



Ayudantía 2

Median, Mergesort, Quicksort

Cristobal Berrios
Nicolas Fraga
Juan Isamitt
José Tomás Jiménez



Repaso de algoritmos

Algoritmos de dividir y conquistar

1. Mergesort
2. Quicksort



MERGESORT != MERGE



Merge

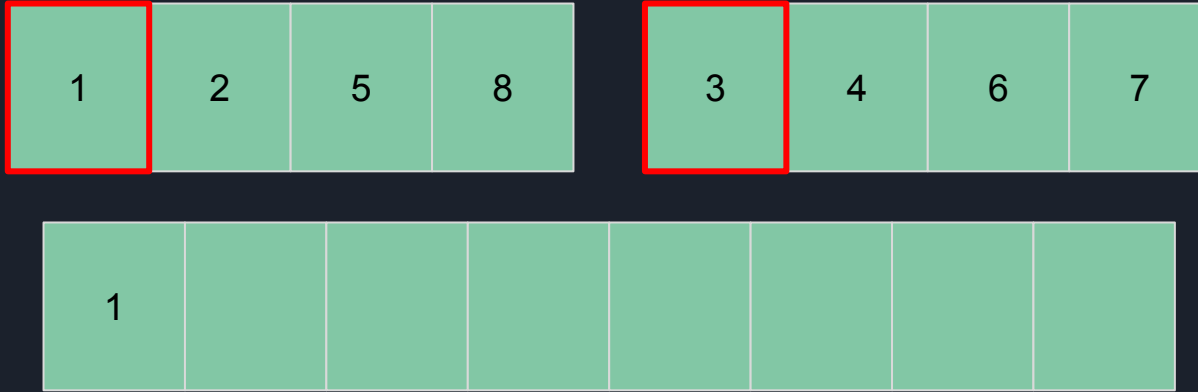
1	2	5	8
---	---	---	---

3	4	6	7
---	---	---	---

--	--	--	--	--	--	--	--

- Se tienen dos conjuntos de elementos ORDENADOS por sí solos.
- Se crea un conjunto vacío de tamaño igual a la suma del tamaño de ambos conjuntos.

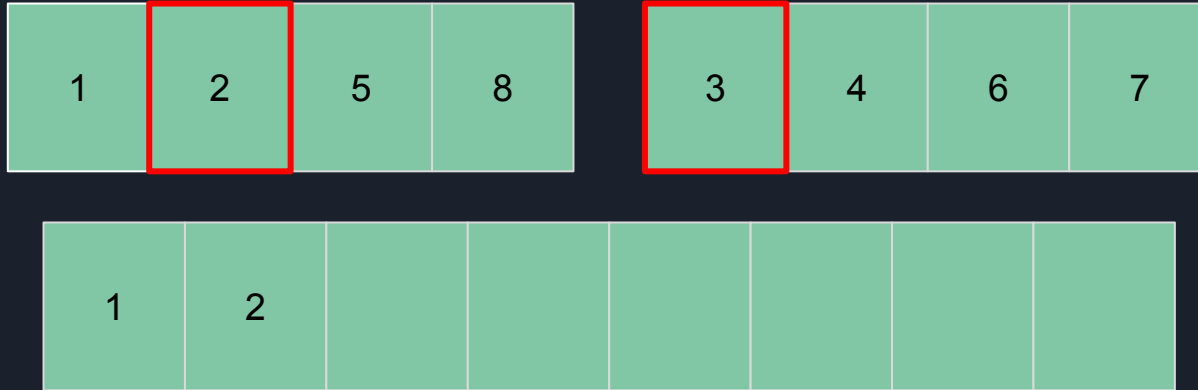
Merge



- Elegir el valor más chico.
- Ponerlo en la lista vacía.
- Seguir con el siguiente de la lista de donde viene el elemento que se escogió.



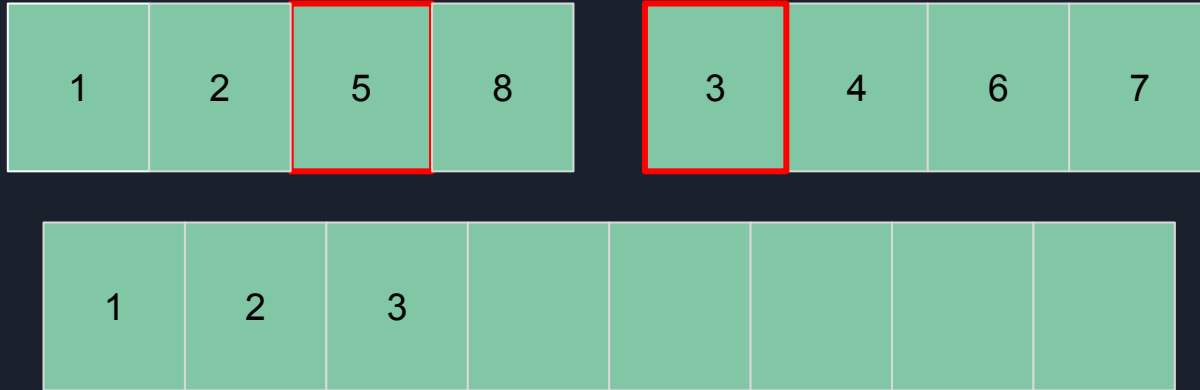
Merge



- Seguir así.



Merge





Merge

1	2	5	8
---	---	---	---

3	4	6	7
---	---	---	---

1	2	3	4				
---	---	---	---	--	--	--	--



Merge

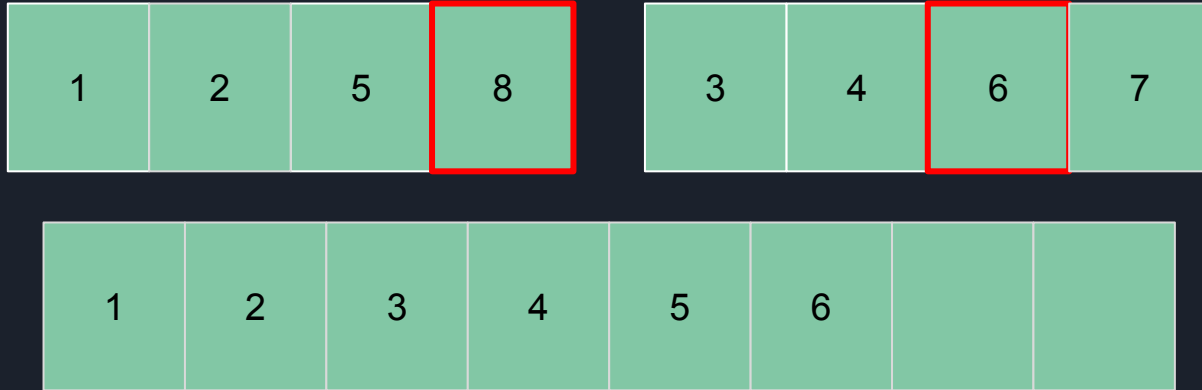
1	2	5	8
---	---	---	---

3	4	6	7
---	---	---	---

1	2	3	4	5			
---	---	---	---	---	--	--	--



Merge





Merge

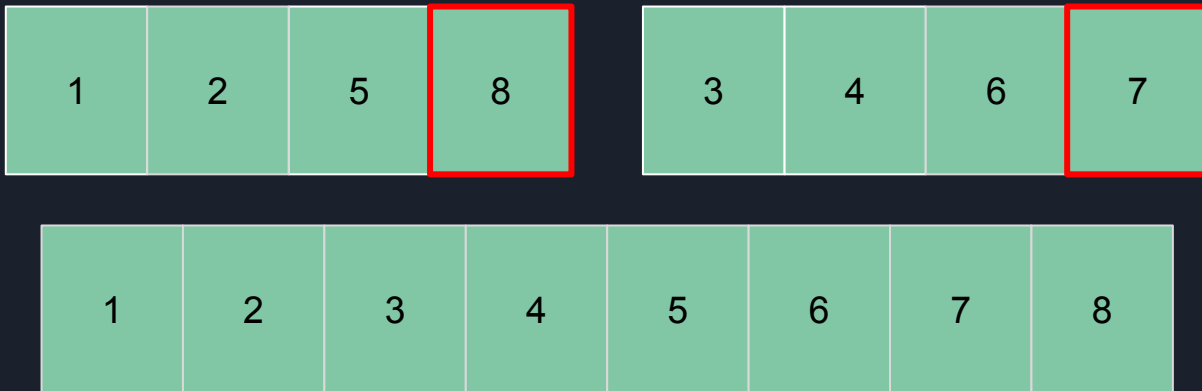
1	2	5	8
---	---	---	---

3	4	6	7
---	---	---	---

1	2	3	4	5	6	7	
---	---	---	---	---	---	---	--



Merge



- El algoritmo debe saber qué tan larga es cada lista para detenerse con el último elemento.



Mergesort

1	8	5	2	4	6	7	3
---	---	---	---	---	---	---	---

- Dividir secuencia a la mitad
- Ordenar cada mitad de forma recursiva
- Unir las mitades mediante merge



Mergesort

1	8	5	2	4	6	7	3
---	---	---	---	---	---	---	---



Mergesort

1	8	5	2
---	---	---	---

4	6	7	3
---	---	---	---



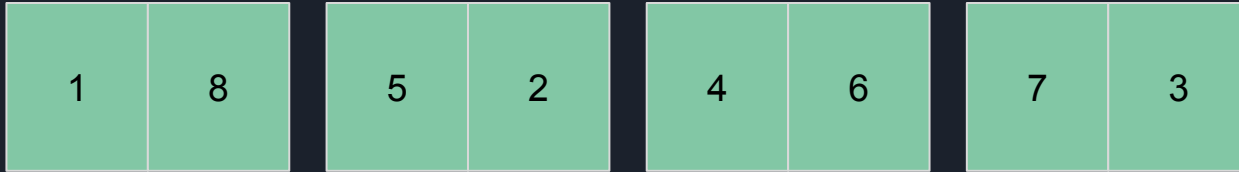
Mergesort

1	8	5	2
---	---	---	---

4	6	7	3
---	---	---	---

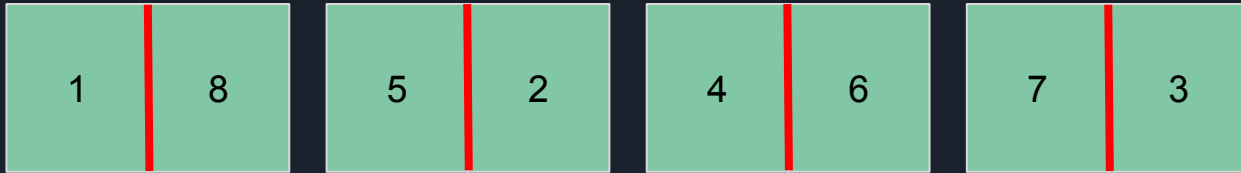


Mergesort





Mergesort





Mergesort

1

8

5

2

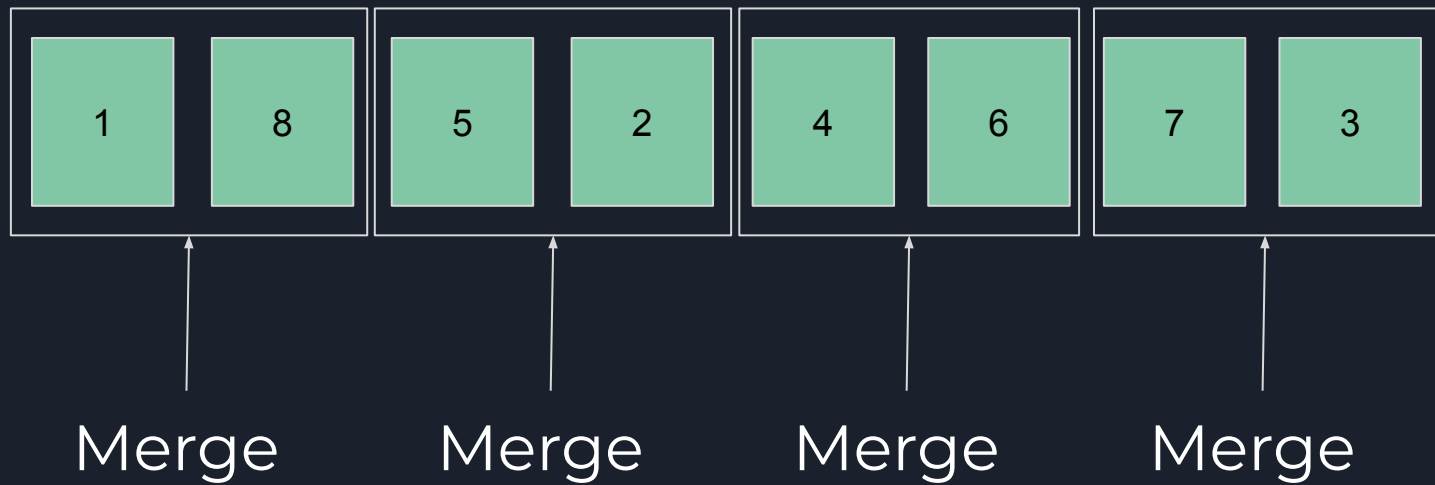
4

6

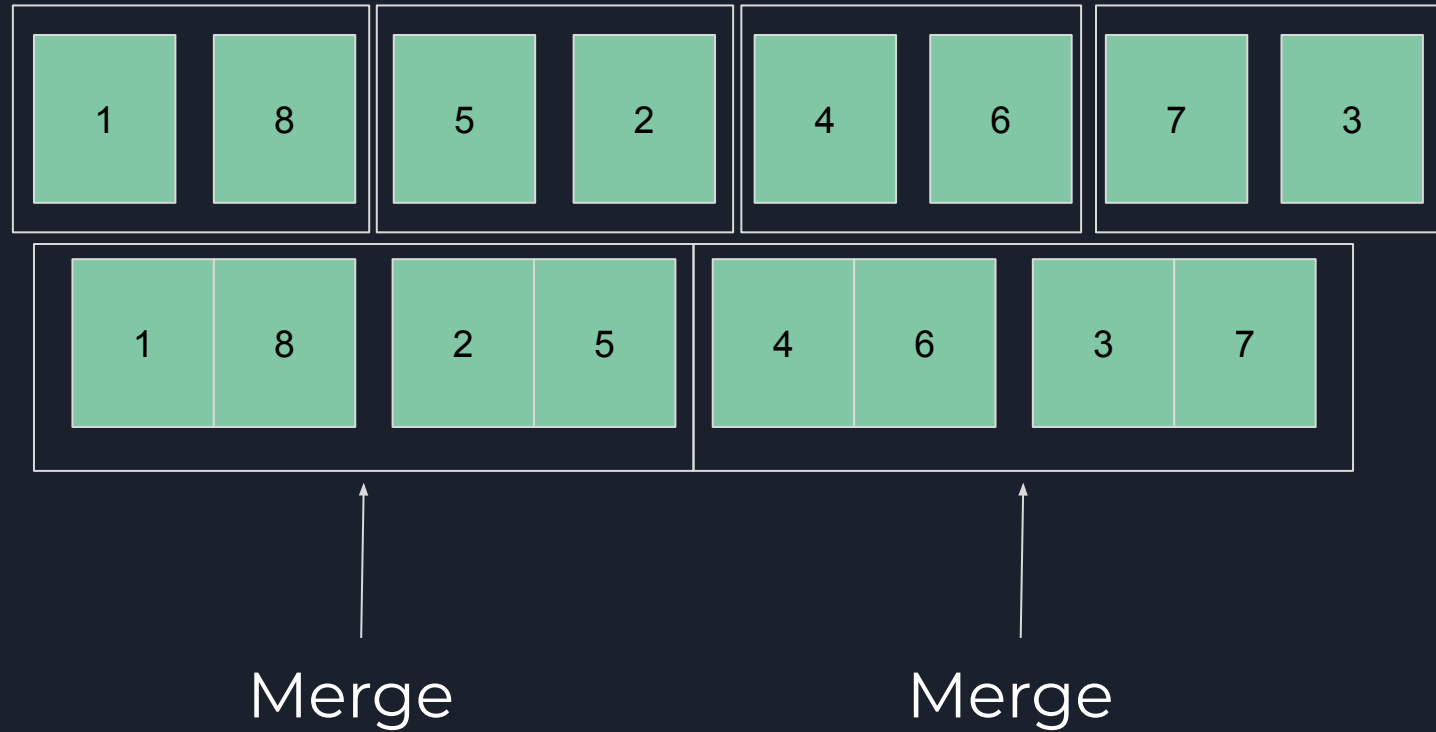
7

3

Mergesort



Mergesort

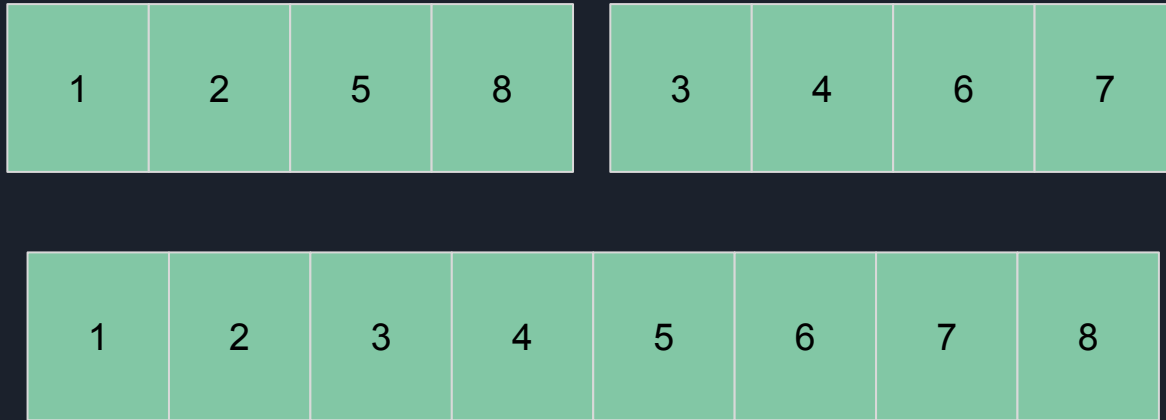


Mergesort





Mergesort



ORDENADO!!!!



Mergesort

```
merge(A, B):  
    C = Arreglo vacío  
    while A y B no estén vacíos:  
        a = primer elemento de A  
        b = primer elemento de B  
        if a <= b:  
            C.append(a)  
            Eliminamos a de A  
        else if a > b:  
            C.append(b)  
            Eliminamos b de B  
    if A está vacío:  
        C.append(B)  
    else if B.está vacío:  
        C.append(A)  
    return C
```

```
mergeSort(A, i, f):  
    if f - i > 1:  
        m = round((f - i)/2)  
        B = mergeSort(A, i, m)  
        C = mergeSort(A, m, f)  
        A[i:f] = merge(B, C)  
    return A[i:f]
```




Mergesort

1. Suponiendo que Merge es correcto y $O(n)$:

- a) Demuestre que MergeSort es correcto
- b) Calcule la complejidad de MergeSort

```
mergeSort(A, i, f):  
    if f - i > 1:  
        m = round((f - i)/2)  
        B = mergeSort(A, i, m)  
        C = mergeSort(A, m, f)  
        A[i:f] = merge(B, C)  
    return A[i:f]
```



Mergesort

1.a. Demuestre que MergeSort es correcto

```
mergeSort(A, i, f):  
    if f - i > 1:  
        m = round((f - i)/2)  
        B = mergeSort(A, i, m)  
        C = mergeSort(A, m, f)  
        A[i:f] = merge(B, C)  
    return A[i:f]
```



Mergesort

1.a. Demuestre que MergeSort es correcto

```
mergeSort(A, i, f):  
    if f - i > 1:  
        m = round((f - i)/2)  
        B = mergeSort(A, i, m)  
        C = mergeSort(A, m, f)  
        A[i:f] = merge(B, C)  
    return A[i:f]
```

Caso Base:



Mergesort

1.a. Demuestre que MergeSort es correcto

```
mergeSort(A, i, f):  
    if f - i > 1:  
        m = round((f - i)/2)  
        B = mergeSort(A, i, m)  
        C = mergeSort(A, m, f)  
        A[i:f] = merge(B, C)  
    return A[i:f]
```

Caso Base:

Para A de largo $n = 1$, A está ordenado trivialmente.



Mergesort

1.a. Demuestre que MergeSort es correcto

```
mergeSort(A, i, f):  
    if f - i > 1:  
        m = round((f - i)/2)  
        B = mergeSort(A, i, m)  
        C = mergeSort(A, m, f)  
        A[i:f] = merge(B, C)  
    return A[i:f]
```

Caso Base:

Para A de largo $n = 1$, A está ordenado trivialmente.

Se retorna A y MergeSort es correcto.



Mergesort

1.a. Demuestre que MergeSort es correcto

```
mergeSort(A, i, f):  
    if f - i > 1:  
        m = round((f - i)/2)  
        B = mergeSort(A, i, m)  
        C = mergeSort(A, m, f)  
        A[i:f] = merge(B, C)  
    return A[i:f]
```

Hipótesis Inductiva:



Mergesort

1.a. Demuestre que MergeSort es correcto

```
mergeSort(A, i, f):  
    if f - i > 1:  
        m = round((f - i)/2)  
        B = mergeSort(A, i, m)  
        C = mergeSort(A, m, f)  
        A[i:f] = merge(B, C)  
    return A[i:f]
```

Hipótesis Inductiva:

Supongamos que MergeSort es
correcto para A de largo $k \leq n$



Mergesort

1.a. Demuestre que MergeSort es correcto

```
mergeSort(A, i, f):  
    if f - i > 1:  
        m = round((f - i)/2)  
        B = mergeSort(A, i, m)  
        C = mergeSort(A, m, f)  
        A[i:f] = merge(B, C)  
    return A[i:f]
```

Tesis Inductiva:



Mergesort

1.a. Demuestre que MergeSort es correcto

```
mergeSort(A, i, f):  
    if f - i > 1:  
        m = round((f - i)/2)  
        B = mergeSort(A, i, m)  
        C = mergeSort(A, m, f)  
        A[i:f] = merge(B, C)  
    return A[i:f]
```

Tesis Inductiva:

Tomamos A de largo $n + 1$.

Mergesort

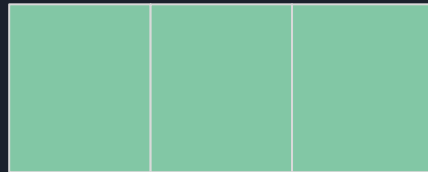
1.a. Demuestre que MergeSort es correcto

```
mergeSort(A, i, f):  
    if f - i > 1:  
        m = round((f - i)/2)  
        B = mergeSort(A, i, m)  
        C = mergeSort(A, m, f)  
        A[i:f] = merge(B, C)  
    return A[i:f]
```

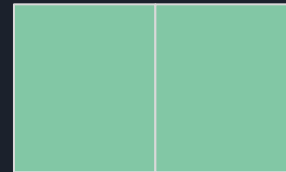
Tesis Inductiva:

Tomamos A de largo $n + 1$.

A:



...



Mergesort

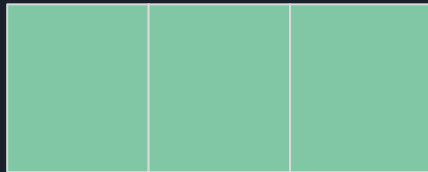
1.a. Demuestre que MergeSort es correcto

```
mergeSort(A, i, f):  
    if f - i > 1:  
        m = round((f - i)/2) ←  
        B = mergeSort(A, i, m)  
        C = mergeSort(A, m, f)  
        A[i:f] = merge(B, C)  
    return A[i:f]
```

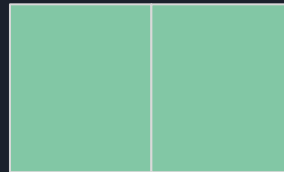
Tesis Inductiva:

Tomamos A de largo $n + 1$.

A:



...



Mergesort

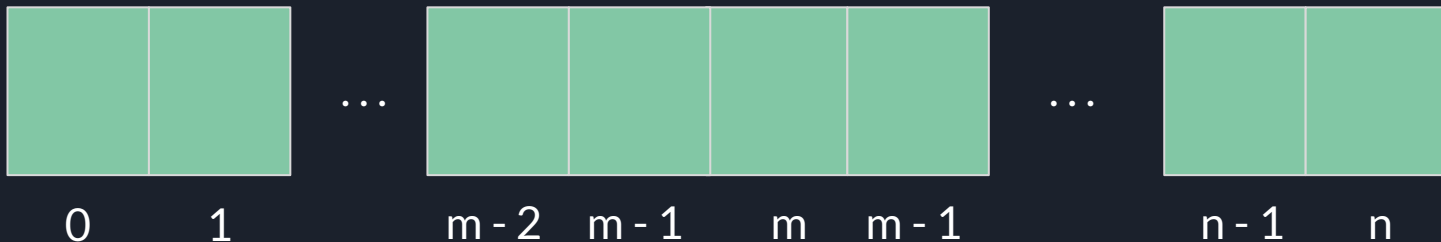
1.a. Demuestre que MergeSort es correcto

```
mergeSort(A, i, f):  
    if f - i > 1:  
        m = round((f - i)/2) ←  
        B = mergeSort(A, i, m)  
        C = mergeSort(A, m, f)  
        A[i:f] = merge(B, C)  
    return A[i:f]
```

Tesis Inductiva:

Tomamos A de largo $n + 1$.

A:



Mergesort

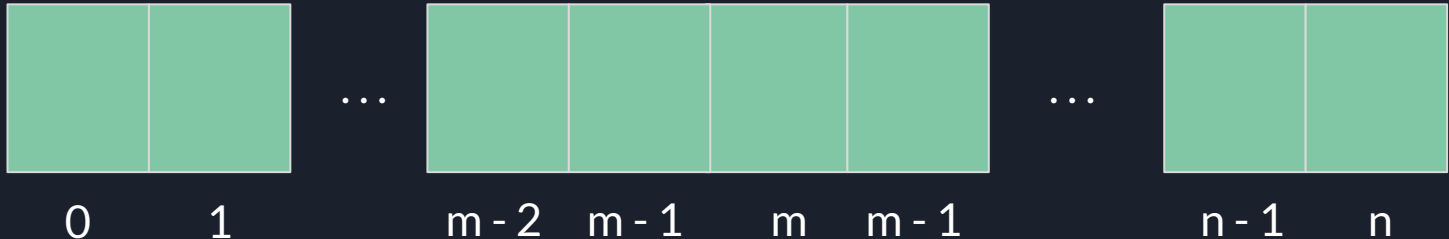
1.a. Demuestre que MergeSort es correcto

```
mergeSort(A, i, f):  
    if f - i > 1:  
        m = round((f - i)/2)  
        B = mergeSort(A, i, m) ←  
        C = mergeSort(A, m, f) ←  
        A[i:f] = merge(B, C)  
    return A[i:f]
```

Tesis Inductiva:

Se llama a MergeSort con las dos mitades de A

A:



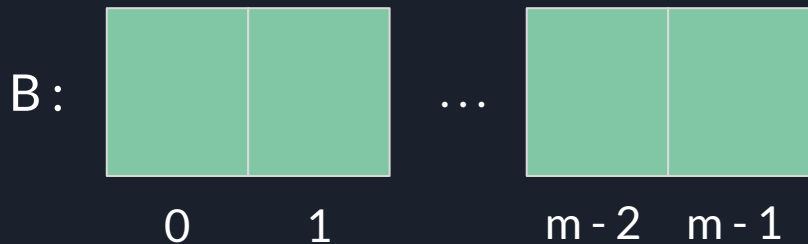
Mergesort

1.a. Demuestre que MergeSort es correcto

```
mergeSort(A, i, f):  
    if f - i > 1:  
        m = round((f - i)/2)  
        B = mergeSort(A, i, m) ←  
        C = mergeSort(A, m, f) ←  
        A[i:f] = merge(B, C)  
    return A[i:f]
```

Tesis Inductiva:

Se llama a MergeSort con las dos mitades de A



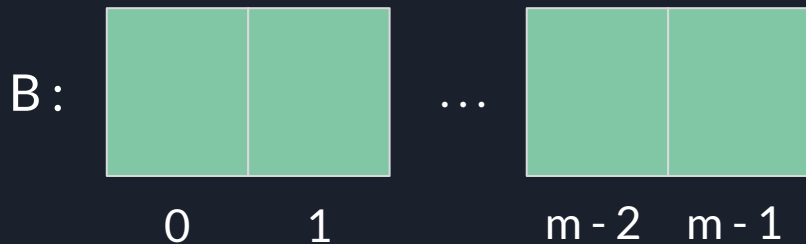
Mergesort

1.a. Demuestre que MergeSort es correcto

```
mergeSort(A, i, f):  
    if f - i > 1:  
        m = round((f - i)/2)  
        B = mergeSort(A, i, m) ←  
        C = mergeSort(A, m, f) ←  
        A[i:f] = merge(B, C)  
    return A[i:f]
```

Tesis Inductiva:

Los largos de B y C son $\leq n$.



Mergesort

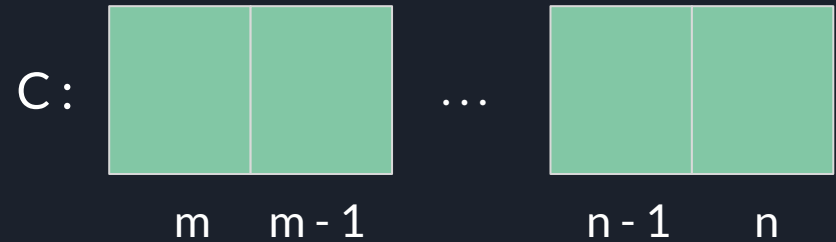
1.a. Demuestre que MergeSort es correcto

```
mergeSort(A, i, f):  
    if f - i > 1:  
        m = round((f - i)/2)  
        B = mergeSort(A, i, m) ←  
        C = mergeSort(A, m, f) ←  
        A[i:f] = merge(B, C)  
    return A[i:f]
```

Tesis Inductiva:

Los largos de B y C son $\leq n$.

Por HI, MergeSort las ordena correctamente.

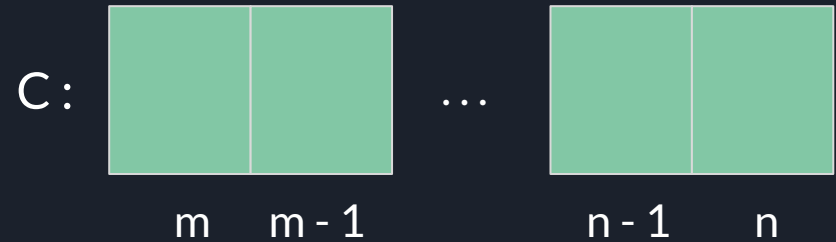


Mergesort

1.a. Demuestre que MergeSort es correcto

```
mergeSort(A, i, f):  
    if f - i > 1:  
        m = round((f - i)/2)  
        B = mergeSort(A, i, m)  
        C = mergeSort(A, m, f)  
        A[i:f] = merge(B, C) ←  
    return A[i:f]
```

Tesis Inductiva:



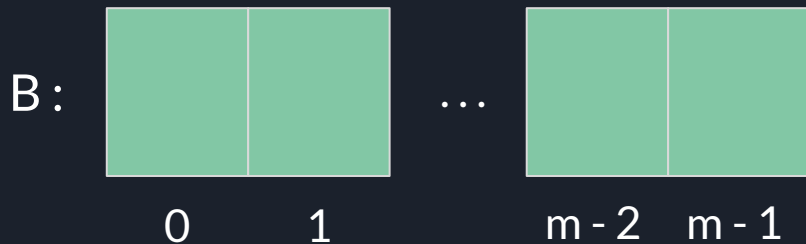
Mergesort

1.a. Demuestre que MergeSort es correcto

```
mergeSort(A, i, f):  
    if f - i > 1:  
        m = round((f - i)/2)  
        B = mergeSort(A, i, m)  
        C = mergeSort(A, m, f)  
        A[i:f] = merge(B, C) ←  
    return A[i:f]
```

Tesis Inductiva:

Merge une los arreglos ordenados B y C para volver a formar el arreglo A, ahora ordenado.



Mergesort

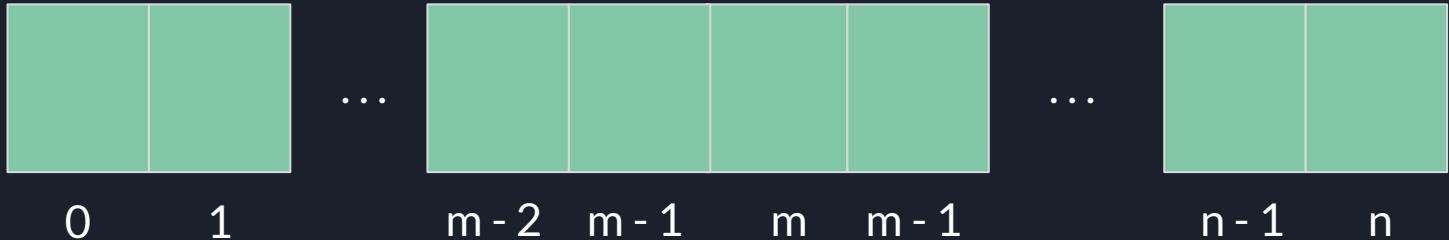
1.a. Demuestre que MergeSort es correcto

```
mergeSort(A, i, f):  
    if f - i > 1:  
        m = round((f - i)/2)  
        B = mergeSort(A, i, m)  
        C = mergeSort(A, m, f)  
        A[i:f] = merge(B, C) ←  
    return A[i:f]
```

Tesis Inductiva:

Merge une los arreglos ordenados B y C para volver a formar el arreglo A, ahora ordenado.

A:



Mergesort

1.a. Demuestre que MergeSort es correcto

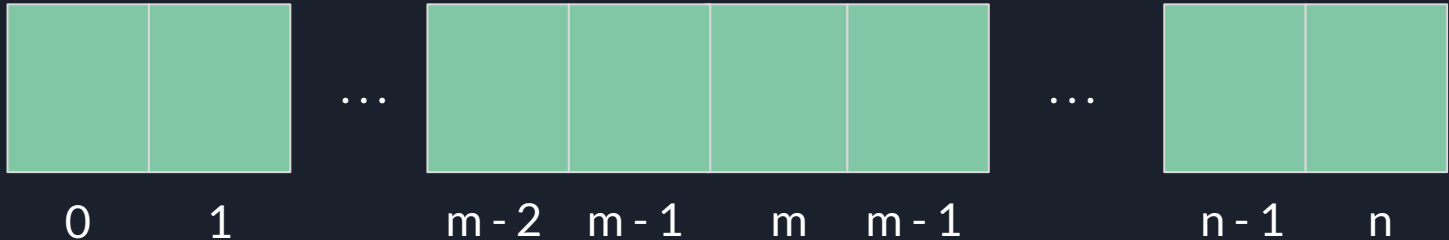
```
mergeSort(A, i, f):  
    if f - i > 1:  
        m = round((f - i)/2)  
        B = mergeSort(A, i, m)  
        C = mergeSort(A, m, f)  
        A[i:f] = merge(B, C) ←  
    return A[i:f]
```

Tesis Inductiva:

Merge une los arreglos ordenados B y C para volver a formar el arreglo A, ahora ordenado.

Por enunciado, Merge es correcto.

A:



Mergesort

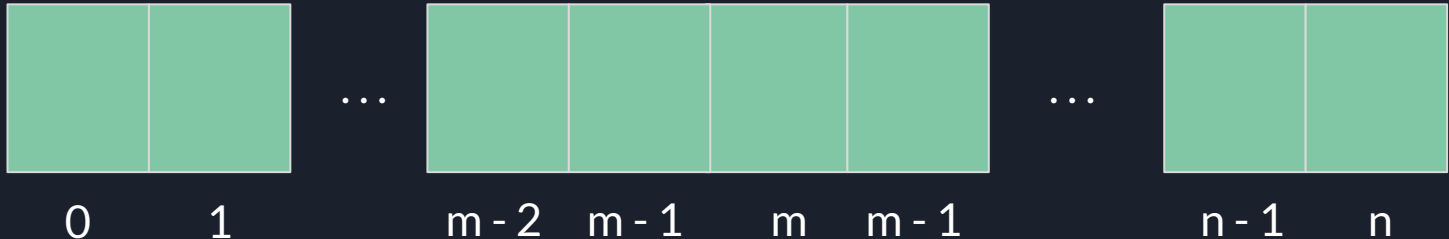
1.a. Demuestre que MergeSort es correcto

```
mergeSort(A, i, f):  
    if f - i > 1:  
        m = round((f - i)/2)  
        B = mergeSort(A, i, m)  
        C = mergeSort(A, m, f)  
        A[i:f] = merge(B, C)  
    return A[i:f] ←
```

Tesis Inductiva:

Por lo tanto, MergeSort puede ordenar correctamente un arreglo de largo $n + 1$

A:



Mergesort

1.a. Demuestre que MergeSort es correcto

```
mergeSort(A, i, f):  
    if f - i > 1:  
        m = round((f - i)/2)  
        B = mergeSort(A, i, m)  
        C = mergeSort(A, m, f)  
        A[i:f] = merge(B, C)  
    return A[i:f] ←
```

Tesis Inductiva:

Por lo tanto, MergeSort puede ordenar correctamente un arreglo de largo $n + 1$

MergeSort es correcto!

A:



Mergesort

1.a. Demuestre que MergeSort es correcto

```
mergeSort(A, i, f):  
    if f - i > 1:  
        m = round((f - i)/2)  
        B = mergeSort(A, i, m)  
        C = mergeSort(A, m, f)  
        A[i:f] = merge(B, C)  
    return A[i:f] ←
```

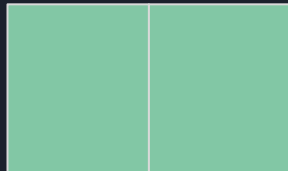
Tesis Inductiva:

Por lo tanto, MergeSort puede ordenar correctamente un arreglo de largo $n + 1$

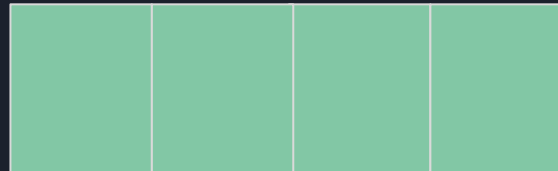
MergeSort es correcto!



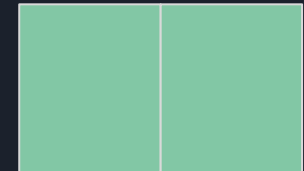
A:



...



...





Mergesort

1.b. Calcule la complejidad de MergeSort

```
mergeSort(A, i, f):  
    if f - i > 1:  
        m = round((f - i)/2)  
        B = mergeSort(A, i, m)  
        C = mergeSort(A, m, f)  
        A[i:f] = merge(B, C)  
    return A[i:f]
```




Mergesort

1.b. Calcule la complejidad de MergeSort

```
mergeSort(A, i, f):  
    if f - i > 1:  
        m = round((f - i)/2)  
        B = mergeSort(A, i, m)  
        C = mergeSort(A, m, f)  
        A[i:f] = merge(B, C)  
    return A[i:f]
```

- Notamos que si $n=1$, MergeSort corre en $O(1)$



Mergesort

1.b. Calcule la complejidad de MergeSort

```
mergeSort(A, i, f):  
    if f - i > 1:  
        m = round((f - i)/2)  
        B = mergeSort(A, i, m)  
        C = mergeSort(A, m, f)  
        A[i:f] = merge(B, C)  
    return A[i:f]
```

- Notamos que si $n=1$, MergeSort corre en $O(1)$
- Pero si $n > 1$...



Mergesort

1.b. Calcule la complejidad de MergeSort

```
mergeSort(A, i, f):  
    if f - i > 1:  
        m = round((f - i)/2)  
        B = mergeSort(A, i, m)  
        C = mergeSort(A, m, f)  
        A[i:f] = merge(B, C)  
    return A[i:f]
```

- Notamos que si $n=1$, MergeSort corre en $O(1)$
- Pero si $n > 1$...

Se llama nuevamente a MergeSort con nuevos inputs

Y también se llama a Merge



Mergesort

1.b. Calcule la complejidad de MergeSort

```
mergeSort(A, i, f):  
    if f - i > 1:  
        m = round((f - i)/2)  
        B = mergeSort(A, i, m)  
        C = mergeSort(A, m, f)  
        A[i:f] = merge(B, C)  
    return A[i:f]
```

$$T(n) = \begin{cases} 1 & \text{if } n = 1 \\ T\left(\lfloor \frac{n}{2} \rfloor\right) + T\left(\lceil \frac{n}{2} \rceil\right) + n & \text{if } n > 1 \end{cases}$$



Mergesort

1.b. Calcule la complejidad de MergeSort

Ecuación de Recurrencia:

$$T(n) = \begin{cases} 1 & \text{if } n = 1 \\ T\left(\lfloor \frac{n}{2} \rfloor\right) + T\left(\lceil \frac{n}{2} \rceil\right) + n & \text{if } n > 1 \end{cases}$$



Mergesort

1.b. Calcule la complejidad de MergeSort

Ecuación de Recurrencia:

$$T(n) = \begin{cases} 1 & \text{if } n = 1 \\ T\left(\lfloor \frac{n}{2} \rfloor\right) + T\left(\lceil \frac{n}{2} \rceil\right) + n & \text{if } n > 1 \end{cases}$$

Y... ahora qué?



Mergesort

1.b. Calcule la complejidad de MergeSort

Ecuación de Recurrencia:

$$T(n) = \begin{cases} 1 & \text{if } n = 1 \\ T\left(\lfloor \frac{n}{2} \rfloor\right) + T\left(\lceil \frac{n}{2} \rceil\right) + n & \text{if } n > 1 \end{cases}$$

- Tomamos un k tal que $n \leq 2^k < 2n$



Mergesort

1.b. Calcule la complejidad de MergeSort

Ecuación de Recurrencia:

$$T(n) = \begin{cases} 1 & \text{if } n = 1 \\ T\left(\lfloor \frac{n}{2} \rfloor\right) + T\left(\lceil \frac{n}{2} \rceil\right) + n & \text{if } n > 1 \end{cases}$$

- Tomamos un k tal que $n \leq 2^k < 2n$
- Tendremos entonces que $T(n) \leq T(2^k)$



Mergesort

1.b. Calcule la complejidad de MergeSort

Ecuación de Recurrencia:

$$T(n) = \begin{cases} 1 & \text{if } n = 1 \\ T\left(\lfloor \frac{n}{2} \rfloor\right) + T\left(\lceil \frac{n}{2} \rceil\right) + n & \text{if } n > 1 \end{cases}$$

- Tomamos un k tal que $n \leq 2^k < 2n$
- Tendremos entonces que $T(n) \leq T(2^k)$

Tenemos una nueva Ecuación de Recurrencia



Mergesort

1.b. Calcule la complejidad de MergeSort

$$T(n) \leq T(2^k) = \begin{cases} 1 & \text{if } k = 0 \\ 2^k + 2 \cdot T(2^{k-1}) & \text{if } k > 0 \end{cases}$$



Mergesort

1.b. Calcule la complejidad de MergeSort

$$T(n) \leq T(2^k) = \begin{cases} 1 & \text{if } k = 0 \\ 2^k + 2 \cdot T(2^{k-1}) & \text{if } k > 0 \end{cases}$$

$$T(n) \leq T(2^k) = 2^k + 2 * T(2^{k-1})$$



Mergesort

1.b. Calcule la complejidad de MergeSort

$$T(n) \leq T(2^k) = \begin{cases} 1 & \text{if } k = 0 \\ 2^k + 2 \cdot T(2^{k-1}) & \text{if } k > 0 \end{cases}$$

$$\begin{aligned} T(n) \leq T(2^k) &= 2^k + 2 * T(2^{k-1}) \\ &= 2^k + 2 * (2^{k-1} + 2 * T(2^{k-2})) \end{aligned}$$



Mergesort

1.b. Calcule la complejidad de MergeSort

$$T(n) \leq T(2^k) = \begin{cases} 1 & \text{if } k = 0 \\ 2^k + 2 \cdot T(2^{k-1}) & \text{if } k > 0 \end{cases}$$

$$\begin{aligned} T(n) \leq T(2^k) &= 2^k + 2 * T(2^{k-1}) \\ &= 2^k + (2^k + 4 * T(2^{k-2})) \end{aligned}$$



Mergesort

1.b. Calcule la complejidad de MergeSort

$$T(n) \leq T(2^k) = \begin{cases} 1 & \text{if } k = 0 \\ 2^k + 2 \cdot T(2^{k-1}) & \text{if } k > 0 \end{cases}$$

$$\begin{aligned} T(n) \leq T(2^k) &= 2^k + 2 * T(2^{k-1}) \\ &= 2^k + (2^k + 4 * T(2^{k-2})) \\ &= 2^k + 2^k + 4 * (2^{k-2} + 2 * T(2^{k-3})) \end{aligned}$$



Mergesort

1.b. Calcule la complejidad de MergeSort

$$T(n) \leq T(2^k) = \begin{cases} 1 & \text{if } k = 0 \\ 2^k + 2 \cdot T(2^{k-1}) & \text{if } k > 0 \end{cases}$$

$$\begin{aligned} T(n) \leq T(2^k) &= 2^k + 2 * T(2^{k-1}) \\ &= 2^k + (2^k + 4 * T(2^{k-2})) \\ &= 2^k + 2^k + (2^k + 8 * T(2^{k-3})) \end{aligned}$$



Mergesort

1.b. Calcule la complejidad de MergeSort

$$T(n) \leq T(2^k) = \begin{cases} 1 & \text{if } k = 0 \\ 2^k + 2 \cdot T(2^{k-1}) & \text{if } k > 0 \end{cases}$$

$$\begin{aligned} T(n) \leq T(2^k) &= 1 * 2^k + 2^1 * T(2^{k-1}) \\ &= 2 * 2^k + 2^2 * T(2^{k-2}) \\ &= 3 * 2^k + 2^3 * T(2^{k-3}) \end{aligned}$$



Mergesort

1.b. Calcule la complejidad de MergeSort

$$\begin{aligned}T(n) \leq T(2^k) &= 1 * 2^k + 2^1 * T(2^{k-1}) \\&= 2 * 2^k + 2^2 * T(2^{k-2}) \\&= 3 * 2^k + 2^3 * T(2^{k-3}) \\&= \dots\end{aligned}$$



Mergesort

1.b. Calcule la complejidad de MergeSort

$$T(n) \leq T(2^k) = 1 * 2^k + 2^1 * T(2^{k-1})$$

$$= 2 * 2^k + 2^2 * T(2^{k-2})$$

$$= 3 * 2^k + 2^3 * T(2^{k-3})$$

$$= \dots$$

$$= i * 2^k + 2^i * T(2^{k-i}) \quad \text{en la } i\text{-ésima iteración}$$



Mergesort

1.b. Calcule la complejidad de MergeSort

$$T(n) \leq T(2^k) = i * 2^k + 2^i * T(2^{k-i}) \quad \text{en la } i\text{-ésima iteración}$$



Mergesort

1.b. Calcule la complejidad de MergeSort

$$T(n) \leq T(2^k) = i * 2^k + 2^i * T(2^{k-i}) \quad \text{en la } i\text{-ésima iteración}$$

- Pero nuestro algoritmo no puede correr para siempre...



Mergesort

1.b. Calcule la complejidad de MergeSort

$$T(n) \leq T(2^k) = i * 2^k + 2^i * T(2^{k-i}) \quad \text{en la } i\text{-ésima iteración}$$

- Pero nuestro algoritmo no puede correr para siempre...
- Tomamos el caso $k = i$



Mergesort

1.b. Calcule la complejidad de MergeSort

$$T(n) \leq T(2^k) = i * 2^k + 2^i * T(2^{k-i}) \quad \text{en la } i\text{-ésima iteración}$$

- Pero nuestro algoritmo no puede correr para siempre...
- Tomamos el caso $k = i$

$$\begin{aligned} T(n) \leq T(2^k) &= i * 2^k + 2^i * T(2^{k-i}) \\ &= k * 2^k + 2^k * T(2^0) \end{aligned}$$



Mergesort

1.b. Calcule la complejidad de MergeSort

$$T(n) \leq T(2^k) = i * 2^k + 2^i * T(2^{k-i}) \quad \text{en la } i\text{-ésima iteración}$$

- Pero nuestro algoritmo no puede correr para siempre...
- Tomamos el caso $k = i$

$$\begin{aligned} T(n) \leq T(2^k) &= i * 2^k + 2^i * T(2^{k-i}) \\ &= k * 2^k + 2^k * T(1) \end{aligned}$$



Mergesort

1.b. Calcule la complejidad de MergeSort

$$T(n) \leq T(2^k) = i * 2^k + 2^i * T(2^{k-i}) \quad \text{en la } i\text{-ésima iteración}$$

- Pero nuestro algoritmo no puede correr para siempre...
- Tomamos el caso $k = i$

$$\begin{aligned} T(n) \leq T(2^k) &= i * 2^k + 2^i * T(2^{k-i}) \\ &= k * 2^k + 2^k \end{aligned}$$



Mergesort

1.b. Calcule la complejidad de MergeSort

Nos quedamos con $T(n) \leq k * 2^k + 2^k$



Mergesort

1.b. Calcule la complejidad de MergeSort

Nos quedamos con $T(n) \leq k * 2^k + 2^k$

Pero tenemos que volver a términos de n ... cómo?



Mergesort

1.b. Calcule la complejidad de MergeSort

Nos quedamos con $T(n) \leq k * 2^k + 2^k$

Pero tenemos que volver a términos de n ... cómo?

Recordemos que: $n \leq 2^k < 2n$



Mergesort

1.b. Calcule la complejidad de MergeSort

Nos quedamos con $T(n) \leq k * 2^k + 2^k$

Pero tenemos que volver a términos de n ... cómo?

Recordemos que: $n \leq 2^k < 2n$

$$\log(n) \leq k < \log(2n)$$



Mergesort

1.b. Calcule la complejidad de MergeSort

Recordemos que: $n \leq 2^k < 2n$

$$\log(n) \leq k < \log(2n)$$



Mergesort

1.b. Calcule la complejidad de MergeSort

Recordemos que: $n \leq 2^k < 2n$

$$\log(n) \leq k < \log(2n)$$

Ahora tenemos $T(n) \leq k * 2^k + 2^k < \log(2n) * n + n$



Mergesort

1.b. Calcule la complejidad de MergeSort

Como $T(n) < \log(2n) * n + n$

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

Mergesort

1.b. Calcule la complejidad de MergeSort

Como $T(n) < \log(2n) * n + n$

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





Quicksort

```
quicksort( $A, i, f$ ):
```

```
  if  $i \leq f$ :
```

```
     $p \leftarrow \textit{partition}(A, i, f)$ 
```

```
    quicksort( $A, i, p - 1$ )
```

```
    quicksort( $A, p + 1, f$ )
```



Quicksort

```
quicksort(A, i, f):
```

```
  if  $i \leq f$ :
```

```
     $p \leftarrow \textit{partition}(A, i, f)$ 
```

```
    quicksort(A, i,  $p - 1$ )
```

```
    quicksort(A,  $p + 1$ , f)
```

```
partition(A, i, f):
```

```
   $x \leftarrow$  un indice aleatorio en  $[i, f]$ ,  $p \leftarrow A[x]$ 
```

```
   $A[x] \rightleftharpoons A[f]$ 
```

```
   $j \leftarrow i$ 
```

```
  for  $k \in [i, f - 1]$ :
```

```
    if  $A[k] < p$ :
```

```
       $A[j] \rightleftharpoons A[k]$ 
```

```
       $j \leftarrow j + 1$ 
```

```
   $A[j] \rightleftharpoons A[f]$ 
```

```
  return j
```

Quicksort





Quicksort

```
quicksort( $A, i, f$ ):
```

```
  if  $i \leq f$ :
```

```
     $p \leftarrow \textit{partition}(A, i, f)$ 
```

```
    quicksort( $A, i, p - 1$ )
```

```
    quicksort( $A, p + 1, f$ )
```



Quicksort

partition(A, i, f):

$x \leftarrow$ un indice aleatorio en $[i, f]$, $p \leftarrow A[x]$

$A[x] \rightleftharpoons A[f]$

$j \leftarrow i$

for $k \in [i, f - 1]$:

if $A[k] < p$:

$A[j] \rightleftharpoons A[k]$

$j \leftarrow j + 1$

$A[j] \rightleftharpoons A[f]$

return j



Pregunta 2

El algoritmo de ordenamiento QuickSort no se comporta bien cuando existen muchos datos repetidos en el arreglo a ordenar.

- a) ¿Por qué sucede esto?
- b) Proponga una modificación al algoritmo visto en clase que permita mejorar esta situación.



Pregunta 2

```
partition(A, i, f):  
     $x \leftarrow \text{índice aleatorio} \in [i, f]$   
     $A[x] \rightleftharpoons A[f]$   
     $j \leftarrow i$   
    for  $k$  in  $(i, f - i)$ :  
        if  $A[j] < p$ :  
             $A[k] \rightleftharpoons A[j]$   
             $j \leftarrow j + 1$   
     $A[j] \rightleftharpoons A[f]$   
    return  $i + j$ 
```



Pregunta 2

partition(A, i, f):

$x \leftarrow \text{índice aleatorio} \in [i, f]$

$A[x] \rightleftharpoons A[f]$

$j \leftarrow i$

for k in $(i, f - i)$:

if $A[j] < p$:

$A[k] \rightleftharpoons A[j]$

$j \leftarrow j + 1$

$A[j] \rightleftharpoons A[f]$

return $i + j$

3_wide_partition(A, i, f):

$p \leftarrow \text{pivote aleatorio en } A$

$j, k \leftarrow i$

$u \leftarrow f$

while $k \leq u$:

if $A[k] < p$:

$A[k] \rightleftharpoons A[j]$

$j \leftarrow j + 1$

$k \leftarrow k + 1$

else if $A[k] > p$:

$A[k] \rightleftharpoons A[u]$

$u \leftarrow u - 1$

else: *// $p = A[k]$*

$k \leftarrow k + 1$

return $i + j, i + u$



Propuesto

- a) Complejidad en el mejor caso.
- b) Complejidad en el peor caso.
- c) Comentarios sobre si mejora QuickSort (Se les subirá solución)

Enfoque en sacar complejidades de partition y median.

La demostración de la complejidad de las recursiones de QuickSort serán vistas la clase siguiente.



Propuesto

```
median_modificado(A):  
    i ← 0  
    f ← n - 1  
    x ← partition(A, i, f)  
    while x ≠ n/2:  
        if x < n/2:  
            i ← x + 1  
        else:  
            f ← x - 1  
        x ← partition(A, i, f)  
    return x    // modificación
```



Propuesto: Mejor caso

- Median encuentra en el primer intento la mediana $\Rightarrow O(1)$
- Partition recorre n elementos.

$$\text{mejor caso} \Rightarrow \text{median} \sim 1 \cdot O(n) \sim O(n)$$



Propuesto: Peor caso

- Si llamamos a *median*, en el peor caso recorre los n elementos antes de encontrar la mediana.
 $\Rightarrow O(n)$
- Cada llamada a *median* llama a *partition*, que siempre recorre cerca de n elementos.

$$\text{mejor caso} \Rightarrow \text{median} \sim n \cdot O(n) \sim O(n^2)$$



Propuesto

Para encontrar la complejidad de este QuickSort novedoso que propuso el compañero del ejercicio propuesto, les servirá la clase siguiente, donde deberían ver cómo se llega a la complejidad de QuickSort.

Se les subirá la solución del propuesto el lunes :)