Ayudantía N° 3

Insertion y Merge

Repaso

Pseudocódigo

Insertion Sort

Version Inplace

Input: secuencia A, de largo n >= 2

Output: Nada

```
InsertionSort (A, n):
1     for i = 1 ... n − 1 :
2         j = i
3         while (j > 0) ∧ (A[j] < A[j − 1]) :
4             Intercambiar A[j] con A[j − 1]
5             j = j − 1
```

Pseudocódigo
Analogía

Insertion Sort

Version Not Inplace

Notar que la versión no in place requiere crear una lista nueva a la que se le agrega de forma ordenada el primer elemento de la lista original el cual se elimina de ella.
Notar que esto es finito dado el largo finito de la secuencia original

Veamos la visualización !

https://visualgo.net/en/sorting

Selection Sort v/s Insertion Sort

Forma de ordenación	Selecciona ordenadamente e inserta en el primero	Toma el primero y lo inserta de forma ordenada
Secuencia original ordenada	No cambia la complejidad	Beneficia la complejidad
Mejor caso	O(n^2)	O(n)
Caso promedio	O(n^2)	O(n^2)
Peor caso	O(n^2)	O(n^2)
Memoria adicional (versiones InPlace)	O(1)	O(1)

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).

Utilicemos Merge

Merge Sort

Pseudocódigo

Version not Inplace

Input: Secuencia A

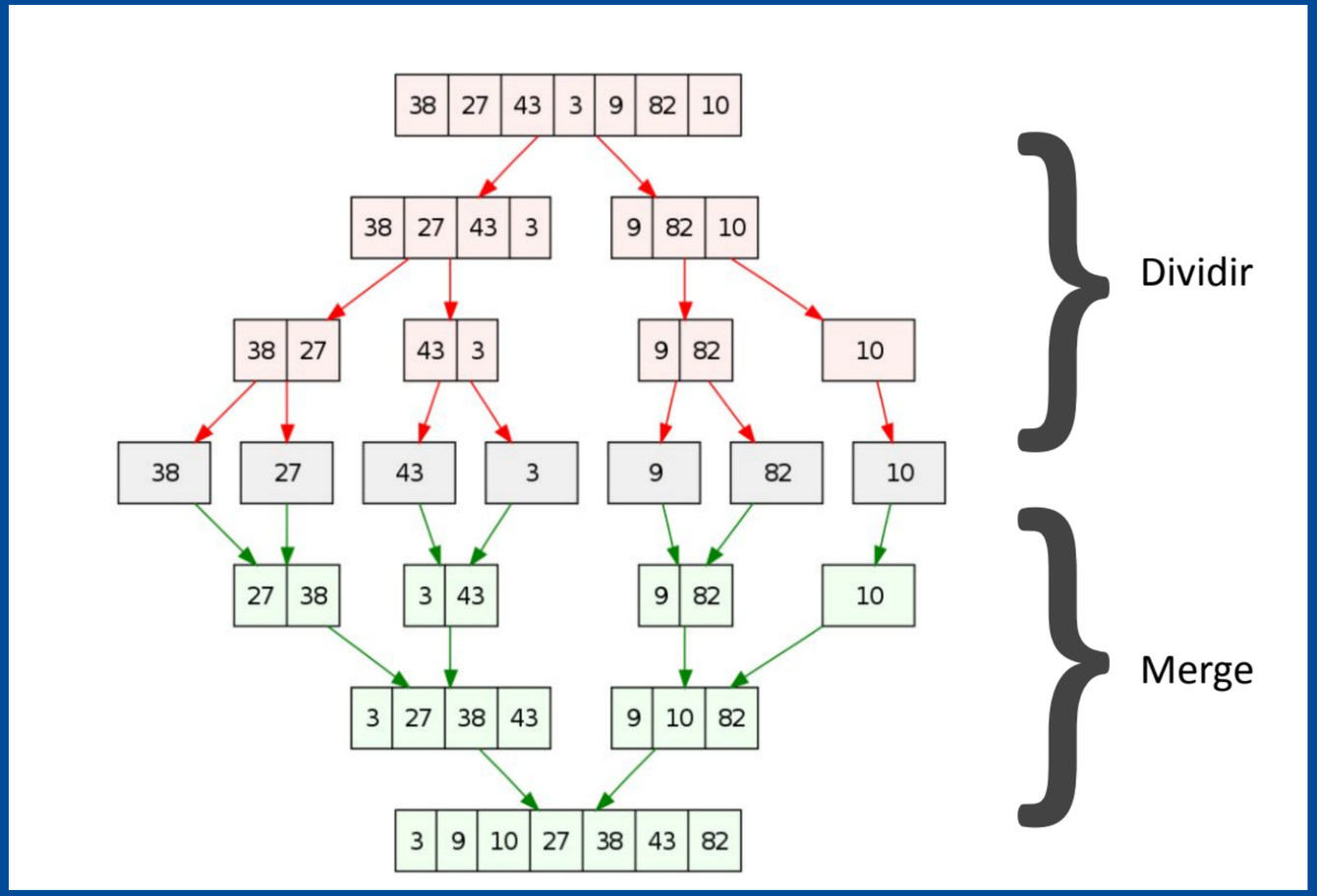
Output: Secuencia ordenada B

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

$O(n)$

Veamos la visualización !

https://visualgo.net/en/sorting

Extra 1: Ejercitemos

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

Extra 2: Ejercitemos

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 $T(n)$ del tiempo que toma este nuevo algoritmo para un arreglo de 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

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),

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

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: 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:

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

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

Finalmente, ¿Qué sucede con la complejidad del algoritmo cuando k tiende a 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

Se divide el arreglo en k arreglos de al menos $\lceil \frac{n}{k} \rceil$ elementos.

$$T(n,k) \leq \underbrace{log_2(k) \cdot n}_{\text{Costo de realizar merge para k arreglos ordenados}} + \overbrace{T\left(\left\lceil \frac{n}{k} \right\rceil, k\right) + T\left(\left\lceil \frac{n}{k} \right\rceil, k\right) + ... + T\left(\left\lceil \frac{n}{k} \right\rceil, k\right)}$$

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).