a 2
5. Concatenar a C la secuencia no vacía

return C

Pseudocódigo
Analogía

Merge

Version Inplace

Notamos que para la versión no in place utilizamos memoria adicional al usar una secuencia nueva para almacenar los elementos de A y B, por ende usamos O(n) en memoria adicional, es decir, |A|+|B| = n.
Por lo cual para la versión in place lo que se hace es usar el mismo espacio reservado a A y B, osea que la memoria adicional es O(1), y luego se a de mover todos los datos mayores al insertado, lo cual claramente genera un costo en el tiempo al tener complejidad de o(n^2).

Merge Sort

Pseudocódigo

Version not Inplace

Input: Secuencia A

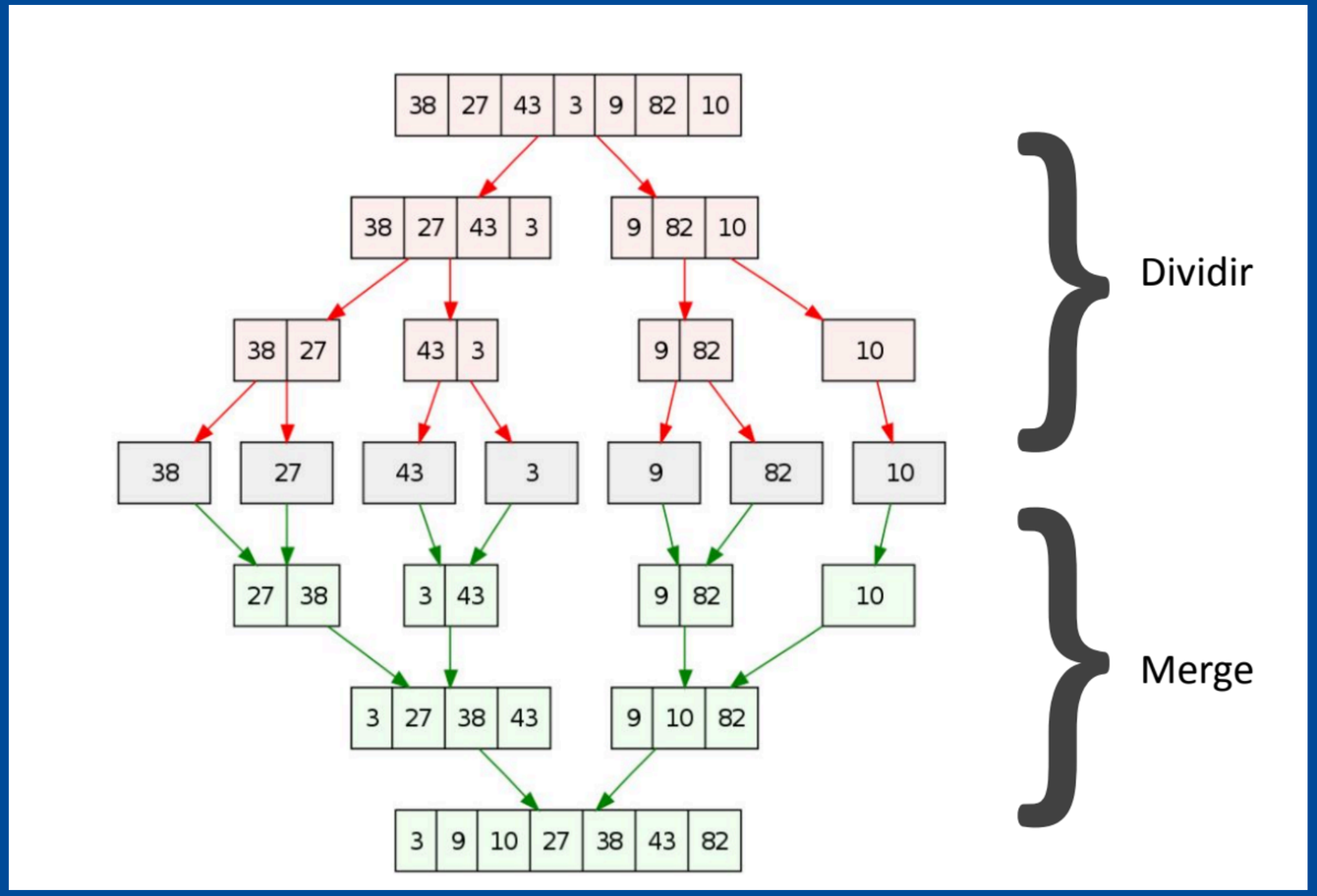
Output: Secuencia ordenada B

Qué estrategia algorítmica ocupa?

```
MergeSort (A):
1    if |A| = 1 : return A
2    Dividir A en A1 y A2
3    B1 ← MergeSort(A1)
4    B2 ← MergeSort(A2)
5    B ← Merge(B1, B2)
6    return B
```

Merge Sort

Debido a la recursión el orden de los pasos no es el siguiente pero la ilustración no quita precisión sobre la idea principal y resultado del algoritmo



Complejidad

Mejor, promedio, peor caso

$O(n\log(n))$

Memoria adicional

$O(n)$

Pseudocódigo

Quick Sort

Inplace

Input: secuencia A y dos enteros i, f

Output: Nada

```
QuickSort (A, i, f):
1     if i ≤ f :
2         p ← Partition(A, i, f)
3         Quicksort(A, i, p − 1)
4         Quicksort(A, p + 1, f)
```

Pseudocódigo

Quick Sort

- Elige el pivote
- Coloca al pivote en su posición correcta en el arreglo ordenado
- Pone los elementos menores que él a su izquierda y los mayores a su derecha

```
   Partition (A, i, f):
1      x ← índice aleatorio en {i, ..., f}
2      p ← A[x]
3      A[x] ⇄ A[f]
4      j ← i
5      for k = i ... f − 1 :
6          if A[k] < p :
7              A[j] ⇄ A[k]
8              j ← j + 1
9      A[j] ⇄ A[f]
10     return j
```

Quick Sort

```
QuickSort (A, i, f):
1     if i ≤ f :
2         p ← Partition(A, i, f)
3         Quicksort(A, i, p − 1)
4         Quicksort(A, p + 1, f)
```

```
Partition (A, i, f):
1     x ← índice aleatorio en {i, . . . , f}
2     p ← A[x]
3     A[x] ⇄ A[f]
4     j ← i
5     for k = i . . . f − 1 :
6         if A[k] < p :
7             A[j] ⇄ A[k]
8             j ← j + 1
9     A[j] ⇄ A[f]
10    return j
```

Complejidad

Mejor y caso promedio

O(nlog(n))

Peor caso

O(n^2)

Memoria adicional

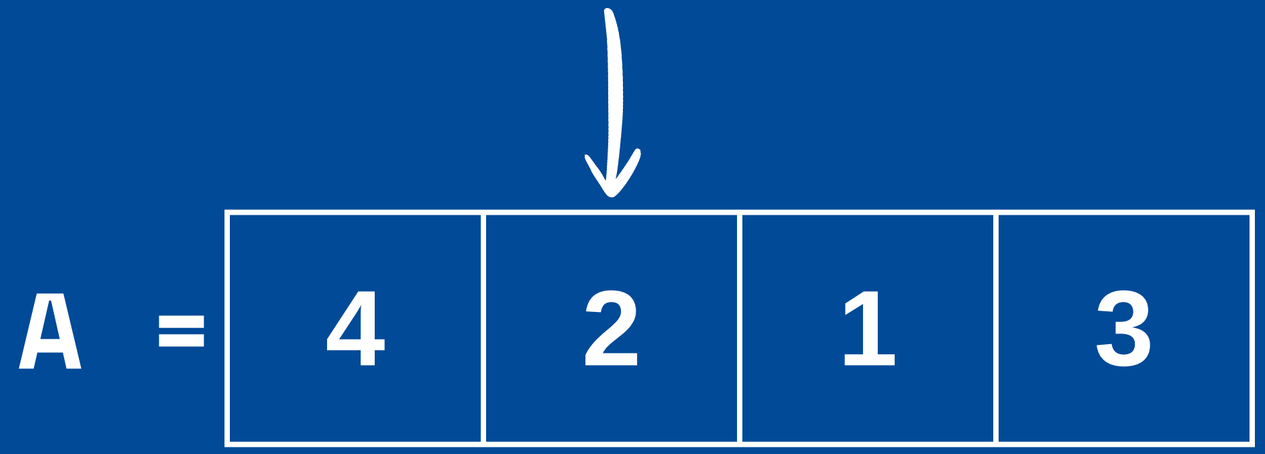
O(1)

quick sort




```
QuickSort(A, 0, 3)
```

quick sort

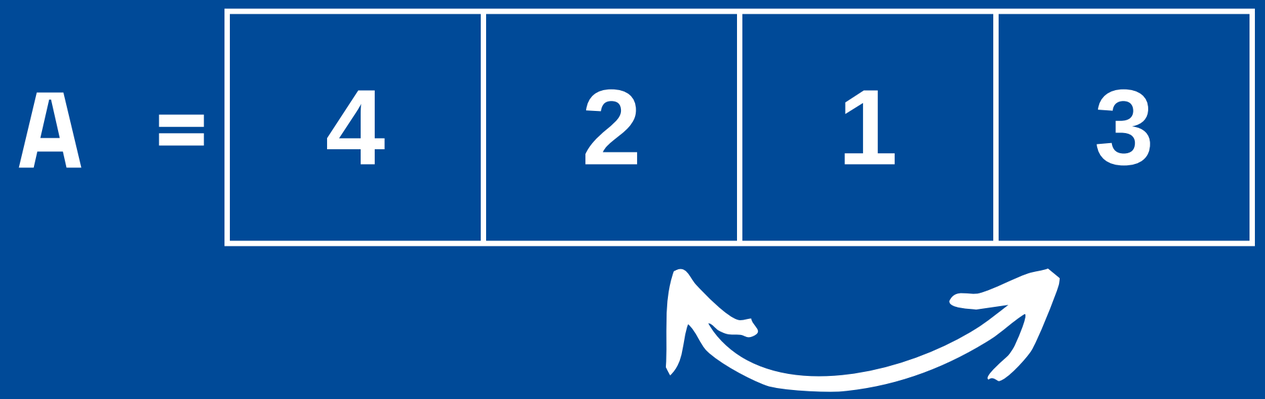


Elegimos un pivote aleatorio

x = 1

p = A[1] = 2

quick sort



Movemos el pivote al final

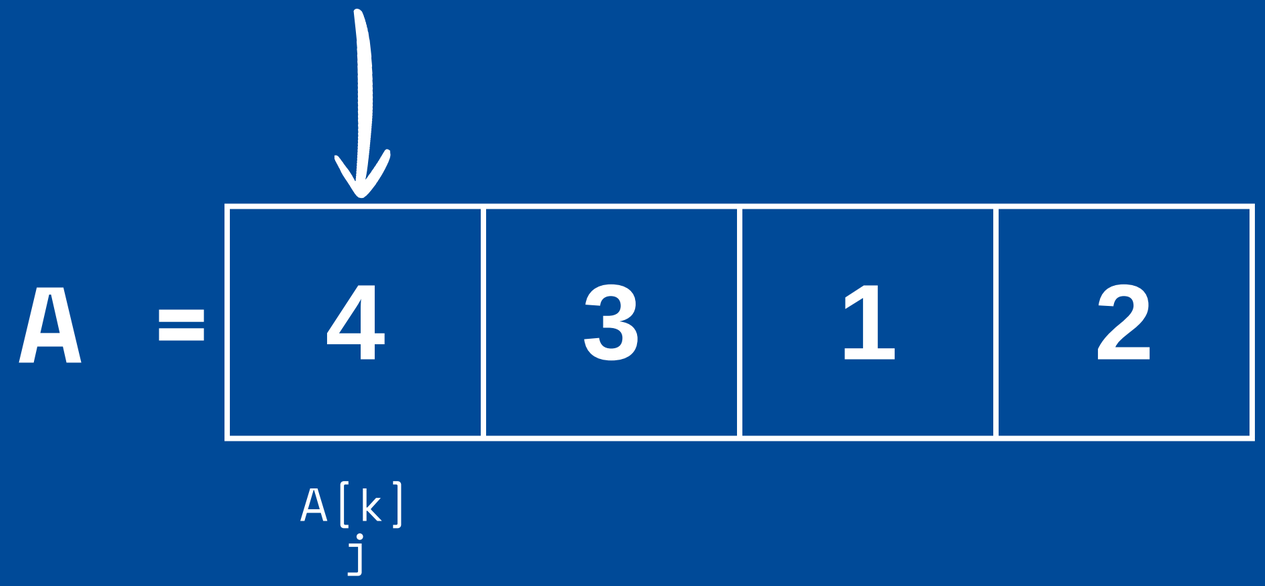
quick sort



j = i = 0

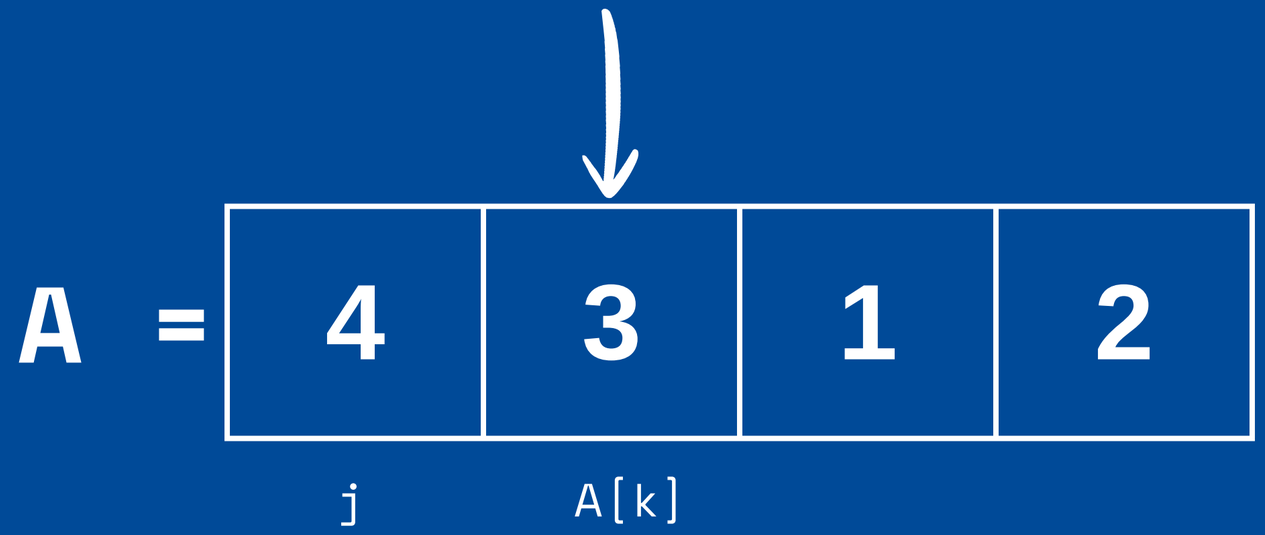
Vamos a recorrer con k

quick sort



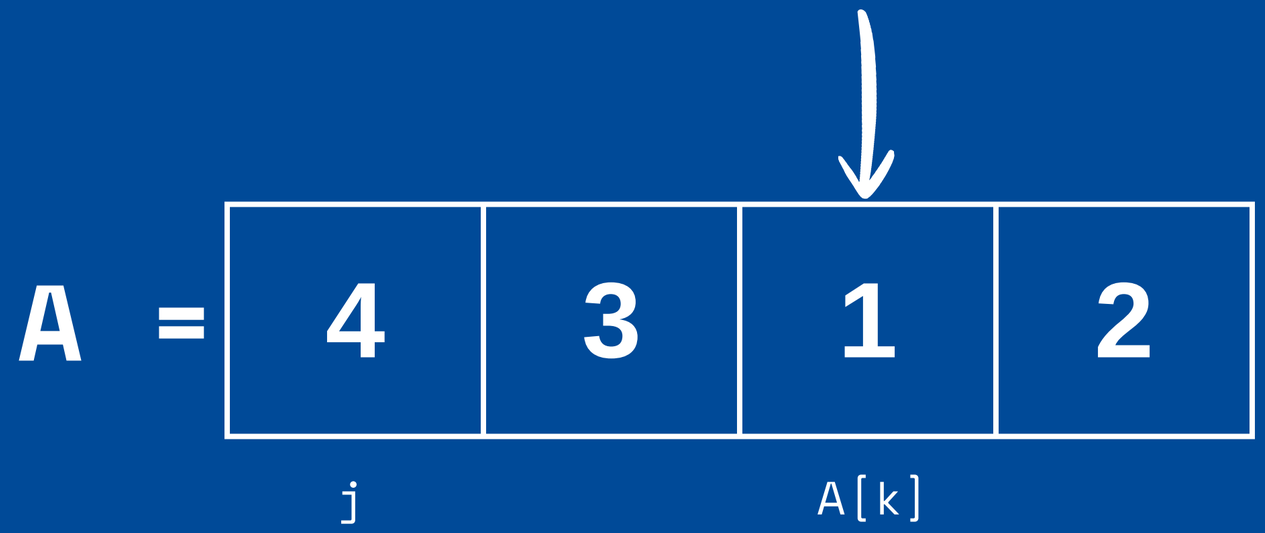
¿4 < 2? No, no hacemos nada

quick sort



¿3 < 2? No, no hacemos nada

quick sort



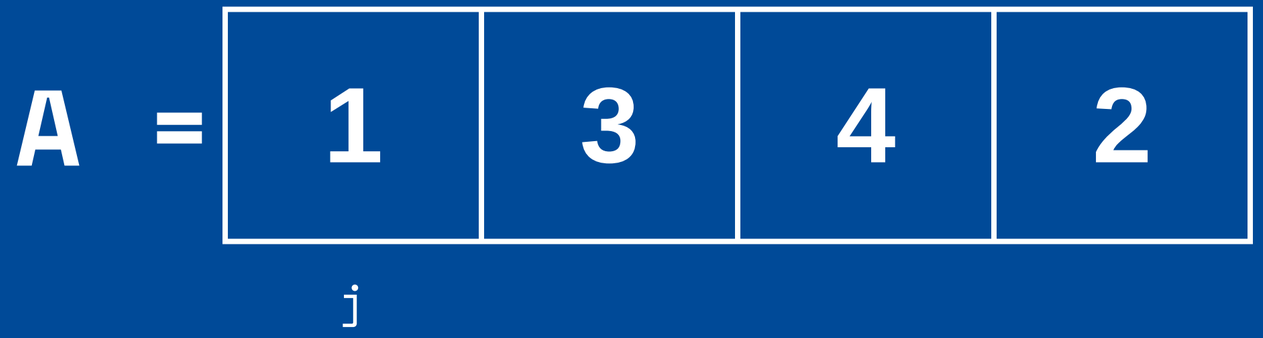
¿1 < 2? Si

quick sort

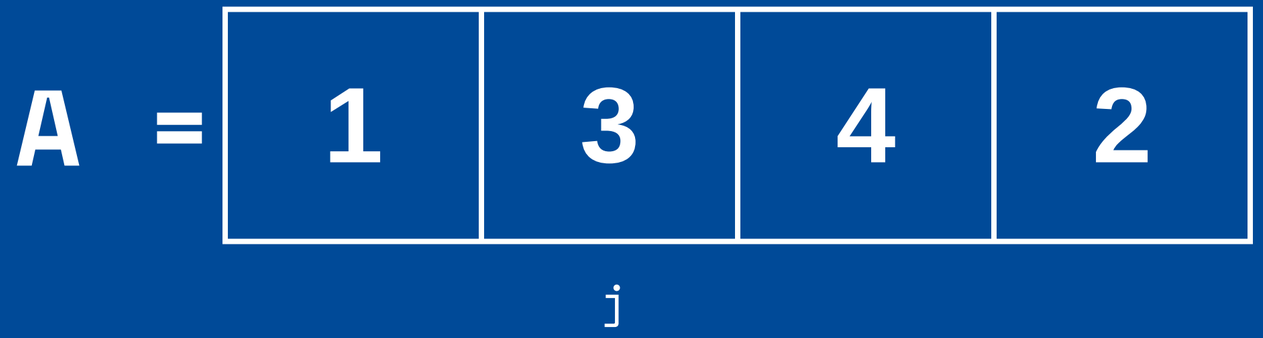


$$A[j] \rightleftarrows A[k] = A[0] \rightleftarrows A[2]$$

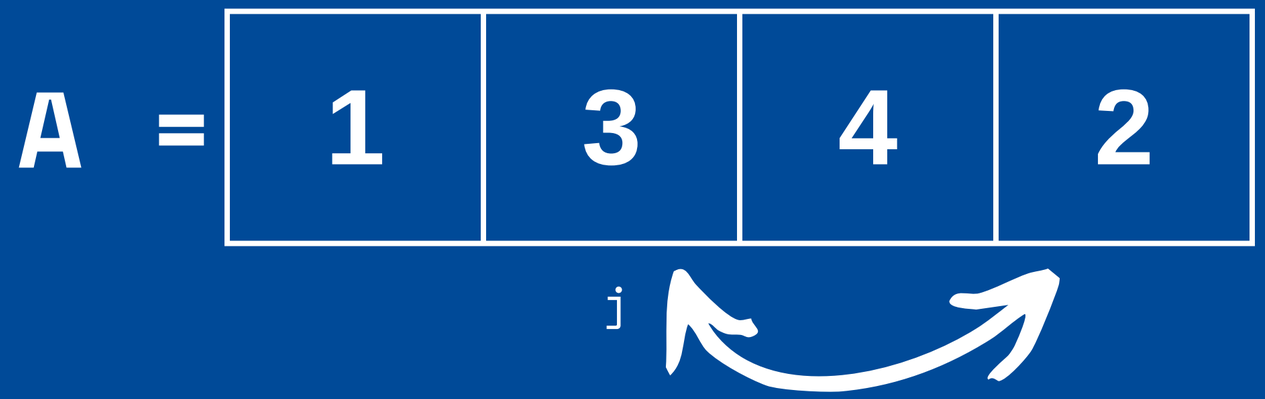
quick sort



quick sort



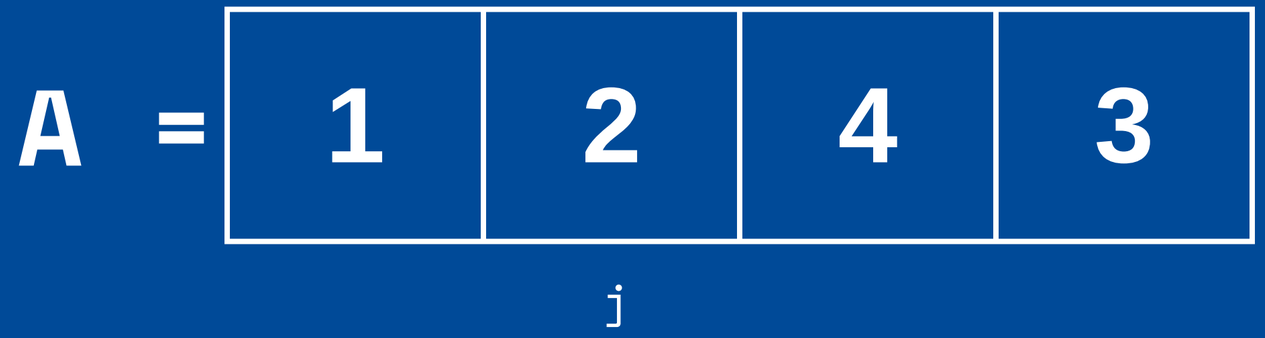
quick sort



LLevamos el pivote a su lugar

A[j] ⇄ A[f] = A[1] ⇄ A[3]

quick sort



2 quedó en su posición final

quick sort



Siguientes llamados:

QuickSort(A, 0, 0) → termina
QuickSort(A, 2, 3) → seguimos

quick sort

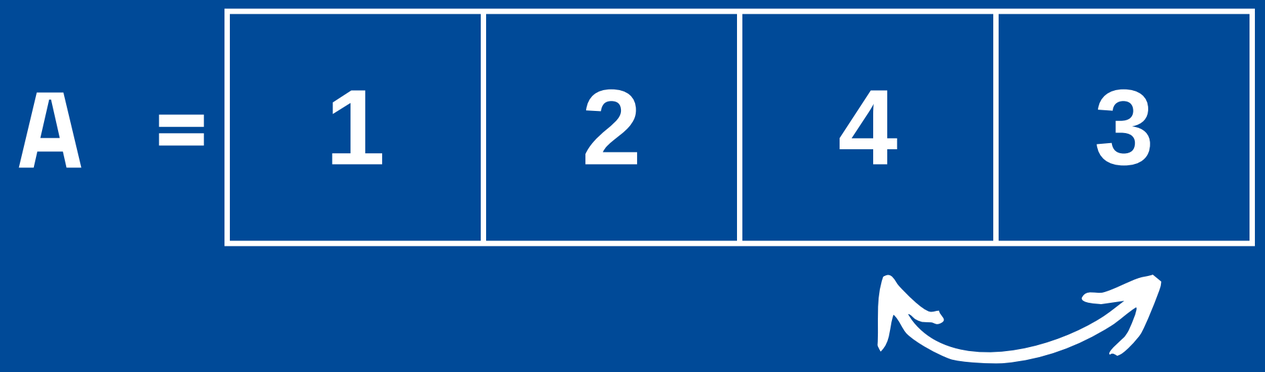


Elegimos un pivote aleatorio

x = 2

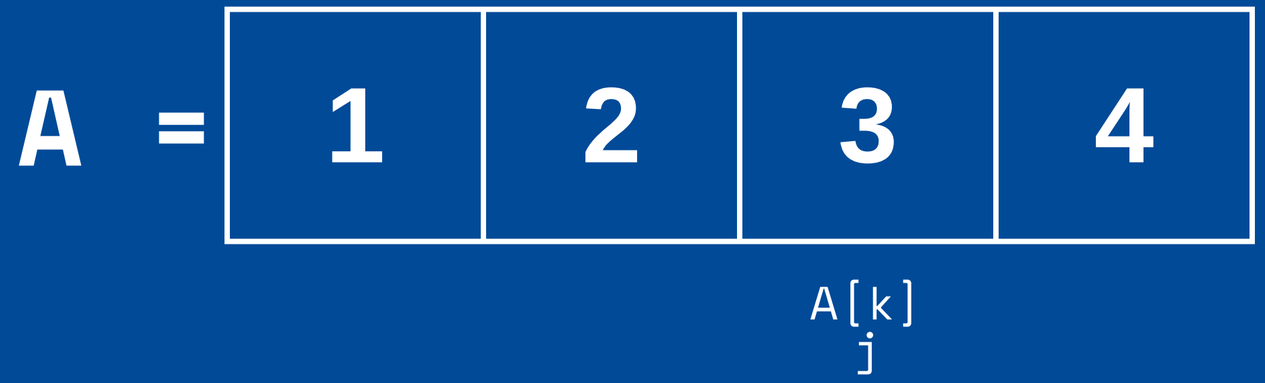
p = A[2] = 4

quick sort



Movemos el pivote al final

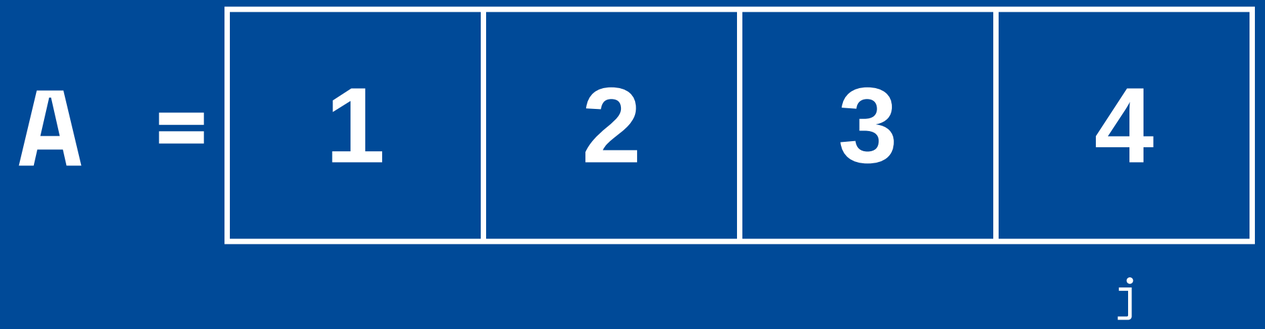
quick sort



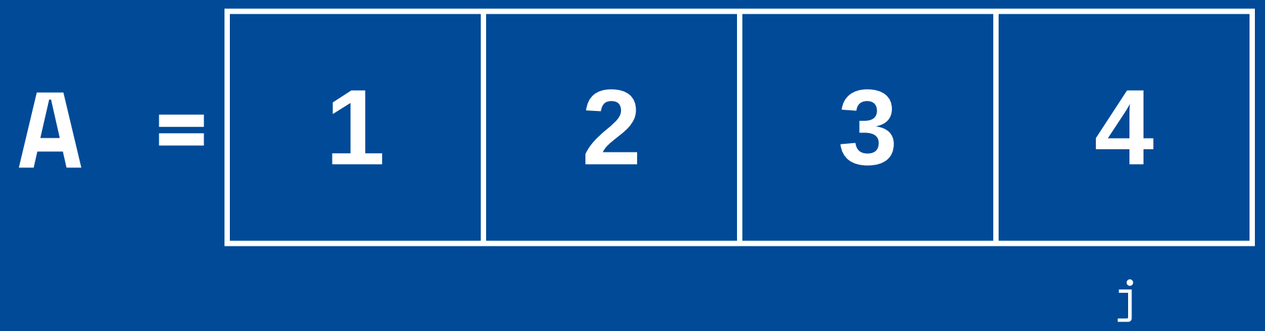
¿3 < 4? Si

Swap consigo mismo (porque k = j)

quick sort



quick sort



Llevamos el pivote a su lugar

A[j] ⇄ A[f] = A[3] ⇄ A[3]

(Ya está en su lugar)

Ejercicio 1

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.



Pregunta 1 - I1-2020-1

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?

Pregunta 1 - I1-2020-1

Pregunta 1 - Solución

¿Qué es T(n)?

T(n) es el tiempo (o costo) que toma ejecutar un algoritmo con input de tamaño n.

Si el algoritmo es recursivo:

- se divide en subproblemas
- T(n) depende de esos subproblemas

En este caso, el algoritmo divide el arreglo en tres subarreglos, por lo que T(n) depende de las tres llamadas recursivas más el costo de combinarlos (merge).

Pregunta 1 - Solución

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:

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

Alternativamente:

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

Pregunta 1: Resolviendo recurrencia

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:

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

- Trabajar directamente con ⌈ ⌉ y ⌋ ⌊ es algebraicamente incómodo, por lo que acotamos el problema usando potencias de 3.
- Expandiremos la recurrencia usando potencias de 3 y al final volveremos a expresar el resultado en términos de n.

Pregunta 1: Resolviendo recurrencia

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:

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

Pregunta 1: Resolviendo recurrencia

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:

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

i es un contador de de pasos de la expansión de la recursión

llegamos al caso base cuando i = k

Pregunta 1: Resolviendo recurrencia

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

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

Ejercicio 2

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

Pregunta 2 - I1-2020-2

Pregunta 2a - Solucion

- Intercambios de líneas 3 y 9 se hacen siempre
- Dentro del loop, sólo se hacen cuando A[k]< p
- Si queremos hacer la mayor cantidad de intercambios posibles, queremos que siempre A[k]< p.
- Entonces si tenemos un subarreglo de tamaño m, se hacen 2 + m-1 = m+1 intercambios
- Quedan subarreglos de largos 0 y m-1

```
Partition (A, i, f):
1     x ← índice aleatorio en {i, ..., f}
2     p ← A[x]
3     A[x] ⇄ A[f]
4     j ← i
5     for k = i ... f − 1 :
6         if A[k] < p :
7             A[j] ⇄ A[k]
8             j ← j + 1
9     A[j] ⇄ A[f]
10    return j
```

Pregunta 2a - Solucion

Con un arreglo de tamaño n tenemos:

$$
\begin{aligned}
T(n) &= n + (n-1) + (n-2) + \ldots + 1 \\
&= \frac{n \times (n+1)}{2} = \frac{1}{2}\left(n^2 + n\right) \\
&\in O\left(n^2\right)
\end{aligned}
$$

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

Pregunta 2 - I1-2020-2

Pregunta 2b - Solucion

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 $p > q$.

Pregunta 2b - Solucion

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.

Feedback