Ayudantía N° 3

# Merge y Partition

# 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
```

# 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

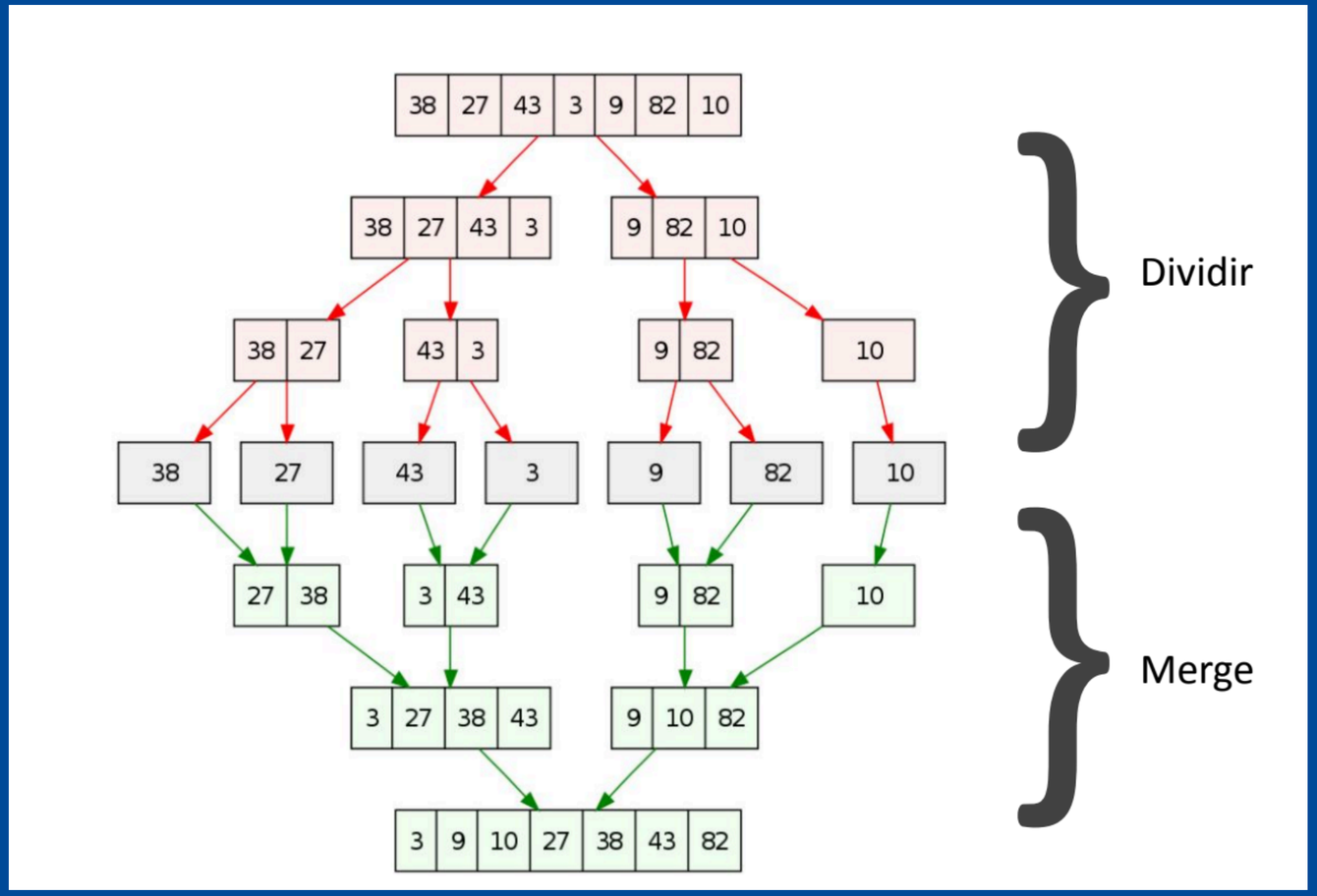
**Output:** Secuencia ordenada B

**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**

$O(nlog(n))$

**Memoria adicional**

$O(n)$

# Veamos la visualización !

https://visualgo.net/en/sorting

# 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**

# 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:

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

# 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

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

# Búsqueda Binaria

# Pseudocodigo

- Elige posicion media m
- Si el elemento en la pos m es igual al buscado retorno m
- Si elemento en la pos m es mayor busco por el lado izquierdo del array. eoc busco por lado derecho.

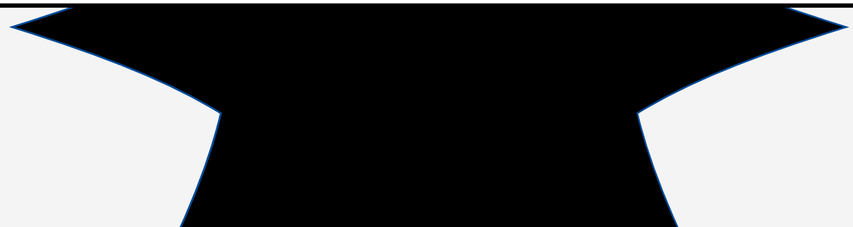
```
BSearch (A, x, i, f):
1       if f < i : return −1
2       m ← ⌊(i + f) / 2⌋
3       if A[m] = x : return m
4       if A[m] > x :
5           return BSearch (A, x, i, m − 1)
6       return BSearch (A, x, m + 1, f)
```

# Ejercicio 3

Tienes dos arrays A y B, de tamaño n y m respectivamente. Para cada elemento del segundo array b_i tienes que encontrar cuantos elementos en A son menores o iguales al valor de b_i.

Source

## Pregunta 3



## Pregunta 3 - Solucion

Uno de los requisitos para que se pueda realizar efectivamente la busqueda binaria, es que el array A se encuentre ordenado.

Luego, para cada elemento b_i haremos una busqueda binaria sobre A para encontrar la posicion del ultimo elemento menor o igual a b_i. Para esto preguntaremos si:

$$A[mid] <= B[i]$$

## Pregunta 3 - Solucion

Por ejemplo, sea:

A = [1 2 3 4 5 6 6 7 8 9 10]
b_i = 6

Si preguntamos A[j] <= b_i para todo j tendremos que:

T T T T T T T F F F F

Finalmente, nos interesa la pos del ultimo True para responder cuantos son menores o iguales a b_i.

# Pregunta 3 - Solucion

Pseudo codigo, version no recursiva de Binary Search

```
// Para cada b_i
for (int i = 0; i < m; i++){

    // Definimos los limites
    int left = 0, right = n - 1, ans = 0;
    while(left <= right)
    {
        // Pos mediana
        int mid = (left + right) / 2;

        // Revisamos la condicion
        if (a[mid] <= b[i]){
            left = mid + 1;
            ans = left;
            }
        else
            right = mid - 1;
    }
    print(ans);
}
```

# Partition

# Repaso

# Pseudocodigo

- 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
```

# Ejercicio 4

1) Escribe el algoritmo partition3, que, en lugar de particionar el arreglo A en dos, 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)

```
1: p ← elemento aleatorio en A[i, f]
2: m, M, P ← secuencias vacías
3: for x in A[i, f] do
4:     if x < p then
5:         Insertar x en m
6:     else if x = p then
7:         Insertar x en P
8:     else if x > p then
9:         Insertar x en M
10:    end if
11: end for
12: A[i, f] ← Concat(m, p, P, M)
13: return i + |m|, i + |m| + |P|
```

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

**Algorithm 2** Partition3(A, i, f)

```
1:  x ← índice aleatorio en {i, ..., f}
2:  p ← A[x]
3:  swap(A[x], A[f])
4:  j ← i
5:  l ← i
6:  for k in {i, ..., f − 1} do
7:      if A[k] < p then
8:          swap(A[k], A[j])
9:          swap(A[j], A[l])
10:         j ← j + 1
11:         l ← l + 1
12:     end if
13:     if A[k] = p then
14:         swap(A[k], A[j])
15:         j ← j + 1
16:     end if
17: end for
18: swap(A[j], A[f])
19: return l, j
```

# Feedback

