Ayudantía N° 1

Insertion, Selection & MergeSort

Repaso

Selection Sort

Pseudocódigo

Input: Secuencia A,

Output: Nueva secuencia B ordenada

Visualización

https://www.youtube.com/shorts/HRwi5gwlB0U

```
SelectionSort (A):
1    Definir secuencia B, inicialmente vacía
2    Buscar el menor dato x en A
3    Sacar x de A e insertarlo al final de B
4    Si quedan valores en A, volver a la línea 2
     return B
```

Insertion Sort

Pseudocódigo

Version Inplace

Input: secuencia A, de largo n >= 2

Output: Nada

Visualización

https://www.youtube.com/watch?v=Q1JdRUh1_98&ab_channel=ProfessorPainter

```
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

Merge

Version Not In place

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
Lógica

Merge

Versiones

In place: Usa el mismo espacio reservado a A y B, o sea que la memoria adicional es O(1), y luego de a de mover todos los datos mayores al insertado, lo cual genera un costo en el tiempo al tener complejidad de o(n^2).

NOT in place: Utiliza *memoria adicional* al usar una secuencia nueva para almacenar los elementos de A y B, por ende usa O(n) en memoria adicional, es decir, |A|+|B| = n.

Merge Sort

Pseudocódigo

Version not Inplace

Input: Secuencia A

Output: Secuencia ordenada B

Qué estrategia algorítmica ocupa?

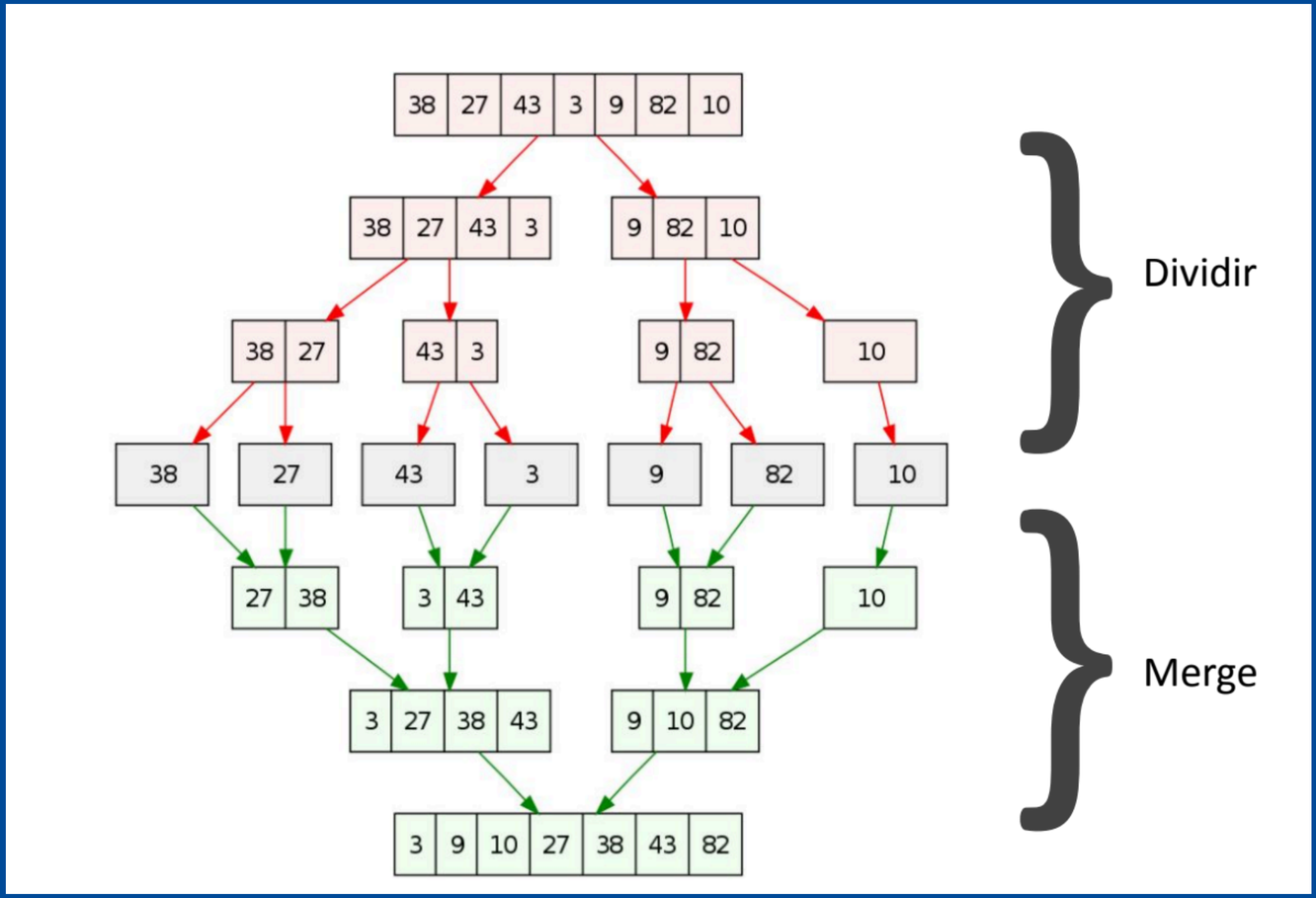
Visualización

https://www.youtube.com/shorts/dZhFmu19N9U

```
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

Ilustración sobre la idea principal y resultado del algoritmo



Complejidad

Mejor, promedio, peor caso

$O(n\log(n))$

Memoria adicional

$O(n)$

Ejercicio 1

Bob ha cocinado **pancakes** y los dejó apilados sobre la mesa, formando una **pila P[0..n - 1] tal que P[0] almacena el pancake que está en la cima de la pila**, mientras que **P[n - 1] representa el que está en la base**. Los pancakes tienen un diámetro representado por un número real. Alice ve la pila y decide **ordenarla** de tal manera que el **diámetro de los pancakes sea descendente desde la base de la pila hacia la cima**. Para realizar la tarea, suponga que cuenta con las siguientes funciones:

- **Diámetro(P[i])**: devuelve el diámetro del pancake P[i], para 0 ≤ i ≤ n - 1. Asuma que todos los diámetros son distintos entre sí.
- **Invertir(i)**: invierte P[0..i], para 0 ≤ i ≤ n - 1. Esta función simula la acción de colocar una espátula debajo del pancake P[i], para luego invertir el orden de los elementos de la pila P[1..i] que queda por encima de la espátula. Asuma que este proceso es in-place. Por ejemplo, P[0..5] = (3, 4, 2, 7, 6, 1), Invertir(3) produce como resultado la pila (7, 2, 4, 3, 6, 1).

Pregunta 1 - I1-2024-1

a) **Escriba el algoritmo PancakeSort(P[0..n - 1])**, que permita ordenar la pila de pancakes P[0..n - 1] como se indica. La única manera permitida de manipular los pancakes de la pila es a través de las funciones Diámetro e Invertir. Asuma que dichas funciones ya están implementadas. Si necesita emplear otra función distinta a esas, debe implementarla. Hay muy poco espacio disponible en la mesa, por lo que su algoritmo debe ser in-place.

Pregunta 1 - I1-2024-1

El algoritmo es similar a Selectionsort, salvo por las restricciones que impone este ejercicio. En cada iteración se debe buscar el siguiente pancake de mayor diámetro, para trasladarlo hacia la parte inferior de la pila usando la función Invertir. El pseudocódigo de una posible solución es el siguiente:

input : Una pila de pancakes $P[0..n-1]$
output: La pila de pancakes P, ordenada

PancakeSort($P[0..n-1]$)
for $i \leftarrow n-1$ **downto** 1 **:**
  $m \leftarrow$ MÁXIMO($P[0..i]$)
  Invertir(m)
  Invertir(i)
return P

Pregunta 1 - I1-2024-1

input : Una pila de pancakes $P[0..n-1]$
output: La pila de pancakes P, ordenada

```
PancakeSort(P[0..n − 1])
for i ← n − 1 downto 1 :
    m ← Máximo(P[0..i])
    Invertir(m)
    Invertir(i)
return P
```

input : Una pila de pancakes $P[0..i]$
output: La posición m tal que $0 \leq m \leq i$ y $P[m]$ es el pancake con el diámetro máximo.

```
Máximo(P[0..i])
m ← 0
for i ← 1 to i :
    if Diámetro(i) > Diámetro(m) :
        m ← i
return m
```

Pregunta 1 - I1-2024-1

b) Demuestre formalmente la correctitud de su algoritmo.

Pregunta 1 - I1-2024-1

b) La demostración de correctitud del algoritmo debe mostrar que:

- PancakeSort termina en una cantidad finita de pasos.
- PancakeSort cumple su propósito.

Para demostrar la finitud del algoritmo, note que realiza siempre n - 1 iteraciones. En cada iteración, busca el pancake de mayor diámetro, lo cual requiere la invocación al algoritmo MÁXIMO(P[0..i]) (que finaliza en i - 1 iteraciones), para luego invocar dos veces a la operación Invertir. En otras palabras, cada una de las operaciones que se realizan en cada iteración de PancakeSort son finitas, y por lo tanto el algoritmo finaliza en una cantidad finita de pasos.

Pregunta 1 - I1-2024-1

La demostración de que PancakeSort cumple su propósito es por inducción, similar a la demostración hecha en clases para SelectionSort. La propiedad que demostraremos es la siguiente:

- **P(i): al finalizar la iteración i (para 1 ≤ i ≤ n) de PancakeSort, P[n - i..n - 1] está ordenada y contiene los i pancakes de mayor diámetro.**

Caso Base: el caso base es para i = 1 (primera iteración). Al finalizar la primera iteración, P(1) se cumple ya que el elemento de mayor diámetro es movido a la base P[n - 1] de la pila.

Hipótesis Inductiva: Al finalizar la iteración i, se cumple P(i).

Paso Inductivo: Demostramos ahora que P(i + 1) se cumple, dado que P(i) (la H.I.) se cumple. Esto último significa que P[n - i..n - 1] está ordenada y contiene los i pancakes de mayor diámetro, como hemos dicho, y por lo tanto P[0..n - i - 1] contiene los n - i pancakes de menor diámetro.

De entre esos pancakes de menor diámetro se selecciona aquel que tiene mayor diámetro, el cual es movido a la posición P[n - (i + 1)] de la pila. Eso significa que P[n - (i + 1)..n - 1] está ordenada y contiene los i + 1 pancakes de mayor diámetro, por lo que P(i + 1) se cumple.

Pregunta 1 - I1-2024-1

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 $T(n)$ 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) 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

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

Ejercicio 3

Una observación que hicimos en clase sobre los algoritmos de ordenación por comparación de elementos adyacentes, p.ej. insertionSort(), es que su debilidad (#operaciones ejecutadas) radica justamente en que sólo comparan e intercambian elementos adyacentes. Así, si tuviéramos un algoritmo que usara la misma estrategia de insertionSort(), pero que comparara elementos que están a una distancia > 1 entre ellos, entonces podríamos esperar un mejor desempeño.

Pregunta 3

Pregunta 3 - a)

a) Calcula cuántas comparaciones entre elementos hace insertionSort() para ordenar el siguiente arreglo a de menor a mayor; muestra que entiendes cómo funciona insertionSort():

a = [11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1]

Pregunta 3 - a) Solución

insertionSort() coloca el segundo elemento ordenado con respecto al primero, luego el tercero ordenado con respecto a los dos primeros (ya ordenados entre ellos), luego el cuarto ordenado con respecto a los tres primeros (ya ordenados entre ellos), etc.

En el caso del arreglo de a), insertionSort() básicamente va moviendo cada elemento, 10, 9, …, 1, hasta la primera posición del arreglo. Para ello, el 10 es comparado una vez (con el 11), el 9 es comparado dos veces (con el 11 y con el 10), el 8 es comparado tres veces (con el 11, el 10 y el 9), y así sucesivamente; finalmente, el 1 es comparado 10 veces.

Luego el total de comparaciones es:
1 + 2 + 3 + … + 10 = 55.

Pregunta 3 - b)

b) Calcula ahora cuántas comparaciones entre elementos hace el siguiente algoritmo **shellSort()** para ordenar el mismo arreglo a. Muestra que entiendes cómo funciona shellSort(); en particular, ¿qué relación tiene con insertionSort()?

```
shellSort(a):
   gaps[] = {5,3,1}
   t = 0
   while t < 3:
      gap = gaps[t]
      j = gap
      while j < a.length]:
         tmp = a[j]
         k = j
         while k >= gap and tmp < a[k-gap]:
            a[k] = a[k-gap]
            k = k-gap
         a[k] = tmp
         j = j+1
   t = t+1
```

Pregunta 3 - b)

```
shellSort(a):
    gaps[] = {5,3,1}
    t = 0
    while t < 3:
        gap = gaps[t]
        j = gap
        while j < a.length]:
            tmp = a[j]
            k = j
            while k >= gap and tmp < a[k-gap]:
                a[k] = a[k-gap]
                k = k-gap
            a[k] = tmp
            j = j+1
    t = t+1
```

Pregunta 3 - b) Solución

Notemos que las comparaciones entre elementos de a se dan sólo en la comparación tmp < a[k-gap]; el algoritmo realiza 11 de estas comparaciones con resultado true y otras 17 con resultado false; en total, 28.

Primero, realiza insertionSort entre elementos que están a distancia 5 entre ellos (según las posiciones que ocupan en a, no en cuanto a sus valores): el 6 con respecto al 11, el 5 c/r al 10, el 4 c/r al 9, el 3 c/r al 8, el 2 c/r al 7, el 1 c/r al 11, y el 1 c/r al 6.

Luego, realiza insertionSort entre elementos que están a distancia 3 (nuevamente, según sus posiciones en el arreglo): el 2 c/r al 5 y el 7 c/r al 10.

Finalmente, realiza insertionSort entre elementos que están a distancia 1: el 3 c/r al 4 y el 8 c/r al 9; estos son los únicos pares de valores que aún están “desordenados” al finalizar el paso anterior.

Feedback