Ayudantía N° 2

mergeSort & quickSort

Repaso

Pseudocódigo

mergeSort

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

mergeSort

Pseudocódigo
Analogía

Version Inplace

Notamos que para la <u>versión no in place</u> 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 <u>versión in place</u> 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).

Utilicemos Merge

Pseudocódigo

mergeSort

Version not Inplace

Input: Secuencia A

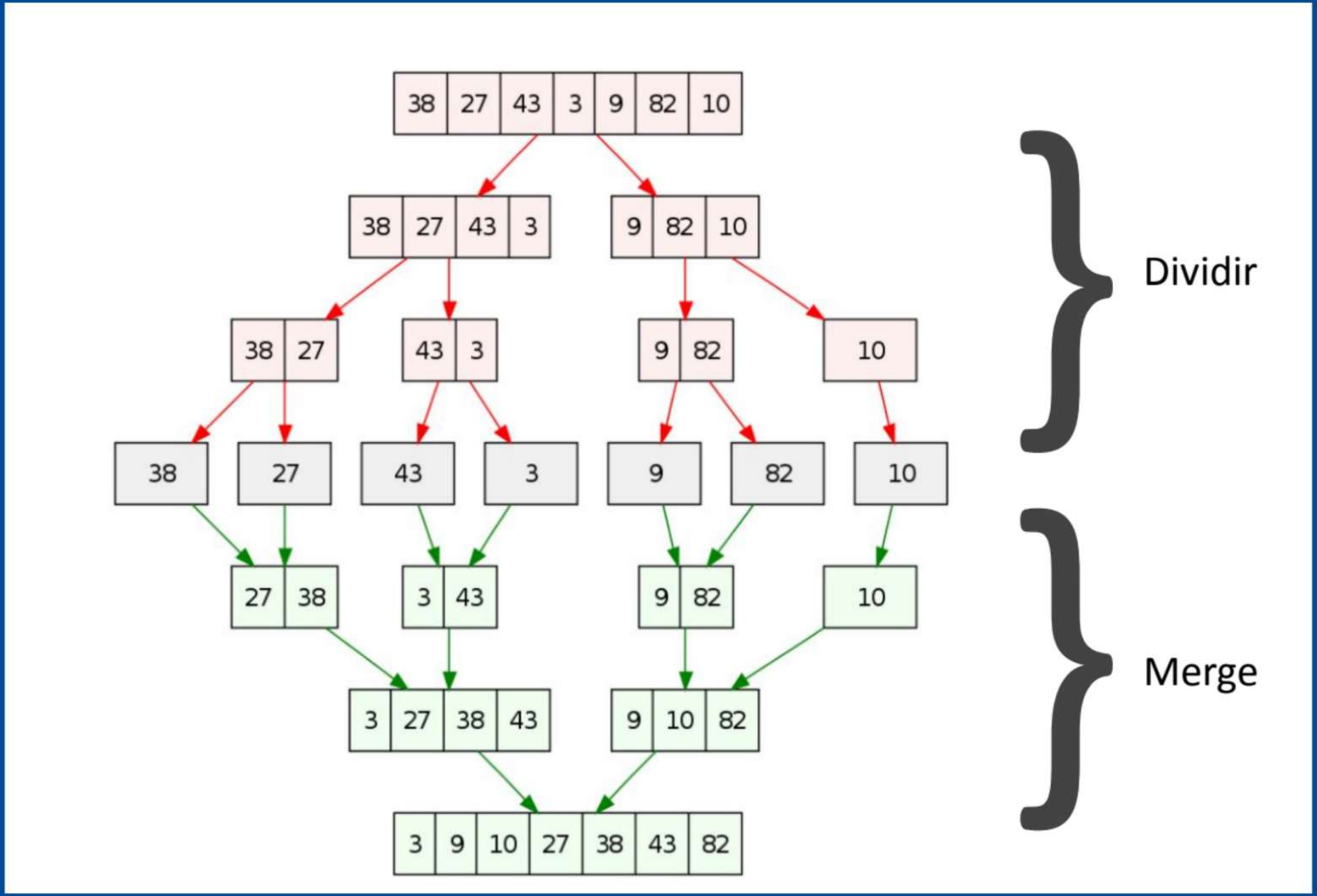
Output: Secuencia ordenada B

Qué estrategia algorítmica ocupa?

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

mergeSort

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(nlog(n))

Memoria adicional

O(n)

Veamos la visualización !

https://visualgo.net/en/sorting

mergeSort

EJERCICIO 1

A pesar que **MergeSort** es **O(n*log(n))**, e **InsertionSort** es **O(n^2)**, en la práctica **InsertionSort** funciona mejor para problemas pequeños.

Sea **n** la cantidad de elementos en una secuencia por ordenar, y **k** un valor a determinar con **k ≤ n** . Considera una modificación de **MergeSort** llamada **MergeInserSort** en la que **n/k** sublistas de largo **k** son ordenadas con **InsertionSort** y luego unidas usando **Merge**.

Pregunta 1 - I1-2021-1

a) Muestra que con **InsertionSort** se pueden ordenar **n/k** sublistas, cada una de largo **k**, obteniendo **n/k** sublistas ordenadas, en tiempo **O(nk)** en el peor caso.

Pregunta 1 - I1-2021-1

- Sabemos que InsertionSort toma tiempo O(n^2) en arreglos de largo n
- Luego en un arreglo de largo k, toma tiempo O(k^2)
- Como tenemos n/k sub listas, correr todos los InsertionSort nos tomaría tiempo

$$O(k^2 \frac{n}{k}) = O(nk)$$

Pregunta 1 - I1-2021-1

b) Muestra cómo se pueden mezclar las **sublistas ordenadas**, obteniendo finalmente una sola lista ordenada, en tiempo **$O(n \log(n/k))$** en el peor caso

Pregunta 1 - I1-2021-1

- Podemos juntar las sublistas de a pares, y correr el algoritmo **Merge** conocido, que corre en tiempo **O(2k)** con **k** el largo de cada lista

- Si las juntamos de a pares, vamos a tener que correr **Merge** una cantidad **n/2k** de listas, por lo que la complejidad queda en **O(n)**.

- Ahora, repetimos el proceso, que va a tener nuevamente complejidad **O(n)**

- Cuántas veces se repite el proceso? Se repite **log_2 (n/k)** veces

$$O(nlog(\frac{n}{k}))$$

Pregunta 1 - I1-2021-1

c) Dado que MergeInserSort corre en tiempo O(nk + n log(n/k)) en el peor caso, ¿cuál es el valor máximo de k, en función de n (en notación O) para el cual MergeInserSort corre en el mismo tiempo que MergeSort normal?

Hint: log(log(n)) es despreciable, relativo a log(n), para n suficientemente grande

Pregunta 1 - I1-2021-1

Vemos que si tomamos un **k** en **O(1)**, entonces cumplimos con lo pedido:

$$O(nk + nlog(n/k)) = O(nlog(n))$$

Aprovechando el Hint, podemos probar con un k en O(log(n))

$$\begin{aligned} O(nk + nlog(n/k)) &= O(nk + nlog(n) - nlog(k)) \\ &= O(nlog(n) + nlog(n) - nlog(log(n))) \\ &= O(2nlog(n) - nlog(log(n))) \\ &= O(nlog(n)) \end{aligned}$$

Pregunta 1 - I1-2021-1

mergeSort

EJERCICIO 2

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 2

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

Pregunta 2

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 2

Recordemos el Teorema Maestro

Master Theorem

Theorem (master theorem, simple form):

For positive constants a, b, c, and d, and $n = b^k$ for some integer k, consider the recurrence

$$r(n) = \begin{cases} a, & \text{if } n = 1 \\ cn + d \cdot r(n/b), & \text{if } n \geq 2 \end{cases}$$

then

$$r(n) = \begin{cases} \Theta(n), & \text{if } d < b \\ \Theta(n \log n), & \text{if } d = b \\ \Theta(n^{\log_b d}) & \text{if } d > b. \end{cases}$$

Pregunta 2 - a: Usando el Teorema Maestro

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:

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

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:

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

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),

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

Pregunta 2 - a: 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:

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

Pregunta 2 - a: 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

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

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

Finalmente, ¿Qué sucede con la complejidad del algoritmo cuando $\boldsymbol{k}$ tiende a $\boldsymbol{n}$?

Pregunta 2

Pregunta 2 - b: Primera Solución

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

$$T(n,k) \leq \underbrace{log_2(k) \cdot n}_{\text{Costo de realizar merge para k arreglos ordenados}} + \overbrace{T\left(\left\lceil\frac{n}{k}\right\rceil, k\right) + T\left(\left\lceil\frac{n}{k}\right\rceil, k\right) + ... + T\left(\left\lceil\frac{n}{k}\right\rceil, k\right)}^{\text{Se divide el arreglo en k arreglos de al menos } \lceil\frac{n}{k}\rceil \text{ elementos.}}$$

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

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

Pregunta 2 - b: Primera Solución

Quedando finalmente

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

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

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

Pregunta 2 - b: 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).

Repaso

Pseudocódigo

QuickSort

Input: secuencia A y dos enteros i, f

Output: Nada

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

El verdadero héroe:

- 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):

$x \leftarrow$ índice aleatorio en $\{i, \ldots, f\}$
$p \leftarrow A[x]$
$A[x] \rightleftarrows A[f]$
$j \leftarrow i$
for $k = i \ldots f - 1$:
    if $A[k] < p$:
        $A[j] \rightleftarrows A[k]$
        $j \leftarrow j + 1$
$A[j] \rightleftarrows A[f]$
return j

Resumen QuickSort

- Elegir pivote
- Mover elementos a cada lado del pivote, a un lado los mayores y al otro los menores
- La lista va a quedar separada en dos sublistas, una con los elementos a la izq. del pivote y otra con los de la derecha
- Repetir recursivamente para cada sublista mientras estas contengan más de un elemento.
- Lista ordenada!

Complejidad

Mejor Caso	O(n * log(n))
Caso Promedio	O(n * log(n))
Peor Caso	O(n^2)

¿Cuándo ocurre cada uno de ellos?

En el anexo podrán ver una demostración de las complejidades de los casos peor, mejor y promedio

Veamos la visualización !

https://visualgo.net/en/sorting

quickSort
EJERCICIO 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

Pregunta 1 a) - I1-2020-2

Solución 1 a) - I1-2020-2

- 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. $x \leftarrow$ índice aleatorio en $\{i, \ldots, f\}$
2. $p \leftarrow A[x]$
3. $A[x] \rightleftarrows A[f]$
4. $j \leftarrow i$
5. **for** $k = i \ldots f-1$:
6.   **if** $A[k] < p$:
7. $A[j] \rightleftarrows A[k]$
8. $j \leftarrow j+1$
9. $A[j] \rightleftarrows A[f]$
10. **return** j

Solución 1 a) - I1-2020-2

Con un arreglo de tamaño n tenemos:

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

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 1 b) - I1-2020-2

Solución P1 b) - I1-2020-2

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 $p > q$.

Solución P1 b) - I1-2020-2

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.

quickSort

EJERCICIO 2

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.

Pregunta 1 - C1-2019-1

Solución 1 - C1-2019-1 - Partition3

Algorithm 1 Partition3(A, i, f)

$p \leftarrow$ elemento aleatorio en $A[i, f]$
$m, M, P \leftarrow$ secuencias vacías
for x in $A[i, f]$ **do**
    if $x < p$ **then**
        Insertar x en m
    else if $x = p$ **then**
        Insertar x en P
    else if $x > p$ **then**
        Insertar x en M
    end if
end for
$A[i, f] \leftarrow$ Concat$(m,\ p,\ P,\ M)$
return $i + |m|,\ i + |m| + |P|$

Solución 1 - C1-2019-1 (versión in-place)

Algorithm 2 Partition3(A, i, f)

```
x ← índice aleatorio en {i, . . . , f}
p ← A[x]
swap(A[x], A[f])
j ← i
l ← i
for k in {i, . . . , f − 1} do
    if A[k] < p then
        swap(A[k], A[j])
        swap(A[j], A[l])
        j ← j + 1
        l ← l + 1
    end if
    if A[k] = p then
        swap(A[k], A[j])
        j ← j + 1
    end if
end for
swap(A[j], A[f])
return l, j
```

Solución 1 - C1-2019-1 - Llamada a Quicksort

Algorithm 3 Quicksort3(A, i, f)

if $i \leq f$ **then**
    $p_1, p_2 =$ Partition3(A, i, f)
    Quicksort3$(A, i, p_1 - 1)$
    Quicksort3$(A, p_2 + 1, f)$
end if

ANEXO - ANÁLISIS DE:

- Mejor Caso
- Peor Caso
- Caso promedio

Mejor Caso

- Ocurre cuando los pivotes elegidos dividen los arreglos y sub-arreglos en 2 mitades del mismo tamaño.

  T(1) = 1

  T(n) = 2 T(n/2) + n

- Es la misma ecuación que MergeSort, por ende la misma demostración

Peor Caso

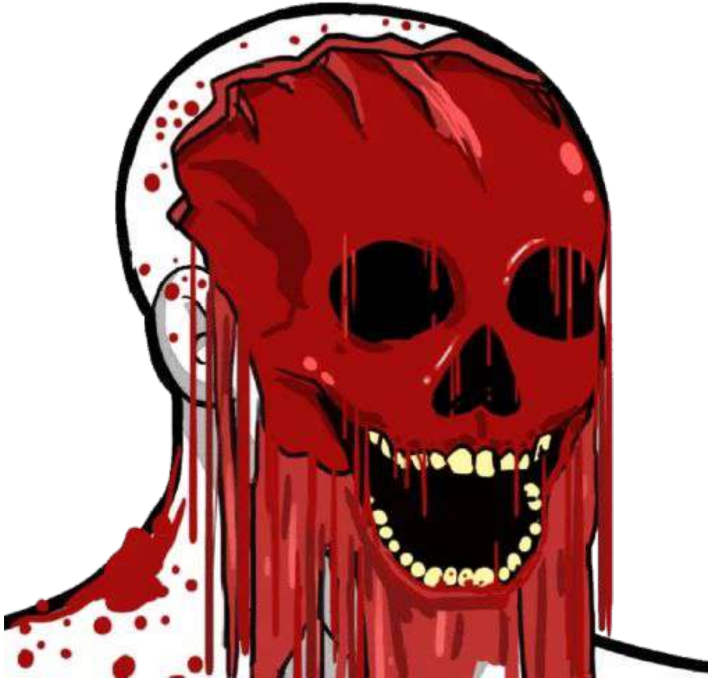
- Ocurre cuando los pivotes elegidos corresponden a los mínimos o máximos de los arreglos y sub-arreglos.

  T(1) = 1

  T(n) = T(n-1) + n

- Vemos que los llamados solo van eliminando de a 1 elemento con pasadas lineales.

Caso Promedio



Caso Promedio

- Consideremos 2 elementos cualquiera i y j dentro de nuestro arreglo. Definamos Y_{ij} como la cantidad de veces que se comparan estos 2 elementos.
- Si pensamos en todos los posibles pares, tenemos:

$$\sum_{i=1}^{n-1} \sum_{i+1}^{n} Y_{ij}$$

- ¿A qué se deben esos subíndices?
- Nos interesa saber la esperanza de esta sumatoria. ¿Por qué?

$$\sum_{i=1}^{n-1} \sum_{i+1}^{n} E(Y_{ij})$$

- Tengamos en cuenta a p, nuestro pivote tras 1 aplicación del algoritmo de partición.

- Si **(i < p < j)** → el pivote separó para siempre i y j, por ende $Y_{ij} = 0$

- Si **(p < i < j)** o **(i < j < p)** → el pivote dejo a i y j en la misma partición, con posibilidad de que se comparen a futuro.

- Si **(p = i < j)** o **(i < j = p)** → los elementos fueron comparados, por ende no vuelven a ser comparados. De esta forma $Y_{ij} = 1$

- Ya que Y_{ij} solo toma los valores 0 y 1:

$$E(Y_{ij}) = 0 \cdot Pr(Y_{ij} = 0) + 1 \cdot Pr(Y_{ij} = 1)$$

$$\mathbf{= Pr(Y_{ij} = 1)}$$

- Para calcular $Pr(Y_{ij} = 1)$ pensemos en el siguiente conjunto

$$\mathbf{\{i, i+1, i+2, \cdot \cdot \cdot, j-2, j-1, j\}}$$

- Este arreglo posee j-i+1 elementos, si asumimos una distribución uniforme sobre la elección del pivote, cada elemento tiene una posibilidad de ser el pivote de 1 / (j-i+1). Recordemos cuando $Y_{ij} = 1$.

- Esto nos deja un valor muy claro, ya que solo hay 2 casos, cuando i es el pivote o cuando j es el pivote, así:

$$\mathbf{Pr(Y_{ij} = 1) = 2 / (j-i+1)}$$

(casi) Finalmente vemos que

$$\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \mathrm{E}(Y_{i,j})$$

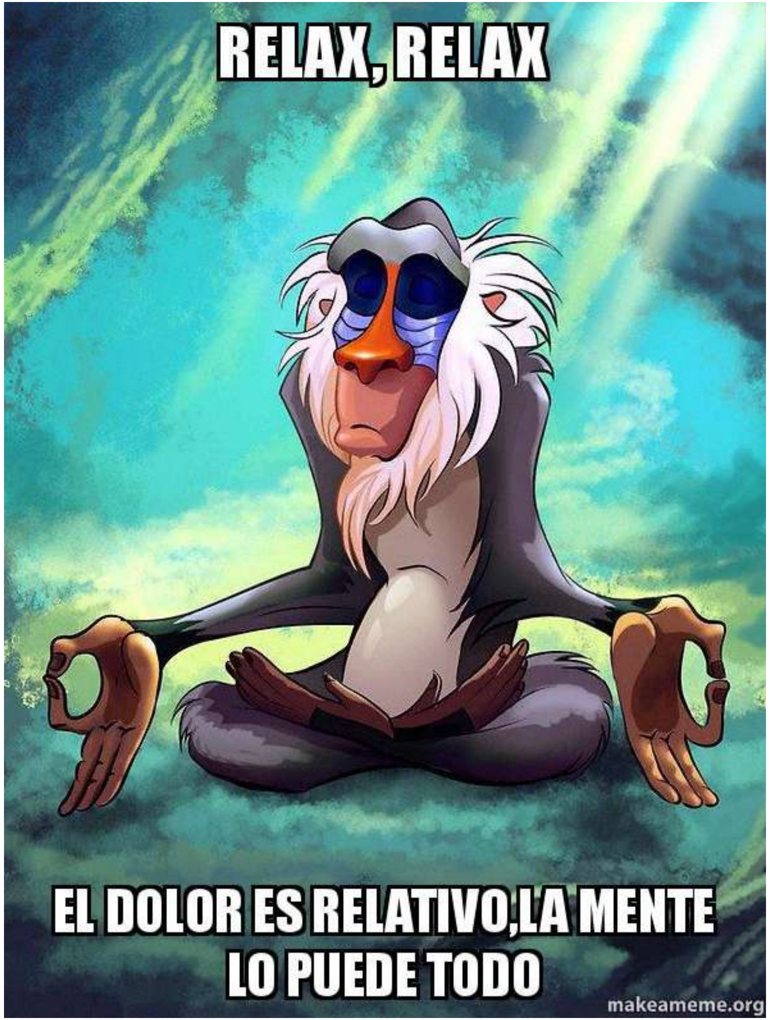
$$\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1}$$

$$\sum_{i=1}^{n-1} \sum_{k=2}^{n-i+1} \frac{2}{k}$$

$$\sum_{k=2}^{n} (n+1-k) \cdot \frac{2}{k}$$

$$2 \cdot (n+1) \cdot \left(\sum_{k=2}^{n} \frac{1}{k} \right) - 2 \cdot (n-1)$$

$$2 \cdot (n+1) \cdot \left(\sum_{k=1}^{n} \frac{1}{k} \right) - 4 \cdot n$$


RELAX, RELAX
EL DOLOR ES RELATIVO,LA MENTE
LO PUEDE TODO
makeameme.org

$$2 \cdot (n+1) \cdot \left(\sum_{k=1}^{n} \frac{1}{k} \right) - 4 \cdot n$$

Lineal

Lineal

¡Es la suma armónica!

$$\sum_{k=1}^{n} \frac{1}{k}$$

¡Es la suma armónica!

$$\log(n) \;\leq\; \sum_{k=1}^{n} \frac{1}{k} \;\leq\; \log(n) + 1$$

$$2 \cdot (n+1) \cdot \left(\sum_{k=1}^{n} \frac{1}{k}\right) - 4 \cdot n$$

Lineal

Logarítmico

Lineal

$$\mathcal{O}(n \log(n))$$