Repaso I1

SelectionSort, InsertionSort, Correctitud y Divide & Conquer

Tips generales

- 4 Preguntas (Correctitud, Sorting, Dividir y vencer)
- Preguntas en partes (Leer toda la prueba antes para de esa forma analizar la dificultad)
- Torpedo: Hoja tamaño carta manuscrita (Puede ser escrita digitalmente, pero con manuscrita)

Cuando un algoritmo es correcto

Cuando un algoritmo es correcto

- Termina en un número finito de pasos

Cuando un algoritmo es correcto

- Termina en un número finito de pasos
- Para todo input valido, Cumple su propósito

Cuando un algoritmo es correcto

Un buen método para demostrar correctitud en un algoritmo es utilizar inducción matemática

- Esto se debe a que en teoría debemos probar el algoritmo para cualquier input valido posible.
  Y esto es precisamente lo que la inducción nos permite hacer

Recordatorio Inducción

- Sea $S(n) = \Sigma^{n}_{i}$, Esto es, la suma de todos los naturales hasta n
- Definimos nuestro *"algoritmo"* Como una forma mecánica y eficiente de llegar al valor buscado

$$F(n) = \frac{n(n+1)}{2}$$

¿Es nuestro algoritmo correcto?

Recordatorio Inducción

- Base Inductiva (BI):

Para la base inductiva hemos de mostrar que al aplicar el algoritmo sobre el primer elemento válido, entonces el output es correcto.

$$F(1) = S(1) \Rightarrow \frac{1(2)}{1} = 1$$

- Hipotesis de Induccion (HI):

En la hipótesis inductiva hemos de declarar nuestra hipótesis, para posteriormente utilizarla de forma inductiva

$$F(n) = \frac{n(n+1)}{2} = S(n) = \sum_{i=0}^{n} i$$

Recordatorio Inducción

- Tesis Inductiva (TI):

En este paso hemos de demostrar que si F(n) se cumple, entonces F(n+1) también ha de cumplirse (F(n) → F(n+ 1))

$$F(n+1) = F(n) + (n+1) = \frac{n(n+1)}{2} + n + 1 = \frac{n^2 + n + 2n + 2}{2}$$

$$\frac{n^2 + 3n + 2}{2} = \frac{(n+2) * (n+1)}{2}$$

$$F(n+1) = F(n) + (n+1) = S(n) + (n+1)$$

Por Inducción, F(n) es Correcto

Pregunta 1 - I1-2019-2

El siguiente algoritmo retorna el índice de un elemento x en el arreglo ordenado A de n datos, entre los índices i y f. Si el elemento no está en A[i,f], entonces retorna -1.

```
search(A, x, i, f):
    if f < i:
        return −1
    m = ⌊(i+f)/2⌋
    if A[m] > x:
        return search(A, x, i, m − 1)
    else if A[m] < x:
        return search(A, x, m + 1, f)
    else:
        return m
```

a) Demuestra que este algoritmo es correcto, es decir, que termina y efectivamente retorna lo que queremos que retorne
b) Plantea la recurrencia de tiempo T(n) que toma el algoritmo
c) Obtén la complejidad de este algoritmo en notación O, a partir de la recurrencia

Pregunta 1 - I1-2019-2

a) Demuestra que este algoritmo es correcto, es decir, que termina y efectivamente retorna lo que queremos que retorne

Finitud:

Tenemos dos casos

```
search(A, x, i, f):
    if f < i:
        return −1
    m = ⌊(i+f)/2⌋
    if A[m] > x:
        return search(A, x, i, m − 1)
    else if A[m] < x:
        return search(A, x, m + 1, f)
    else:
        return m
```

Pregunta 1 - I1-2019-2

a) Demuestra que este algoritmo es correcto, es decir, que termina y efectivamente retorna lo que queremos que retorne

Finitud:

Tenemos dos casos

1) Al elegir m, este es el elemento que buscamos, y el algoritmo termina

```
search(A, x, i, f):
    if f < i:
        return −1
    m = ⌊(i+f)/2⌋
    if A[m] > x:
        return search(A, x, i, m − 1)
    else if A[m] < x:
        return search(A, x, m + 1, f)
    else:
        return m
```

Pregunta 1 - I1-2019-2

a) Demuestra que este algoritmo es correcto, es decir, que termina y efectivamente retorna lo que queremos que retorne

Finitud:

Tenemos dos casos

1) Al elegir m, este es el elemento que buscamos, y el algoritmo termina

1) Se hace un llamado recursivo, con la mitad del arreglo

```
search(A, x, i, f):
    if f < i:
        return −1
    m = ⌊(i+f)/2⌋
    if A[m] > x:
        return search(A, x, i, m − 1)
    else if A[m] < x:
        return search(A, x, m + 1, f)
    else:
        return m
```

Pregunta 1 - I1-2019-2

a) Demuestra que este algoritmo es correcto, es decir, que termina y efectivamente retorna lo que queremos que retorne

Finitud:

Si se hace un llamado recursivo, el tamaño del arreglo disminuye

```
search(A, x, i, f):
    if f < i:
        return -1
    m = ⌊(i+f)/2⌋
    if A[m] > x:
        return search(A, x, i, m-1)
    else if A[m] < x:
        return search(A, x, m+1, f)
    else:
        return m
```

Pregunta 1 - I1-2019-2

a) Demuestra que este algoritmo es correcto, es decir, que termina y efectivamente retorna lo que queremos que retorne

Finitud:

Si se hace un llamado recursivo, el tamaño del arreglo disminuye

Si se sigue repitiendo el caso 2), el tamaño del arreglo disminuye hasta que no tenga elementos, y se retorna -1.

```
search(A, x, i, f):
    if f < i:
        return −1
    m = ⌊(i+f)/2⌋
    if A[m] > x:
        return search(A, x, i, m − 1)
    else if A[m] < x:
        return search(A, x, m + 1, f)
    else:
        return m
```

Pregunta 1 - I1-2019-2

a) Demuestra que este algoritmo es correcto, es decir, que termina y efectivamente retorna lo que queremos que retorne

Correctitud:

```
search(A, x, i, f):
    if f < i:
        return -1
    m = ⌊(i+f)/2⌋
    if A[m] > x:
        return search(A, x, i, m-1)
    else if A[m] < x:
        return search(A, x, m+1, f)
    else:
        return m
```

Pregunta 1 - I1-2019-2

a) Demuestra que este algoritmo es correcto, es decir, que termina y efectivamente retorna lo que queremos que retorne

Correctitud:

BI: caso con largo de A = 1

```
search(A, x, i, f):
    if f < i:
        return -1
    m = ⌊(i+f)/2⌋
    if A[m] > x:
        return search(A, x, i, m-1)
    else if A[m] < x:
        return search(A, x, m+1, f)
    else:
        return m
```

Pregunta 1 - I1-2019-2

a) Demuestra que este algoritmo es correcto, es decir, que termina y efectivamente retorna lo que queremos que retorne

Correctitud:

BI: caso con largo de A = 1

HI: suponemos que el algoritmo es correcto para un largo de A < n

```
search(A, x, i, f):
    if f < i:
        return -1
    m = ⌊(i+f)/2⌋
    if A[m] > x:
        return search(A, x, i, m-1)
    else if A[m] < x:
        return search(A, x, m+1, f)
    else:
        return m
```

Pregunta 1 - I1-2019-2

a) Demuestra que este algoritmo es correcto, es decir, que termina y efectivamente retorna lo que queremos que retorne

Correctitud:

BI: caso con largo de A = 1

HI: suponemos que el algoritmo es correcto para un largo de A < n

TI: tenemos un arreglo A de largo n

```
search(A, x, i, f):
    if f < i:
        return -1
    m = ⌊(i+f)/2⌋
    if A[m] > x:
        return search(A, x, i, m-1)
    else if A[m] < x:
        return search(A, x, m+1, f)
    else:
        return m
```

Pregunta 1 - I1-2019-2

a) Demuestra que este algoritmo es correcto, es decir, que termina y efectivamente retorna lo que queremos que retorne

Correctitud:

BI: caso con largo de A = 1

HI: suponemos que el algoritmo es correcto para un largo de A < n

TI: tenemos un arreglo A de largo n

1) A[m] es el elemento, se retorna A[m]
2) Se hace un llamado recursivo, con un arreglo de largo menor a n, se aplica HI

```
search(A, x, i, f):
    if f < i:
        return -1
    m = ⌊(i+f)/2⌋
    if A[m] > x:
        return search(A, x, i, m-1)
    else if A[m] < x:
        return search(A, x, m+1, f)
    else:
        return m
```

Pregunta 1 - I1-2019-2

b) Plantea la recurrencia de tiempo T(n) que toma el algoritmo

```
search(A, x, i, f):
    if f < i:
        return −1
    m = ⌊(i+f)/2⌋
    if A[m] > x:
        return search(A, x, i, m − 1)
    else if A[m] < x:
        return search(A, x, m + 1, f)
    else:
        return m
```

Pregunta 1 - I1-2019-2

b) Plantea la recurrencia de tiempo T(n) que toma el algoritmo

$$T(n) = \begin{cases} 1, & n = 1 \\ T\left(\left\lceil \frac{n-1}{2} \right\rceil\right) + 1, & n > 1 \end{cases}$$

```
search(A, x, i, f):
    if f < i:
        return −1
    m = ⌊(i+f)/2⌋
    if A[m] > x:
        return search(A, x, i, m − 1)
    else if A[m] < x:
        return search(A, x, m + 1, f)
    else:
        return m
```

Pregunta 1 - I1-2019-2

c) Obtén la complejidad de este algoritmo en notación O, a partir de la recurrencia

$$T(n) = \begin{cases} 1, & n = 1 \\ T\left(\left\lceil \frac{n-1}{2} \right\rceil\right) + 1, & n > 1 \end{cases}$$

Pregunta 1 - I1-2019-2

c) Obtén la complejidad de este algoritmo en notación O, a partir de la recurrencia

$$T(n) = \begin{cases} 1, & n = 1 \\ T\left(\left\lceil \frac{n-1}{2} \right\rceil\right) + 1, & n > 1 \end{cases}$$

Las operaciones “piso” o “techo” son muy molestas, cómo nos las sacamos de encima?

Pregunta 1 - I1-2019-2

c) Obtén la complejidad de este algoritmo en notación O, a partir de la recurrencia

$$T(n) = \begin{cases} 1, & n = 1 \\ T\left(\left\lceil \frac{n-1}{2} \right\rceil\right) + 1, & n > 1 \end{cases}$$

Las operaciones “piso” o “techo” son muy molestas, cómo nos las sacamos de encima?

Potencias de 2!

Pregunta 1 - I1-2019-2

c) Obtén la complejidad de este algoritmo en notación O, a partir de la recurrencia

$$T(n) = \begin{cases} 1, & n = 1 \\ T\left(\left\lceil \frac{n-1}{2} \right\rceil\right) + 1, & n > 1 \end{cases}$$

Las operaciones “piso” o “techo” son muy molestas, cómo nos las sacamos de encima?

Potencias de 2!

Acotaremos por arriba nuestro T(n), para poder utilizar potencias de 2 y eliminar el techo

Pregunta 1 - I1-2019-2

c) Obtén la complejidad de este algoritmo en notación O, a partir de la recurrencia

$$T(n) = \begin{cases} 1, & n = 1 \\ T\left(\left\lceil \frac{n-1}{2} \right\rceil\right) + 1, & n > 1 \end{cases}$$

$$T(n) \leq T'(n) = \begin{cases} 1, & n = 1 \\ T'\left(\frac{n}{2}\right) + 1, & n > 1 \end{cases}$$

Pregunta 1 - I1-2019-2

c) Obtén la complejidad de este algoritmo en notación O, a partir de la recurrencia

Tomaremos un 2^k tal que $n \leq 2^k < 2n$, donde tendremos que $T'(n) \leq T'(2^k)$

Pregunta 1 - I1-2019-2

c) Obtén la complejidad de este algoritmo en notación O, a partir de la recurrencia

Tomaremos un 2^k tal que n ≤ 2^k < 2n, donde tendremos que T'(n) ≤ T'(2^k)

$$T'(2^k) = T'(2^{k-1}) + 1$$

Pregunta 1 - I1-2019-2

c) Obtén la complejidad de este algoritmo en notación O, a partir de la recurrencia

Tomaremos un 2^k tal que n ≤ 2^k < 2n, donde tendremos que T'(n) ≤ T'(2^k)

$$T'(2^k) = T'(2^{k-1}) + 1$$

$$T'(2^k) = T'(2^{k-2}) + 1 + 1$$

$$T'(2^k) = T'(2^{k-3}) + 1 + 1 + 1$$

Pregunta 1 - I1-2019-2

c) Obtén la complejidad de este algoritmo en notación O, a partir de la recurrencia

Tomaremos un 2^k tal que n ≤ 2^k < 2n, donde tendremos que T'(n) ≤ T'(2^k)

$$T'(2^k) = T'(2^{k-1}) + 1$$

$$T'(2^k) = T'(2^{k-2}) + 1 + 1$$

$$T'(2^k) = T'(2^{k-3}) + 1 + 1 + 1$$

$$T'(2^k) = T'(2^{k-i}) + i$$

Pregunta 1 - I1-2019-2

c) Obtén la complejidad de este algoritmo en notación O, a partir de la recurrencia

$$T'(2^k) = T'(2^{k-i}) + i$$

Pregunta 1 - I1-2019-2

c) Obtén la complejidad de este algoritmo en notación O, a partir de la recurrencia

$$T'(2^k) = T'(2^{k-i}) + i$$

Y si k = i:

Pregunta 1 - I1-2019-2

c) Obtén la complejidad de este algoritmo en notación O, a partir de la recurrencia

$$T'(2^k) = T'(2^{k-i}) + i$$

Y si k = i:

$$T'(2^k) = T'(1) + k$$

Pregunta 1 - I1-2019-2

c) Obtén la complejidad de este algoritmo en notación O, a partir de la recurrencia

$$T'(2^k) = T'(2^{k-i}) + i$$

Y si k = i:

$$T'(2^k) = T'(1) + k$$

$$T'(1) = 1 \qquad 2^k < 2n \qquad k < \log_2 2n:$$

Pregunta 1 - I1-2019-2

c) Obtén la complejidad de este algoritmo en notación O, a partir de la recurrencia

Recordando que

$$T(n) \leq T'(n) \leq T'(2^k)$$

Y con

$$T'(2^k) < 1 + \log_2 2n$$

Pregunta 1 - I1-2019-2

c) Obtén la complejidad de este algoritmo en notación O, a partir de la recurrencia

Recordando que

$$T(n) \leq T'(n) \leq T'(2^k)$$

Y con

$$T'(2^k) < 1 + \log_2 2n$$

Tenemos que

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

Pregunta 2 - l1-2021-1

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 2 - I1-2021-1

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

Pregunta 2 - I1-2021-1

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

Sabemos que **insertionSort** toma tiempo **O(n^2)** en arreglos de largo **n**

Pregunta 2 - I1-2021-1

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

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)**

Pregunta 2 - I1-2021-1

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

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

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

Pregunta 2 - 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 2 - 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

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

Pregunta 2 - 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

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)**.

Pregunta 2 - 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

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)**

Pregunta 2 - 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

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?

Pregunta 2 - 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

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 **log2(n/k)** veces

Pregunta 2 - 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

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 **log2(n/k)** veces

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

Pregunta 2 - 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 2 - 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

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

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

Pero, podemos hacerlo mejor todavía?

Pregunta 2 - 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

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

Pregunta 2 - 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

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

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

Pregunta 2 - 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

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

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

Pregunta 2 - 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

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

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

3) Estrategias algorítmicas. Sean X e Y conjuntos de datos **no** ordenados. Lo que se busca hacer es lo siguiente: Se calcula el producto cartesiano entre ambos conjuntos; o sea, todos los pares posibles con un elemento de X y otro de Y. Sea Z este nuevo conjunto.

Formalmente, producto cartesiano $Z = \{(x_i, y_i) \mid x_i \in X, y_i \in Y\}$.

Se pide ordenar dichos pares según el valor de la suma $z = x + y$. Y en caso de empates, dejar primero al que estaba primero con respecto a x. Asume que cada suma en Z es única.

a) Describe una forma de calcular las sumas de forma eficiente en un entorno altamente paralelizado
b) Dadas las sumas, describe una estructura que permita mantener los pares ordenados y la suma correspondiente
c) Finalmente, describe un algoritmo de ordenación que ordene eficientemente los pares ordenados basándose en el resultado de la suma

Hint: Recuerda estrategias algorítmicas vistas en clases.

Solución propuesta: a y b)

```
Z_sum(arr, i,f):
  if(i==f){
    arr[i] = (arr[0]+arr[1],arr[0],arr[1]);
  }
  else if (i<f){
    m = i+(f-1)/2;
    Z_sum(arr,i,m);
    Z_sum(arr,m+1,f);
```

Solución propuesta: c)

```
Z_sort(arr, i,f):
  if (i<f){
    m = i+(f-1)/2;
    Z_sort(arr,i,m);
    Z_sort(arr,m+1,f);
    merge(arr,i,m,f);}
```

```
merge(arr,i,m,f):
    l1= m-i+1;
    l2= f-m;
    arr_1[l1], arr_2[l2];
    for(int e=0;e<l1;e++){
        arr_1[e] = arr[i+e];}
    for(int e=0;e<l2;e++){
        arr_2[e] = arr[m+1+e];
    int a=0;
    int b=0;
    int k=i;
    while(a<l1 && b<l2){
        if(arr_1[a][0] <= arr_2[b][0]){
            arr[k] = arr_1[a];
            a++;}
        else{
            arr[k] = arr_2[b];
            b++;}
        k++;
    }
    while (a<l1){
        arr[k] = arr_1[a];
        a++;
        k++;
    }
    while (b<l2){
        arr[k] = arr_2[b];
        b++;
        k++;
    }
}
```

- Quicksort

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

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.

- Quicksort

La subrutina partition para arreglos que vimos en clases:

```
partition(A, i, f):
    x ← un indice aleatorio en [i, f]
    p ← A[x]
    A[x] ⇄ A[f]
    j ← i
    for k ∈ [i, f − 1]:
        if A[k] < p:
            A[j] ⇄ A[k]
            j ← j + 1
    A[j] ⇄ A[f]
    return j
```

- Quicksort

Podemos ver que hay dos intercambios que siempre se hacen, que son poner el pivote al final del arreglo y luego moverlo a la posición que separa los menores de los mayores.

Los demás intercambios se hacen solo cuando encontramos una clave que es menor al pivote. Por lo tanto, en el caso de que se hagan la mayor cantidad de intercambios es cuando todas las claves del subarreglo son menores al pivote.

Esto significa que si el subarreglo tenía m elementos, se hacen $m+1$ intercambios, y el subarreglo se separa en dos subarreglos disparejos, uno de largo 0 y el otro de largo $m-1$.

Esto significa que el tiempo que toma Quicksort para un arreglo inicial de n elementos es:

$$T(n) = n + (n-1) + (n-2) + \cdots + 1$$

$$T(n) \in \mathcal{O}(n^2)$$

El cual es el peor caso de Quicksort.

Por otro lado, sabemos que el mejor caso de Quicksort se produce cuando partition separa el subarreglo en dos mitades de el mismo largo (o lo más parejo posible).

Esto significa que la mitad de los elementos del subarreglo deben ser mayores al pivote y otra mitad menores al pivote. Ya vimos que los intercambios se hacen sólo para los elementos menores al pivote, por lo que en este caso tendríamos **la mitad** de intercambios que en el caso anterior.

- Mergesort

$MergeSort$ 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.

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?

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?

- Mergesort

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:

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

- Mergesort

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),

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

- Mergesort

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:

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

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

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

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 $p > q$.

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:

Tips generales

- Leer la prueba con calma
- Escribir en cada pregunta una primera idea de solución pensada
- Hacer el resumen (Sirve como herramienta de estudio)