Merge Sort

Clase 03

IIC 2133 - Sección 3

Prof. Eduardo Bustos

Sumario

Complejidad de algoritmos de ordenación

Resumimos los resultados de complejidad por caso hasta el momento

Algoritmo	Mejor caso	Caso promedio	Peor caso	Memoria adicional
Selection Sort	?	?	?	$\mathcal{O}(1)$
Insertion Sort	$\mathcal{O}(n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(1)$
Merge Sort	?	?	?	?
Quick Sort	?	?	?	?
Heap Sort	?	?	?	?

Complejidad de algoritmos de ordenación

Resumimos los resultados de complejidad por caso hasta el momento

Algoritmo	Mejor caso	Caso promedio	Peor caso	Memoria adicional
Selection Sort	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(1)$
Insertion Sort	$\mathcal{O}(n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(1)$
Merge Sort	?	?	?	?
Quick Sort	?	?	?	?
Heap Sort	?	?	?	?

SelectionSort in place

```
input  : Secuencia $A[0 \dots n-1]$, largo $n \geq 2$
output: $\varnothing$

SelectionSort $(A, n)$:
    for $i = 0 \dots n-2$ :
        $min \leftarrow i$
        for $j = i+1 \dots n-1$ :
            if $A[j] < A[min]$ :
                $min \leftarrow j$
        $A[i] \leftrightharpoons A[min]$
```

SelectionSort e InsertionSort in place

```
SelectionSort $(A, n)$:
    for $i = 0 \dots n-2$ :
        $min \leftarrow i$
        for $j = i+1 \dots n-1$ :
            if $A[j] < A[min]$ :
                $min \leftarrow j$
        $A[i] \leftrightharpoons A[min]$
```

```
InsertionSort $(A, n)$:
    for $i = 1 \dots n-1$ :
        $j \leftarrow i$
        while $(j > 0) \wedge (A[j] < A[j-1])$ :
            $A[j] \leftrightharpoons A[j-1]$
            $j \leftarrow j-1$
```

Ordenación hasta ahora

`SelectionSort`

Ordenación hasta ahora

`SelectionSort`

- No tiene un mejor caso que sea *mejor* que su peor caso: $\mathcal{O}(n^2)$

Ordenación hasta ahora

`SelectionSort`

- No tiene un mejor caso que sea *mejor* que su peor caso: $\mathcal{O}(n^2)$
- Siempre revisa la secuencia completa para determinar el mínimo

Ordenación hasta ahora

`SelectionSort`

- No tiene un mejor caso que sea *mejor* que su peor caso: $\mathcal{O}(n^2)$
- Siempre revisa la secuencia completa para determinar el mínimo

`InsertionSort`

Ordenación hasta ahora

`SelectionSort`

- No tiene un mejor caso que sea *mejor* que su peor caso: $\mathcal{O}(n^2)$
- Siempre revisa la secuencia completa para determinar el mínimo

`InsertionSort`

- Cuando la secuencia está ordenada toma $\mathcal{O}(n)$

Ordenación hasta ahora

`SelectionSort`

- No tiene un mejor caso que sea *mejor* que su peor caso: $\mathcal{O}(n^2)$
- Siempre revisa la secuencia completa para determinar el mínimo

`InsertionSort`

- Cuando la secuencia está ordenada toma $\mathcal{O}(n)$
- En el caso promedio es $\mathcal{O}(n^2)$, tomando el promedio sobre todas las permutaciones igualmente probables

Ordenación hasta ahora

`SelectionSort`

- No tiene un mejor caso que sea *mejor* que su peor caso: $\mathcal{O}(n^2)$
- Siempre revisa la secuencia completa para determinar el mínimo

`InsertionSort`

- Cuando la secuencia está ordenada toma $\mathcal{O}(n)$
- En el caso promedio es $\mathcal{O}(n^2)$, tomando el promedio sobre todas las permutaciones igualmente probables
- Argumentamos esto mediante conteo de inversiones en cada permutación

Ordenación hasta ahora

`SelectionSort`

- No tiene un mejor caso que sea *mejor* que su peor caso: $\mathcal{O}(n^2)$
- Siempre revisa la secuencia completa para determinar el mínimo

`InsertionSort`

- Cuando la secuencia está ordenada toma $\mathcal{O}(n)$
- En el caso promedio es $\mathcal{O}(n^2)$, tomando el promedio sobre todas las permutaciones igualmente probables
- Argumentamos esto mediante conteo de inversiones en cada permutación

¿Podemos tener un algoritmo de ordenación
con mejor complejidad que $\mathcal{O}(n^2)$ en el peor caso?

Hay esperanza: corregir varias inversiones a la vez

Ejemplo

Recordemos que el siguiente arreglo tiene 9 inversiones

34	8	64	51	32	21
0	1	2	3	4	5

Inversiones (definición más general)

Definición

Sea A un arreglo de n números y sean $0 \leq i, j \leq n - 1$ índices del arreglo. Si $i < j$ y $A[i] > A[j]$, entonces diremos que el par ordenado (i, j) es una **inversión**.

Ejemplo

El arreglo

$A =$

34	8	64	51	32	21
0	1	2	3	4	5

tiene 9 inversiones:

- $(0, 1)$, $(0, 4)$, $(0, 5)$
- $(2, 3)$, $(2, 4)$, $(2, 5)$
- $(3, 4)$, $(3, 5)$
- $(4, 5)$

Hay esperanza: corregir varias inversiones a la vez

Ejemplo

Recordemos que el siguiente arreglo tiene 9 inversiones

34	8	64	51	32	21
0	1	2	3	4	5

Si intercambiamos 34 y 8:

8	34	64	51	32	21
0	1	2	3	4	5

- Corregimos $(0, 1)$

Hay esperanza: corregir varias inversiones a la vez

Ejemplo

Recordemos que el siguiente arreglo tiene 9 inversiones

34	8	64	51	32	21
0	1	2	3	4	5

Si intercambiamos 34 y 8:

8	34	64	51	32	21
0	1	2	3	4	5

- Corregimos $(0, 1)$

Si intercambiamos 34 y 21:

21	8	64	51	32	34
0	1	2	3	4	5

- Corregimos $(0, 4)$, $(0, 5)$, $(4, 5)$

Un escenario relacionado

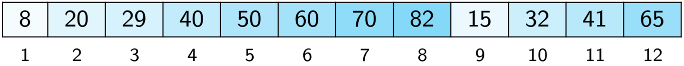
Un escenario relacionado

Consideremos una secuencia parcialmente ordenada

Un escenario relacionado

Consideremos una secuencia parcialmente ordenada

Para ser más precisos, una secuencia que está formada por dos sub-secuencias ordenadas

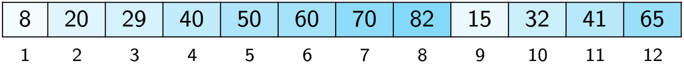


8	20	29	40	50	60	70	82	15	32	41	65
1	2	3	4	5	6	7	8	9	10	11	12

Un escenario relacionado

Consideremos una secuencia parcialmente ordenada

Para ser más precisos, una secuencia que está formada por dos sub-secuencias ordenadas



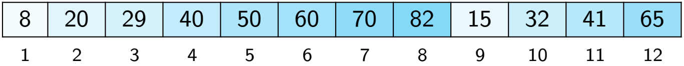
8	20	29	40	50	60	70	82	15	32	41	65
1	2	3	4	5	6	7	8	9	10	11	12

Además, sabemos exactamente dónde comienza la segunda sub-secuencia ordenada

Un escenario relacionado

Consideremos una secuencia parcialmente ordenada

Para ser más precisos, una secuencia que está formada por dos sub-secuencias ordenadas



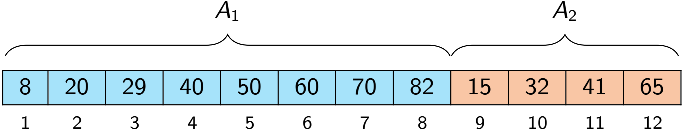
8	20	29	40	50	60	70	82	15	32	41	65
1	2	3	4	5	6	7	8	9	10	11	12

Además, sabemos exactamente dónde comienza la segunda sub-secuencia ordenada

¿Cómo aprovechamos este hecho para ordenar la secuencia completa?

Primer intento: `InsertionSort`

Estado inicial

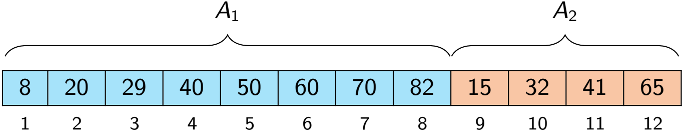


8	20	29	40	50	60	70	82	15	32	41	65
1	2	3	4	5	6	7	8	9	10	11	12

que haría Insertion sort en A1?
cuál sería su complejidad?
que haría en A1+A2?
cual sería su complejidad?

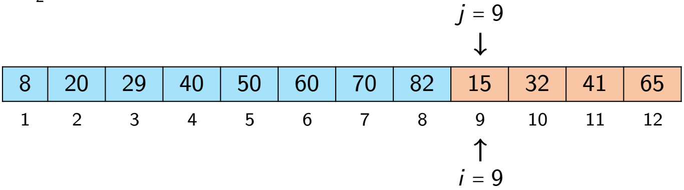
Primer intento: `InsertionSort`

Estado inicial



8	20	29	40	50	60	70	82	15	32	41	65
1	2	3	4	5	6	7	8	9	10	11	12

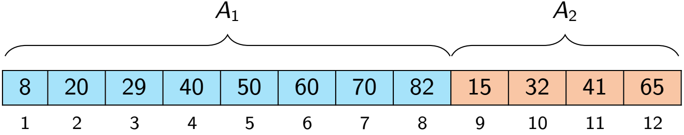
`InsertionSort` no intercambia nada del tramo A_1 y los índices i, j llegan al tramo A_2



8	20	29	40	50	60	70	82	15	32	41	65
1	2	3	4	5	6	7	8	9	10	11	12

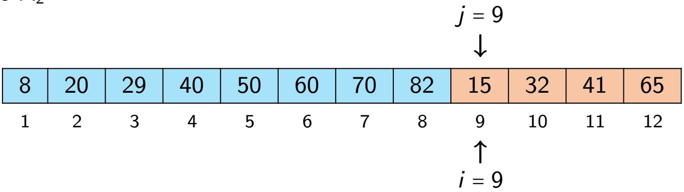
Primer intento: `InsertionSort`

Estado inicial



8	20	29	40	50	60	70	82	15	32	41	65
1	2	3	4	5	6	7	8	9	10	11	12

`InsertionSort` no intercambia nada del tramo A_1 y los índices i, j llegan al tramo A_2



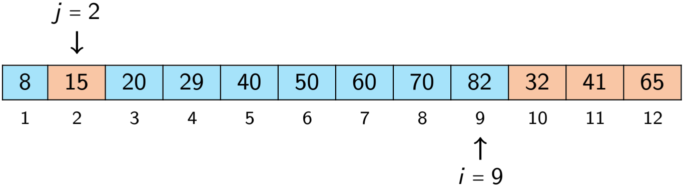
8	20	29	40	50	60	70	82	15	32	41	65
1	2	3	4	5	6	7	8	9	10	11	12

Hasta este punto la ejecución es $\mathcal{O}(n)$

veamos ahora cuando seguimos con A2

Primer intento: `InsertionSort`

El valor 15 se va intercambiando hasta llegar a su posición final $j = 2$



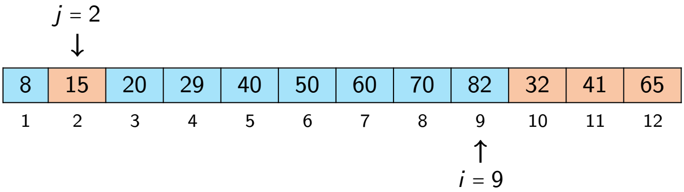
$j = 2$

8	15	20	29	40	50	60	70	82	32	41	65
1	2	3	4	5	6	7	8	9	10	11	12

$i = 9$

Primer intento: `InsertionSort`

El valor 15 se va intercambiando hasta llegar a su posición final $j = 2$

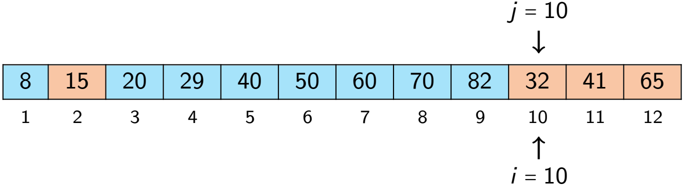


$j = 2$ ↓

8	15	20	29	40	50	60	70	82	32	41	65
1	2	3	4	5	6	7	8	9	10	11	12

↑ $i = 9$

En la siguiente iteración,



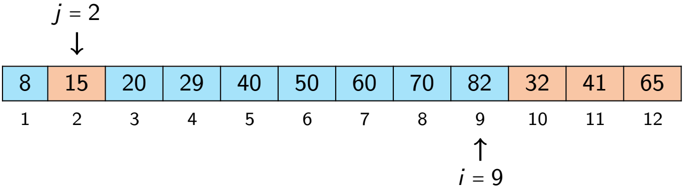
$j = 10$ ↓

8	15	20	29	40	50	60	70	82	32	41	65
1	2	3	4	5	6	7	8	9	10	11	12

↑ $i = 10$

Primer intento: `InsertionSort`

El valor 15 se va intercambiando hasta llegar a su posición final $j = 2$

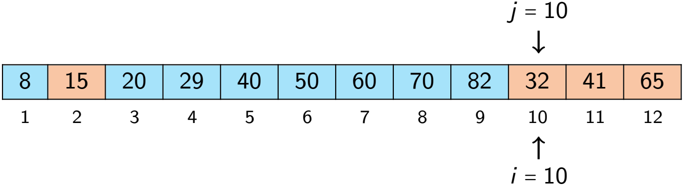


$j = 2$ ↓

8	15	20	29	40	50	60	70	82	32	41	65
1	2	3	4	5	6	7	8	9	10	11	12

↑ $i = 9$

En la siguiente iteración,



$j = 10$ ↓

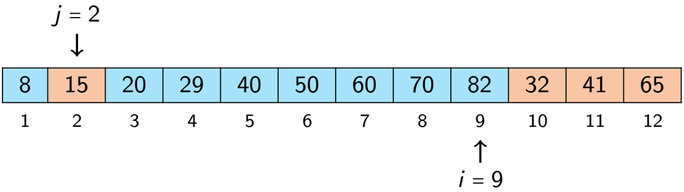
8	15	20	29	40	50	60	70	82	32	41	65
1	2	3	4	5	6	7	8	9	10	11	12

↑ $i = 10$

Conclusión: en este tramo el algoritmo vuelve a ser $\mathcal{O}(n^2)$

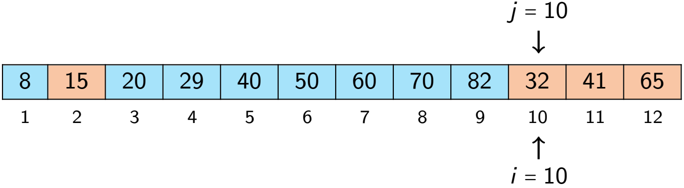
Primer intento: `InsertionSort`

El valor 15 se va intercambiando hasta llegar a su posición final $j = 2$



8	15	20	29	40	50	60	70	82	32	41	65
1	2	3	4	5	6	7	8	9	10	11	12

En la siguiente iteración,



8	15	20	29	40	50	60	70	82	32	41	65
1	2	3	4	5	6	7	8	9	10	11	12

Conclusión: en este tramo el algoritmo vuelve a ser $\mathcal{O}(n^2)$

Hoy veremos una mejor estrategia para aprovechar el orden

Objetivos de la clase

Objetivos de la clase

☐ Comprender el algoritmo `Merge` para combinar secuencias ordenadas

Objetivos de la clase

- ☐ Comprender el algoritmo `Merge` para combinar secuencias ordenadas
- ☐ Determinar complejidad de `Merge` y el *trade off* de ejecutarlo *in place*

Objetivos de la clase

☐ Comprender el algoritmo `Merge` para combinar secuencias ordenadas

☐ Determinar complejidad de `Merge` y el *trade off* de ejecutarlo *in place*

☐ Demostrar correctitud de `Merge`

Objetivos de la clase

☐ Comprender el algoritmo `Merge` para combinar secuencias ordenadas

☐ Determinar complejidad de `Merge` y el *trade off* de ejecutarlo *in place*

☐ Demostrar correctitud de `Merge`

☐ Comprender el uso de `Merge` como algoritmo de ordenación general en `MergeSort`

Objetivos de la clase

- ☐ Comprender el algoritmo `Merge` para combinar secuencias ordenadas
- ☐ Determinar complejidad de `Merge` y el *trade off* de ejecutarlo *in place*
- ☐ Demostrar correctitud de `Merge`
- ☐ Comprender el uso de `Merge` como algoritmo de ordenación general en `MergeSort`
- ☐ Determinar la complejidad de `MergeSort`

Sumario

Mezcla (*merge*) de secuencias ordenadas

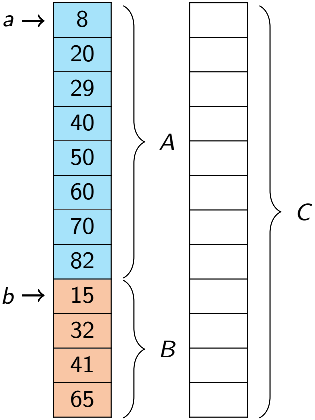
Proponemos el siguiente algoritmo para combinar dos secuencias ordenadas para formar una nueva **ordenada**

Mezcla (*merge*) de secuencias ordenadas

Proponemos el siguiente algoritmo para combinar dos secuencias ordenadas para formar una nueva **ordenada**

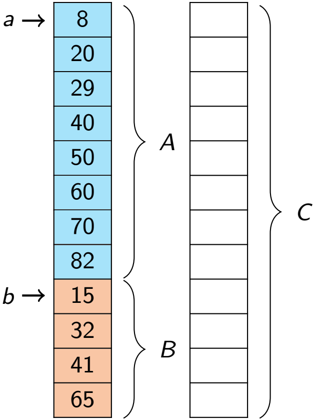
```
input  : Secuencias ordenadas $A$ y $B$
output: Nueva secuencia ordenada $C$

Merge($A, B$):
1      Iniciamos $C$ vacía
2      Sean $a$ y $b$ los primeros elementos de $A$ y $B$
3      Extraer de su secuencia respectiva el menor entre $a$ y $b$
4      Insertar el elemento extraído al final de $C$
5      Si quedan elementos en $A$ y $B$, volver a línea 2
6      Concatenar $C$ con la secuencia que aún tenga elementos
7      return $C$
```

`Merge`: Ejemplo de ejecución

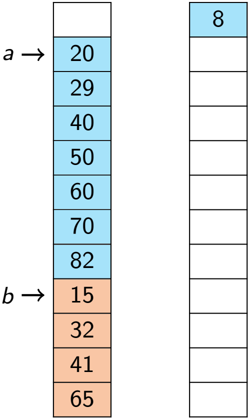


Estado inicial

`Merge`: Ejemplo de ejecución

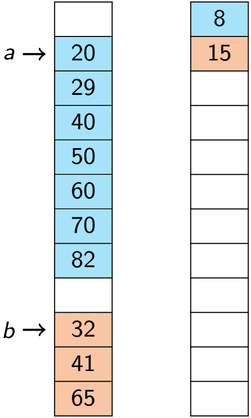


Estado inicial



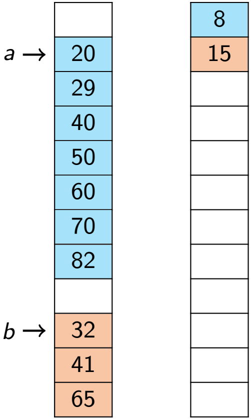
Estado luego de la primera iteración

`Merge`: Ejemplo de ejecución

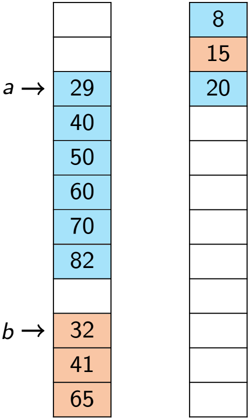


Estado luego de la
segunda iteración

`Merge`: Ejemplo de ejecución

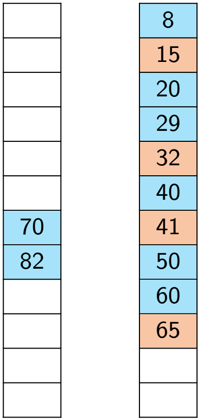


Estado luego de la
segunda iteración



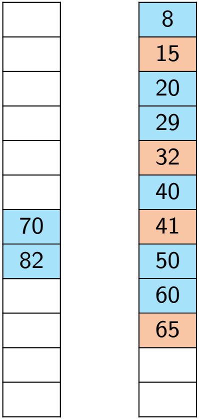
Estado luego de la
tercera iteración

`Merge`: Ejemplo de ejecución

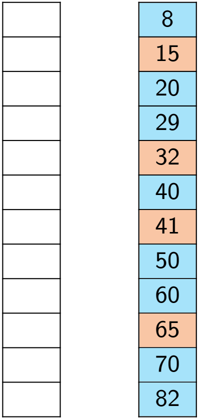


Estado luego de insertar en C
el último elemento de B

`Merge`: Ejemplo de ejecución



Estado luego de insertar en C
el último elemento de B



Estado luego de
concatenar el resto de A

Correctitud de `Merge`

Demostración (finitud)

Correctitud de `Merge`

Demostración (finitud)

En cada iteración del algoritmo antes de ejecutar la línea 6, se extrae siempre un elemento de A o B, y se inserta en C.

Correctitud de `Merge`

Demostración (finitud)

En cada iteración del algoritmo antes de ejecutar la línea 6, se extrae siempre un elemento de A o B, y se inserta en C.

Luego, cuando una de las secuencias se vacía, se insertan todos sus elementos en C.

Correctitud de `Merge`

Demostración (finitud)

En cada iteración del algoritmo antes de ejecutar la línea 6, se extrae siempre un elemento de A o B, y se inserta en C.

Luego, cuando una de las secuencias se vacía, se insertan todos sus elementos en C.

En total se realizan $n = |A| + |B|$ inserciones y un número menor a n de comparaciones entre elementos. Luego, el algoritmo termina en una cantidad finita de pasos. □

Correctitud de `Merge`

Demostración (propósito)

Para A, B inicialmente ordenadas, consideremos la propiedad

$$\mathbf{P}(\mathbf{n}) \quad :=$$

Correctitud de `Merge`

Demostración (propósito)

Para A, B inicialmente ordenadas, consideremos la propiedad

$$\mathbf{P}(\mathbf{n}) \quad := \quad \text{Luego de insertar el } n\text{-ésimo elemento en } C, \; A, B, C \text{ se encuentran ordenadas}$$

1. **Caso base.**

Correctitud de `Merge`

Demostración (propósito)

Para A, B inicialmente ordenadas, consideremos la propiedad

$$\mathbf{P(n)} \quad := \quad \begin{array}{l}\text{Luego de insertar el } n\text{-ésimo elemento en } C,\\ A, B, C \text{ se encuentran ordenadas}\end{array}$$

1. **Caso base.** $\mathbf{P(1)}$ corresponde al estado de las secuencia luego de insertar el primer elemento en C.

Correctitud de `Merge`

Demostración (propósito)

Para A, B inicialmente ordenadas, consideremos la propiedad

$$\mathbf{P}(\mathbf{n}) \quad := \quad \text{Luego de insertar el } n\text{-ésimo elemento en } C, \\ A, B, C \text{ se encuentran ordenadas}$$

1. **Caso base.** $\mathbf{P}(\mathbf{1})$ corresponde al estado de las secuencia luego de insertar el primer elemento en C.
    - Dado que se extrajo el menor elemento de alguna de las otras secuencias, estas se mantienen ordenadas. Esto aplica trivialmente si dicha secuencia queda vacía.

Correctitud de `Merge`

Demostración (propósito)

Para A, B inicialmente ordenadas, consideremos la propiedad

$$\mathbf{P(n)} \quad := \quad \begin{array}{l}\text{Luego de insertar el } n\text{-ésimo elemento en } C, \\ A, B, C \text{ se encuentran ordenadas}\end{array}$$

1. **Caso base.** $\mathbf{P(1)}$ corresponde al estado de las secuencia luego de insertar el primer elemento en C.
    - Dado que se extrajo el menor elemento de alguna de las otras secuencias, estas se mantienen ordenadas. Esto aplica trivialmente si dicha secuencia queda vacía.
    - Dado que C solo tiene un elemento, está ordenada.

Correctitud de `Merge`

Demostración (propósito)

$$\mathbf{P(n)} \quad := \quad \text{Luego de insertar el } n\text{-ésimo elemento en } C, \; A, B, C \text{ se encuentran ordenadas}$$

2. **H.I.** Suponemos que luego de agregar el n-ésimo elemento, A, B, C están ordenadas.

Correctitud de `Merge`

Demostración (propósito)

$$\mathbf{P(n)} \quad := \quad \text{Luego de insertar el } n\text{-ésimo elemento en } C, \; A, B, C \text{ se encuentran ordenadas}$$

2. **H.I.** Suponemos que luego de agregar el n-ésimo elemento, A, B, C están ordenadas.

   P.D. Luego de agregar el $(n+1)$-ésimo elemento, A, B, C siguen ordenadas.

Correctitud de `Merge`

Demostración (propósito)

$$\mathbf{P(n)} \quad := \quad \text{Luego de insertar el } n\text{-ésimo elemento en } C, \; A, B, C \text{ se encuentran ordenadas}$$

2. **H.I.** Suponemos que luego de agregar el n-ésimo elemento, A, B, C están ordenadas.

   P.D. Luego de agregar el $(n+1)$-ésimo elemento, A, B, C siguen ordenadas.

   Tenemos dos casos

Correctitud de `Merge`

Demostración (propósito)

$$\mathbf{P(n)} \quad := \quad \text{Luego de insertar el } n\text{-ésimo elemento en } C, \; A, B, C \text{ se encuentran ordenadas}$$

2. **H.I.** Suponemos que luego de agregar el n-ésimo elemento, A, B, C están ordenadas.

   P.D. Luego de agregar el $(n+1)$-ésimo elemento, A, B, C siguen ordenadas.

   Tenemos dos casos

   - Si quedan elementos en A y en B, sea c_{n+1} el menor entre las cabezas de A y B.

Correctitud de `Merge`

Demostración (propósito)

$$\mathbf{P(n)} \quad := \quad \text{Luego de insertar el } n\text{-ésimo elemento en } C, \\ A, B, C \text{ se encuentran ordenadas}$$

2. **H.I.** Suponemos que luego de agregar el n-ésimo elemento, A, B, C están ordenadas.

   P.D. Luego de agregar el $(n + 1)$-ésimo elemento, A, B, C siguen ordenadas.

   Tenemos dos casos

   - Si quedan elementos en A y en B, sea c_{n+1} el menor entre las cabezas de A y B.
   - Sin pérdida de generalidad, si solo quedan elementos en A, sea c_{n+1} la cabeza de A.

Correctitud de `Merge`

Demostración (propósito)

$$\mathbf{P(n)} \quad := \quad \begin{array}{l} \text{Luego de insertar el } n\text{-ésimo elemento en } C, \\ A, B, C \text{ se encuentran ordenadas} \end{array}$$

2. **H.I.** Suponemos que luego de agregar el n-ésimo elemento, A, B, C están ordenadas.

   P.D. Luego de agregar el $(n+1)$-ésimo elemento, A, B, C siguen ordenadas.

   Tenemos dos casos

   - Si quedan elementos en A y en B, sea c_{n+1} el menor entre las cabezas de A y B.
   - Sin pérdida de generalidad, si solo quedan elementos en A, sea c_{n+1} la cabeza de A.

   Se elimina c_{n+1} de su secuencia respectiva y se inserta al final de C.

Correctitud de `Merge`

Demostración (propósito)

Por **H.I.** tenemos que la secuencia de origen de c_{n+1} se encontraba ordenada antes de sacarlo. Como es el mínimo de la secuencia por ser el primer elemento y ser una secuencia ordenada, se preserva el orden. Si la secuencia se vacía, también está ordenada.

Correctitud de `Merge`

Demostración (propósito)

Por **H.I.** tenemos que la secuencia de origen de c_{n+1} se encontraba ordenada antes de sacarlo. Como es el mínimo de la secuencia por ser el primer elemento y ser una secuencia ordenada, se preserva el orden. Si la secuencia se vacía, también está ordenada.

Por **H.I.** tenemos que los primeros n elementos de C cumplen

$$c_1 \leq \cdots \leq c_n$$

Correctitud de `Merge`

Demostración (propósito)

Por **H.I.** tenemos que la secuencia de origen de c_{n+1} se encontraba ordenada antes de sacarlo. Como es el mínimo de la secuencia por ser el primer elemento y ser una secuencia ordenada, se preserva el orden. Si la secuencia se vacía, también está ordenada.

Por **H.I.** tenemos que los primeros n elementos de C cumplen

$$c_1 \leq \cdots \leq c_n$$

Si c_{n+1} fuera estrictamente menor a alguno de estos elementos, implicaría una de las siguientes contradicciones

Correctitud de `Merge`

Demostración (propósito)

Por **H.I.** tenemos que la secuencia de origen de c_{n+1} se encontraba ordenada antes de sacarlo. Como es el mínimo de la secuencia por ser el primer elemento y ser una secuencia ordenada, se preserva el orden. Si la secuencia se vacía, también está ordenada.

Por **H.I.** tenemos que los primeros n elementos de C cumplen

$$c_1 \leq \cdots \leq c_n$$

Si c_{n+1} fuera estrictamente menor a alguno de estos elementos, implicaría una de las siguientes contradicciones

- A o B no están ordenadas (ya probamos que lo están)

Correctitud de `Merge`

Demostración (propósito)

Por **H.I.** tenemos que la secuencia de origen de c_{n+1} se encontraba ordenada antes de sacarlo. Como es el mínimo de la secuencia por ser el primer elemento y ser una secuencia ordenada, se preserva el orden. Si la secuencia se vacía, también está ordenada.

Por **H.I.** tenemos que los primeros n elementos de C cumplen

$$c_1 \leq \cdots \leq c_n$$

Si c_{n+1} fuera estrictamente menor a alguno de estos elementos, implicaría una de las siguientes contradicciones

- A o B no están ordenadas (ya probamos que lo están)
- c_{n+1} es extraído en una iteración anterior por el criterio de selección

Correctitud de `Merge`

Demostración (propósito)

Por **H.I.** tenemos que la secuencia de origen de c_{n+1} se encontraba ordenada antes de sacarlo. Como es el mínimo de la secuencia por ser el primer elemento y ser una secuencia ordenada, se preserva el orden. Si la secuencia se vacía, también está ordenada.

Por **H.I.** tenemos que los primeros n elementos de C cumplen

$$c_1 \leq \cdots \leq c_n$$

Si c_{n+1} fuera estrictamente menor a alguno de estos elementos, implicaría una de las siguientes contradicciones

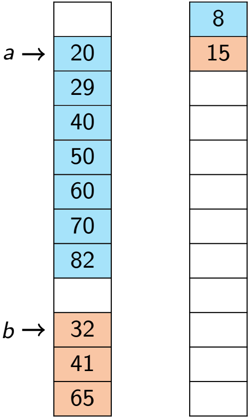
- A o B no están ordenadas (ya probamos que lo están)
- c_{n+1} es extraído en una iteración anterior por el criterio de selección

Luego, concluimos el resultado buscado

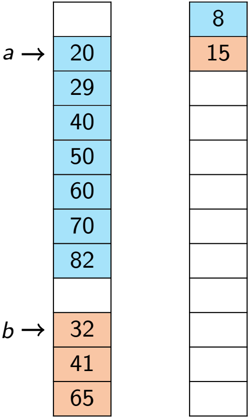
$$c_1 \leq \cdots \leq c_n \leq c_{n+1}$$

□

Complejidad de memoria de `Merge`



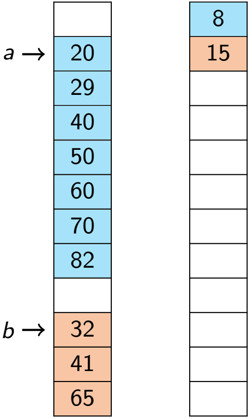
Complejidad de memoria de `Merge`



		8
a →	20	15
	29	
	40	
	50	
	60	
	70	
	82	
b →	32	
	41	
	65	

La ejecución de ejemplo que mostramos considera una nueva secuencia C donde se insertan los valores

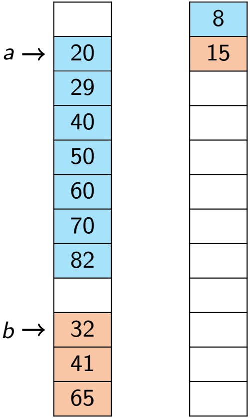
Complejidad de memoria de `Merge`



La ejecución de ejemplo que mostramos considera una nueva secuencia C donde se insertan los valores

- Para $|A| + |B| = n$ necesitamos memoria adicional $\mathcal{O}(n)$

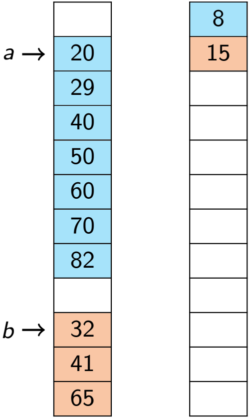
Complejidad de memoria de `Merge`



La ejecución de ejemplo que mostramos considera una nueva secuencia C donde se insertan los valores

- Para $|A| + |B| = n$ necesitamos memoria adicional $\mathcal{O}(n)$
- No necesita mover elementos dentro de ninguna secuencia

Complejidad de memoria de `Merge`

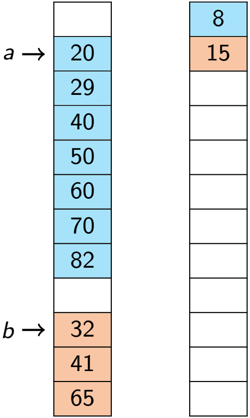


La ejecución de ejemplo que mostramos considera una nueva secuencia C donde se insertan los valores

- Para $|A| + |B| = n$ necesitamos memoria adicional $\mathcal{O}(n)$
- No necesita mover elementos dentro de ninguna secuencia

También se puede realizar *in place*

Complejidad de memoria de `Merge`



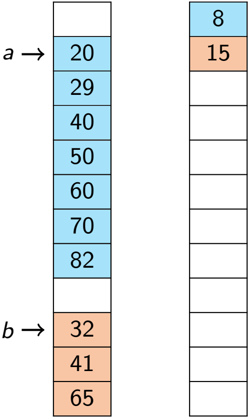
La ejecución de ejemplo que mostramos considera una nueva secuencia C donde se insertan los valores

- Para $|A| + |B| = n$ necesitamos memoria adicional $\mathcal{O}(n)$
- No necesita mover elementos dentro de ninguna secuencia

También se puede realizar *in place*

- Usar el mismo espacio reservado a A y B: memoria adicional $\mathcal{O}(1)$

Complejidad de memoria de `Merge`



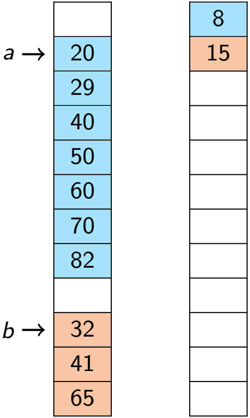
La ejecución de ejemplo que mostramos considera una nueva secuencia C donde se insertan los valores

- Para $|A| + |B| = n$ necesitamos memoria adicional $\mathcal{O}(n)$
- No necesita mover elementos dentro de ninguna secuencia

También se puede realizar *in place*

- Usar el mismo espacio reservado a A y B: memoria adicional $\mathcal{O}(1)$
- Mover todos los datos mayores al insertado

Complejidad de memoria de `Merge`



La ejecución de ejemplo que mostramos considera una nueva secuencia C donde se insertan los valores

- Para $|A| + |B| = n$ necesitamos memoria adicional $\mathcal{O}(n)$
- No necesita mover elementos dentro de ninguna secuencia

También se puede realizar *in place*

- Usar el mismo espacio reservado a A y B: memoria adicional $\mathcal{O}(1)$
- Mover todos los datos mayores al insertado
- Impacta en la complejidad de tiempo...

Complejidad de tiempo de `Merge`

Complejidad de tiempo de `Merge`

Consideramos la implementación sugerida mediante una secuencia **adicional**

Complejidad de tiempo de `Merge`

Consideramos la implementación sugerida mediante una secuencia **adicional**

El algoritmo tiene dos fases

Complejidad de tiempo de `Merge`

Consideramos la implementación sugerida mediante una secuencia **adicional**

El algoritmo tiene dos fases

1. Extracción desde ambas secuencias A y B

Complejidad de tiempo de `Merge`

Consideramos la implementación sugerida mediante una secuencia **adicional**

El algoritmo tiene dos fases

1. Extracción desde ambas secuencias A y B
    - Se decide quién extraer comparando los menores $\mathcal{O}(1)$

Complejidad de tiempo de `Merge`

Consideramos la implementación sugerida mediante una secuencia **adicional**

El algoritmo tiene dos fases

1. Extracción desde ambas secuencias A y B
    - Se decide quién extraer comparando los menores $\mathcal{O}(1)$
    - Se inserta el dato en C $\mathcal{O}(1)$

Complejidad de tiempo de `Merge`

Consideramos la implementación sugerida mediante una secuencia **adicional**

El algoritmo tiene dos fases

1. Extracción desde ambas secuencias A y B
    - Se decide quién extraer comparando los menores $\mathcal{O}(1)$
    - Se inserta el dato en C $\mathcal{O}(1)$
    - Esto se repite $\mathcal{O}(n)$ veces total $\mathcal{O}(n)$

Complejidad de tiempo de `Merge`

Consideramos la implementación sugerida mediante una secuencia **adicional**

El algoritmo tiene dos fases

1. Extracción desde ambas secuencias A y B
    - Se decide quién extraer comparando los menores $\mathcal{O}(1)$
    - Se inserta el dato en C $\mathcal{O}(1)$
    - Esto se repite $\mathcal{O}(n)$ veces total $\mathcal{O}(n)$
2. Reubicación de la secuencia no vacía restante

Complejidad de tiempo de `Merge`

Consideramos la implementación sugerida mediante una secuencia **adicional**

El algoritmo tiene dos fases

1. Extracción desde ambas secuencias A y B
   - Se decide quién extraer comparando los menores $\mathcal{O}(1)$
   - Se inserta el dato en C $\mathcal{O}(1)$
   - Esto se repite $\mathcal{O}(n)$ veces total $\mathcal{O}(n)$
2. Reubicación de la secuencia no vacía restante
   - Se saca un elemento de la restante $\mathcal{O}(1)$

Complejidad de tiempo de `Merge`

Consideramos la implementación sugerida mediante una secuencia **adicional**

El algoritmo tiene dos fases

1. Extracción desde ambas secuencias A y B
    - Se decide quién extraer comparando los menores $\mathcal{O}(1)$
    - Se inserta el dato en C $\mathcal{O}(1)$
    - Esto se repite $\mathcal{O}(n)$ veces total $\mathcal{O}(n)$
2. Reubicación de la secuencia no vacía restante
    - Se saca un elemento de la restante $\mathcal{O}(1)$
    - Se inserta el dato en C $\mathcal{O}(1)$

Complejidad de tiempo de `Merge`

Consideramos la implementación sugerida mediante una secuencia **adicional**

El algoritmo tiene dos fases

1. Extracción desde ambas secuencias A y B
   - Se decide quién extraer comparando los menores $\mathcal{O}(1)$
   - Se inserta el dato en C $\mathcal{O}(1)$
   - Esto se repite $\mathcal{O}(n)$ veces total $\mathcal{O}(n)$
2. Reubicación de la secuencia no vacía restante
   - Se saca un elemento de la restante $\mathcal{O}(1)$
   - Se inserta el dato en C $\mathcal{O}(1)$
   - Esto se repite $\mathcal{O}(n)$ veces total $\mathcal{O}(n)$

Complejidad de tiempo de `Merge`

Consideramos la implementación sugerida mediante una secuencia **adicional**

El algoritmo tiene dos fases

1. Extracción desde ambas secuencias A y B
    - Se decide quién extraer comparando los menores $\mathcal{O}(1)$
    - Se inserta el dato en C $\mathcal{O}(1)$
    - Esto se repite $\mathcal{O}(n)$ veces total $\mathcal{O}(n)$
2. Reubicación de la secuencia no vacía restante
    - Se saca un elemento de la restante $\mathcal{O}(1)$
    - Se inserta el dato en C $\mathcal{O}(1)$
    - Esto se repite $\mathcal{O}(n)$ veces total $\mathcal{O}(n)$

Usando $\mathcal{O}(n)$ memoria adicional, `Merge` es $\mathcal{O}(n)$

Complejidad de tiempo de `Merge` (in place)

Si consideramos usar el espacio reservado para A y B

Complejidad de tiempo de `Merge` (in place)

Si consideramos usar el espacio reservado para A y B

El algoritmo tiene dos fases

Complejidad de tiempo de `Merge` (in place)

Si consideramos usar el espacio reservado para A y B

El algoritmo tiene dos fases

1. Extracción desde ambas secuencias A y B

Complejidad de tiempo de `Merge` (in place)

Si consideramos usar el espacio reservado para A y B

El algoritmo tiene dos fases

1. Extracción desde ambas secuencias A y B
    - Se decide quién extraer comparando los menores $\mathcal{O}(1)$

Complejidad de tiempo de `Merge` (in place)

Si consideramos usar el espacio reservado para A y B

El algoritmo tiene dos fases

1. Extracción desde ambas secuencias A y B
    - Se decide quién extraer comparando los menores $\mathcal{O}(1)$
    - Se inserta el dato en al comienzo $\mathcal{O}(n)$

Complejidad de tiempo de `Merge` (in place)

Si consideramos usar el espacio reservado para A y B

El algoritmo tiene dos fases

1. Extracción desde ambas secuencias A y B
   - Se decide quién extraer comparando los menores $\mathcal{O}(1)$
   - Se inserta el dato en al comienzo $\mathcal{O}(n)$
   - Esto se repite $\mathcal{O}(n)$ veces total $\mathcal{O}(n^2)$

Complejidad de tiempo de `Merge` (in place)

Si consideramos usar el espacio reservado para A y B

El algoritmo tiene dos fases

1. Extracción desde ambas secuencias A y B
    - Se decide quién extraer comparando los menores $\mathcal{O}(1)$
    - Se inserta el dato en al comienzo $\mathcal{O}(n)$
    - Esto se repite $\mathcal{O}(n)$ veces total $\mathcal{O}(n^2)$
2. Reubicación de la secuencia no vacía restante: no necesario
    - El resto de los elementos está en su posición correcta

Complejidad de tiempo de `Merge` (in place)

Si consideramos usar el espacio reservado para A y B

El algoritmo tiene dos fases

1. Extracción desde ambas secuencias A y B
   - Se decide quién extraer comparando los menores $\mathcal{O}(1)$
   - Se inserta el dato en al comienzo $\mathcal{O}(n)$
   - Esto se repite $\mathcal{O}(n)$ veces total $\mathcal{O}(n^2)$
2. Reubicación de la secuencia no vacía restante: no necesario
   - El resto de los elementos está en su posición correcta

Usando $\mathcal{O}(1)$ memoria adicional, `Merge` es $\mathcal{O}(n^2)$

Complejidad de tiempo de `Merge`

- Tenemos un algoritmo **lineal** para obtener una secuencia ordenada

Complejidad de tiempo de `Merge`

- Tenemos un algoritmo **lineal** para obtener una secuencia ordenada
- Pero el requisito de las sub-secuencias ordenadas es demasiado exigente

Complejidad de tiempo de `Merge`

- Tenemos un algoritmo **lineal** para obtener una secuencia ordenada
- Pero el requisito de las sub-secuencias ordenadas es demasiado exigente

¿Podemos usar `Merge` para ordenar una secuencia arbitraria?

Complejidad de tiempo de `Merge`

- Tenemos un algoritmo **lineal** para obtener una secuencia ordenada
- Pero el requisito de las sub-secuencias ordenadas es demasiado exigente

¿Podemos usar `Merge` para ordenar una secuencia arbitraria?

- Dada una secuencia arbitraria

Complejidad de tiempo de `Merge`

- Tenemos un algoritmo **lineal** para obtener una secuencia ordenada
- Pero el requisito de las sub-secuencias ordenadas es demasiado exigente

¿Podemos usar `Merge` para ordenar una secuencia arbitraria?

- Dada una secuencia arbitraria
- Estamos listos si logramos crear dos sub-secuencias ordenadas a partir de ella

Complejidad de tiempo de `Merge`

- Tenemos un algoritmo **lineal** para obtener una secuencia ordenada
- Pero el requisito de las sub-secuencias ordenadas es demasiado exigente

¿Podemos usar `Merge` para ordenar una secuencia arbitraria?

- Dada una secuencia arbitraria
- Estamos listos si logramos crear dos sub-secuencias ordenadas a partir de ella
- Luego las combinamos con `Merge`

Contenidos del curso

1. Estructuras fundamentales
    - Arreglos
    - Listas ligadas
    - Stacks
    - Colas
    - Tablas de Hash
2. Árboles de búsqueda
    - Árboles binarios y balanceados
    - Árboles 2-3
3. Algoritmos de ordenación
    - *insertionsort*
    - *heapsort*
    - *quicksort*
    - *mergesort*
    - *countingsort*
4. Técnicas algorítmicas
    - Dividir para conquistar
    - *Backtracking*
    - Programación dinámica
    - Algoritmos codiciosos
5. Algoritmos en grafos
    - Representación de grafos
    - Exploración
    - Ordenación topológica
    - Componentes fuertemente conectadas
    - Árboles de cobertura
    - Rutas más cortas

Sumario

Dividir para conquistar

Dividir para conquistar

El plan para usar `Merge` en un algoritmo de ordenación sigue la estrategia **dividir para conquistar**

Dividir para conquistar

El plan para usar `Merge` en un algoritmo de ordenación sigue la estrategia **dividir para conquistar**

La estrategia sigue los siguientes pasos

Dividir para conquistar

El plan para usar `Merge` en un algoritmo de ordenación sigue la estrategia **dividir para conquistar**

La estrategia sigue los siguientes pasos

1. Dividir el problema original en dos (o más) **sub-problemas** del mismo tipo

Dividir para conquistar

El plan para usar `Merge` en un algoritmo de ordenación sigue la estrategia **dividir para conquistar**

La estrategia sigue los siguientes pasos

1. Dividir el problema original en dos (o más) **sub-problemas** del mismo tipo
2. Resolver **recursivamente** cada sub-problema

Dividir para conquistar

El plan para usar `Merge` en un algoritmo de ordenación sigue la estrategia **dividir para conquistar**

La estrategia sigue los siguientes pasos

1. Dividir el problema original en dos (o más) **sub-problemas** del mismo tipo
2. Resolver **recursivamente** cada sub-problema
3. Encontrar solución al problema original **combinando** las soluciones a los sub-problemas

Dividir para conquistar

El plan para usar `Merge` en un algoritmo de ordenación sigue la estrategia **dividir para conquistar**

La estrategia sigue los siguientes pasos

1. Dividir el problema original en dos (o más) **sub-problemas** del mismo tipo
2. Resolver **recursivamente** cada sub-problema
3. Encontrar solución al problema original **combinando** las soluciones a los sub-problemas

Los sub-problemas son instancias más pequeñas del problema a resolver

Dividir para conquistar y `Merge`

Dividir para conquistar y `Merge`

Podemos usar la estrategia **dividir para conquistar** en el problema de ordenación, usando `Merge`

Dividir para conquistar y `Merge`

Podemos usar la estrategia **dividir para conquistar** en el problema de ordenación, usando `Merge`

¿En qué parte del dividir para conquistar usaremos `Merge`?

Dividir para conquistar y `Merge`

Podemos usar la estrategia **dividir para conquistar** en el problema de ordenación, usando `Merge`

¿En qué parte del dividir para conquistar usaremos `Merge`?

La idea general para ordenar usando `Merge`define un nuevo algoritmo que llamaremos `MergeSort`

Dividir para conquistar y `Merge`

Podemos usar la estrategia **dividir para conquistar** en el problema de ordenación, usando `Merge`

¿En qué parte del dividir para conquistar usaremos `Merge`?

La idea general para ordenar usando `Merge`define un nuevo algoritmo que llamaremos `MergeSort`

1. Dividir la secuencia original en dos sub-secuencias

Dividir para conquistar y `Merge`

Podemos usar la estrategia **dividir para conquistar** en el problema de ordenación, usando `Merge`

¿En qué parte del dividir para conquistar usaremos `Merge`?

La idea general para ordenar usando `Merge`define un nuevo algoritmo que llamaremos `MergeSort`

1. Dividir la secuencia original en dos sub-secuencias
2. Llamamos recursivamente a `MergeSort` sobre las dos sub-secuencias

Dividir para conquistar y `Merge`

Podemos usar la estrategia **dividir para conquistar** en el problema de ordenación, usando `Merge`

¿En qué parte del dividir para conquistar usaremos `Merge`?

La idea general para ordenar usando `Merge`define un nuevo algoritmo que llamaremos `MergeSort`

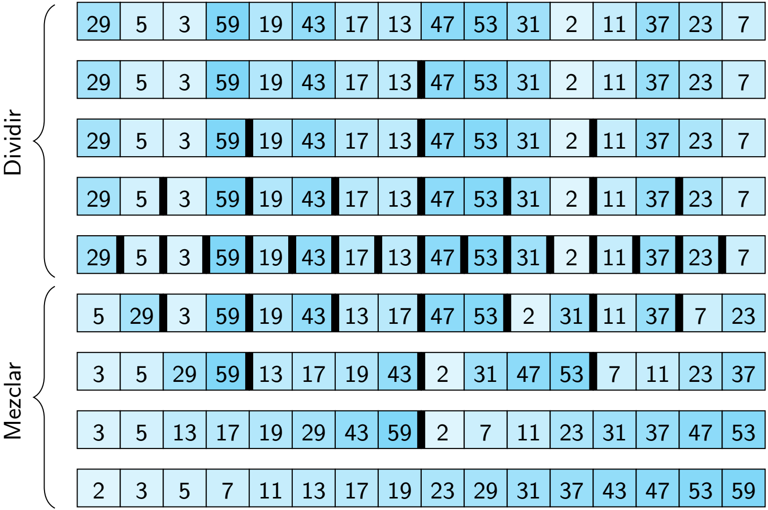
1. Dividir la secuencia original en dos sub-secuencias
2. Llamamos recursivamente a `MergeSort` sobre las dos sub-secuencias
3. Combinamos las secuencias ordenadas resultantes mediante `Merge`

El algoritmo `MergeSort`

A continuación tenemos el pseudocódigo del algoritmo recursivo `MergeSort`

```
input  : Secuencia $A$
output: Secuencia ordenada $B$

MergeSort (A):
    if $|A| = 1$ : return $A$
    Dividir $A$ en mitades $A_1$ y $A_2$
    $B_1 \leftarrow$ MergeSort($A_1$)
    $B_2 \leftarrow$ MergeSort($A_2$)
    $B \leftarrow$ Merge($B_1, B_2$)
    return $B$
```

MergeSort: Ejemplo de ejecución



Dividir	29	5	3	59	19	43	17	13	47	53	31	2	11	37	23	7
	29	5	3	59	19	43	17	13	47	53	31	2	11	37	23	7
	29	5	3	59	19	43	17	13	47	53	31	2	11	37	23	7
	29	5	3	59	19	43	17	13	47	53	31	2	11	37	23	7
	29	5	3	59	19	43	17	13	47	53	31	2	11	37	23	7
Mezclar	5	29	3	59	19	43	13	17	47	53	2	31	11	37	7	23
	3	5	29	59	13	17	19	43	2	31	47	53	7	11	23	37
	3	5	13	17	19	29	43	59	2	7	11	23	31	37	47	53
	2	3	5	7	11	13	17	19	23	29	31	37	43	47	53	59

Correctitud de `MergeSort`

Ejercicio (propuesto)

Demuestre que `MergeSort` es correcto

```
input  : Secuencia $A$
output: Secuencia ordenada $B$
MergeSort $(A)$:
    if $|A| = 1$ : return $A$
    Dividir $A$ en mitades $A_1$ y $A_2$
    $B_1 \leftarrow$ MergeSort($A_1$)
    $B_2 \leftarrow$ MergeSort($A_2$)
    $B \leftarrow$ Merge($B_1, B_2$)
    return $B$
```

Carácter recursivo de `MergeSort`

```
input  : Secuencia $A$
output: Secuencia ordenada $B$

MergeSort (A):
    if $|A| = 1$ : return $A$
    Dividir $A$ en $A_1$ y $A_2$
    $B_1 \leftarrow$ MergeSort($A_1$)
    $B_2 \leftarrow$ MergeSort($A_2$)
    $B \leftarrow$ Merge($B_1, B_2$)
    return $B$
```

Carácter recursivo de `MergeSort`

Todo algoritmo recursivo debe chequear primero el **caso base**

```
input  : Secuencia $A$
output: Secuencia ordenada $B$

MergeSort $(A)$:
    if $|A| = 1$ : return $A$
    Dividir $A$ en $A_1$ y $A_2$
    $B_1 \leftarrow \texttt{MergeSort}(A_1)$
    $B_2 \leftarrow \texttt{MergeSort}(A_2)$
    $B \leftarrow \texttt{Merge}(B_1, B_2)$
    return $B$
```

Carácter recursivo de `MergeSort`

```
input  : Secuencia $A$
output: Secuencia ordenada $B$
MergeSort $(A)$:
    if $|A| = 1$ : return $A$
    Dividir $A$ en $A_1$ y $A_2$
    $B_1 \leftarrow$ MergeSort$(A_1)$
    $B_2 \leftarrow$ MergeSort$(A_2)$
    $B \leftarrow$ Merge$(B_1, B_2)$
    return $B$
```

Todo algoritmo recursivo debe chequear primero el **caso base**

- Es el caso cuya solución no requiere recursión

Carácter recursivo de `MergeSort`

```
input  : Secuencia $A$
output: Secuencia ordenada $B$

MergeSort ($A$):
1    if $|A| = 1$ : return $A$
2    Dividir $A$ en $A_1$ y $A_2$
3    $B_1 \leftarrow$ MergeSort($A_1$)
4    $B_2 \leftarrow$ MergeSort($A_2$)
5    $B \leftarrow$ Merge($B_1, B_2$)
6    return $B$
```

Todo algoritmo recursivo debe chequear primero el **caso base**

- Es el caso cuya solución no requiere recursión
- En `MergeSort`: línea 1

Carácter recursivo de `MergeSort`

```
input  : Secuencia $A$
output: Secuencia ordenada $B$

MergeSort ($A$):
1     if $|A| = 1$ : return $A$
2     Dividir $A$ en $A_1$ y $A_2$
3     $B_1 \leftarrow$ MergeSort($A_1$)
4     $B_2 \leftarrow$ MergeSort($A_2$)
5     $B \leftarrow$ Merge($B_1, B_2$)
6     return $B$
```

Todo algoritmo recursivo debe chequear primero el **caso base**

- Es el caso cuya solución no requiere recursión
- En `MergeSort`: línea 1

Los **llamados recursivos** se hacen sobre casos distintos al original

Carácter recursivo de `MergeSort`

```
input  : Secuencia $A$
output: Secuencia ordenada $B$

MergeSort ($A$):
1    if $|A| = 1$ : return $A$
2    Dividir $A$ en $A_1$ y $A_2$
3    $B_1 \leftarrow$ MergeSort($A_1$)
4    $B_2 \leftarrow$ MergeSort($A_2$)
5    $B \leftarrow$ Merge($B_1, B_2$)
6    return $B$
```

Todo algoritmo recursivo debe chequear primero el **caso base**

- Es el caso cuya solución no requiere recursión
- En `MergeSort`: línea 1

Los **llamados recursivos** se hacen sobre casos distintos al original

- Se acercan un poco más al caso base

Carácter recursivo de `MergeSort`

```
input  : Secuencia $A$
output: Secuencia ordenada $B$

MergeSort $(A)$:
1      if $|A| = 1$ : return $A$
2      Dividir $A$ en $A_1$ y $A_2$
3      $B_1 \leftarrow$ MergeSort$(A_1)$
4      $B_2 \leftarrow$ MergeSort$(A_2)$
5      $B \leftarrow$ Merge$(B_1, B_2)$
6      return $B$
```

Todo algoritmo recursivo debe chequear primero el **caso base**

- Es el caso cuya solución no requiere recursión
- En `MergeSort`: línea 1

Los **llamados recursivos** se hacen sobre casos distintos al original

- Se acercan un poco más al caso base
- En `MergeSort`: líneas 3 y 4

Complejidad de `MergeSort`

Para el análisis de complejidad de tiempo, definimos

$$T(n) \coloneqq \# \text{ pasos para ordenar } n \text{ elementos}$$

Complejidad de `MergeSort`

Para el análisis de complejidad de tiempo, definimos

$$T(n) := \#\text{ pasos para ordenar } n \text{ elementos}$$

Con esto, consideramos los dos casos posibles al llamar a `MergeSort`

```
MergeSort $(A)$:
    if $|A| = 1$ : return $A$
    Dividir $A$ en $A_1$ y $A_2$
    $B_1 \leftarrow$ MergeSort($A_1$)
    $B_2 \leftarrow$ MergeSort($A_2$)
    $B \leftarrow$ Merge($B_1, B_2$)
    return $B$
```

Complejidad de `MergeSort`

Para el análisis de complejidad de tiempo, definimos

$$T(n) := \#\text{ pasos para ordenar } n \text{ elementos}$$

Con esto, consideramos los dos casos posibles al llamar a `MergeSort`

```
MergeSort (A):
    if $|A| = 1$ : return $A$
    Dividir $A$ en $A_1$ y $A_2$
    $B_1 \leftarrow$ MergeSort($A_1$)
    $B_2 \leftarrow$ MergeSort($A_2$)
    $B \leftarrow$ Merge($B_1, B_2$)
    return $B$
```

- Si $n = 1$, aplica el caso base y solo involucra un paso

$$T(1) = 1$$

Complejidad de `MergeSort`

Para el análisis de complejidad de tiempo, definimos

$$T(n) := \# \text{ pasos para ordenar } n \text{ elementos}$$

Con esto, consideramos los dos casos posibles al llamar a `MergeSort`

```
MergeSort (A):
    if $|A| = 1$ : return $A$
    Dividir $A$ en $A_1$ y $A_2$
    $B_1 \leftarrow$ MergeSort($A_1$)
    $B_2 \leftarrow$ MergeSort($A_2$)
    $B \leftarrow$ Merge($B_1, B_2$)
    return $B$
```

- Si $n = 1$, aplica el caso base y solo involucra un paso

$$T(1) = 1$$

- Si $n > 1$, aplican los llamados

Complejidad de `MergeSort`

Para el análisis de complejidad de tiempo, definimos

$$T(n) := \# \text{ pasos para ordenar } n \text{ elementos}$$

Con esto, consideramos los dos casos posibles al llamar a `MergeSort`

```
MergeSort (A):
    if $|A| = 1$ : return $A$
    Dividir $A$ en $A_1$ y $A_2$
    $B_1 \leftarrow$ MergeSort($A_1$)
    $B_2 \leftarrow$ MergeSort($A_2$)
    $B \leftarrow$ Merge($B_1, B_2$)
    return $B$
```

- Si $n = 1$, aplica el caso base y solo involucra un paso

$$T(1) = 1$$

- Si $n > 1$, aplican los llamados
  - Dos llamados de tamaño $n/2$

Complejidad de `MergeSort`

Para el análisis de complejidad de tiempo, definimos

$$T(n) := \# \text{ pasos para ordenar } n \text{ elementos}$$

Con esto, consideramos los dos casos posibles al llamar a `MergeSort`

```
MergeSort (A):
    if $|A| = 1$ : return $A$
    Dividir $A$ en $A_1$ y $A_2$
    $B_1 \leftarrow$ MergeSort($A_1$)
    $B_2 \leftarrow$ MergeSort($A_2$)
    $B \leftarrow$ Merge($B_1, B_2$)
    return $B$
```

- Si $n = 1$, aplica el caso base y solo involucra un paso

$$T(1) = 1$$

- Si $n > 1$, aplican los llamados
  - Dos llamados de tamaño $n/2$
  - Llamado a `Merge`

$$T(n) = 2T(n/2) + n$$

Complejidad de `MergeSort`

Para el análisis de complejidad de tiempo, definimos

$$T(n) := \# \text{ pasos para ordenar } n \text{ elementos}$$

Con esto, consideramos los dos casos posibles al llamar a `MergeSort`

```
MergeSort (A):
    if $|A| = 1$ : return $A$
    Dividir $A$ en $A_1$ y $A_2$
    $B_1 \leftarrow$ MergeSort($A_1$)
    $B_2 \leftarrow$ MergeSort($A_2$)
    $B \leftarrow$ Merge($B_1, B_2$)
    return $B$
```

- Si $n = 1$, aplica el caso base y solo involucra un paso

$$T(1) = 1$$

- Si $n > 1$, aplican los llamados
  - Dos llamados de tamaño $n/2$
  - Llamado a `Merge`

$$T(n) = 2T(n/2) + n$$

Este análisis aplica **para toda** secuencia de input:

Complejidad de `MergeSort`

Para el análisis de complejidad de tiempo, definimos

$$T(n) := \# \text{ pasos para ordenar } n \text{ elementos}$$

Con esto, consideramos los dos casos posibles al llamar a `MergeSort`

```
MergeSort $(A)$:
    if $|A| = 1$ : return $A$
    Dividir $A$ en $A_1$ y $A_2$
    $B_1 \leftarrow$ MergeSort$(A_1)$
    $B_2 \leftarrow$ MergeSort$(A_2)$
    $B \leftarrow$ Merge$(B_1, B_2)$
    return $B$
```

- Si $n = 1$, aplica el caso base y solo involucra un paso

$$T(1) = 1$$

- Si $n > 1$, aplican los llamados
  - Dos llamados de tamaño $n/2$
  - Llamado a `Merge`

$$T(n) = 2T(n/2) + n$$

Este análisis aplica **para toda** secuencia de input:
Nos entregará el resultado de peor, mejor y caso promedio

Complejidad de `MergeSort`

La siguiente relación es una **relación de recurrencia**

$$T(1) = 1, \quad T(n) = 2T\left(\frac{n}{2}\right) + n$$

Complejidad de `MergeSort`

La siguiente relación es una **relación de recurrencia**

$$T(1) = 1, \quad T(n) = 2T\left(\frac{n}{2}\right) + n$$

Podemos resolverla notando que la parte recursiva puede ser reescrita como

$$T(n) = 2T(n/2) + n \quad \Leftrightarrow \quad \frac{T(n)}{n} = \frac{T(n/2)}{n/2} + 1$$

Complejidad de `MergeSort`

La siguiente relación es una **relación de recurrencia**

$$T(1) = 1, \quad T(n) = 2T\left(\frac{n}{2}\right) + n$$

Podemos resolverla notando que la parte recursiva puede ser reescrita como

$$T(n) = 2T(n/2) + n \quad \Leftrightarrow \quad \frac{T(n)}{n} = \frac{T(n/2)}{n/2} + 1$$

La gracia de esta expresión es que numeradores y denominadores incluyen la misma fracción de n

Complejidad de `MergeSort`

La siguiente relación es una **relación de recurrencia**

$$T(1) = 1, \quad T(n) = 2T\left(\frac{n}{2}\right) + n$$

Podemos resolverla notando que la parte recursiva puede ser reescrita como

$$T(n) = 2T(n/2) + n \quad \Leftrightarrow \quad \frac{T(n)}{n} = \frac{T(n/2)}{n/2} + 1$$

La gracia de esta expresión es que numeradores y denominadores incluyen la misma fracción de *n*

Sin pérdida de generalidad, supondremos que *n* es potencia de 2

Complejidad de `MergeSort`

Construimos un sistema de ecuaciones reemplazando el argumento del lado izquierdo por $n, n/2, n/4, \ldots, 2$ de forma que el último término contiene $T(1)$ (nuestro caso base)

Complejidad de `MergeSort`

Construimos un sistema de ecuaciones reemplazando el argumento del lado izquierdo por $n, n/2, n/4, \ldots, 2$ de forma que el último término contiene $T(1)$ (nuestro caso base)

$$\text{ecuación 1} \qquad \frac{T(n)}{n} \qquad = \qquad \frac{T(n/2)}{n/2} + 1$$

Complejidad de `MergeSort`

Construimos un sistema de ecuaciones reemplazando el argumento del lado izquierdo por $n, n/2, n/4, \ldots, 2$ de forma que el último término contiene $T(1)$ (nuestro caso base)

$$\begin{array}{lccc} \text{ecuación 1} & \dfrac{T(n)}{n} & = & \dfrac{T(n/2)}{n/2} + 1 \\ \text{ecuación 2} & \dfrac{T(n/2)}{n/2} & = & \dfrac{T(n/4)}{n/4} + 1 \end{array}$$

Complejidad de `MergeSort`

Construimos un sistema de ecuaciones reemplazando el argumento del lado izquierdo por $n, n/2, n/4, \ldots, 2$ de forma que el último término contiene $T(1)$ (nuestro caso base)

$$\begin{array}{lccc}
\text{ecuación 1} & \dfrac{T(n)}{n} & = & \dfrac{T(n/2)}{n/2} + 1 \\
\text{ecuación 2} & \dfrac{T(n/2)}{n/2} & = & \dfrac{T(n/4)}{n/4} + 1
\end{array}$$

$$\ldots$$

Complejidad de `MergeSort`

Construimos un sistema de ecuaciones reemplazando el argumento del lado izquierdo por $n, n/2, n/4, \ldots, 2$ de forma que el último término contiene $T(1)$ (nuestro caso base)

$$\begin{array}{lccc}
\text{ecuación } 1 & \dfrac{T(n)}{n} & = & \dfrac{T(n/2)}{n/2} + 1 \\[2ex]
\text{ecuación } 2 & \dfrac{T(n/2)}{n/2} & = & \dfrac{T(n/4)}{n/4} + 1 \\[2ex]
 & & \cdots & \\[2ex]
\text{ecuación } k & \dfrac{T(2)}{2} & = & \dfrac{T(1)}{1} + 1
\end{array}$$

Complejidad de `MergeSort`

Construimos un sistema de ecuaciones reemplazando el argumento del lado izquierdo por $n, n/2, n/4, \ldots, 2$ de forma que el último término contiene $T(1)$ (nuestro caso base)

$$\begin{array}{lccc}
\text{ecuación } 1 & \dfrac{T(n)}{n} & = & \dfrac{T(n/2)}{n/2} + 1 \\
\text{ecuación } 2 & \dfrac{T(n/2)}{n/2} & = & \dfrac{T(n/4)}{n/4} + 1 \\
 & & \cdots & \\
\text{ecuación } k & \dfrac{T(2)}{2} & = & \dfrac{T(1)}{1} + 1
\end{array}$$

Como el lado derecho de la i-ésima ecuación considera la potencia 2^i, de la k-ésima ecuación deducimos

Complejidad de `MergeSort`

Construimos un sistema de ecuaciones reemplazando el argumento del lado izquierdo por $n, n/2, n/4, \ldots, 2$ de forma que el último término contiene $T(1)$ (nuestro caso base)

$$\begin{array}{lccc}
\text{ecuación } 1 & \dfrac{T(n)}{n} & = & \dfrac{T(n/2)}{n/2} + 1 \\
\text{ecuación } 2 & \dfrac{T(n/2)}{n/2} & = & \dfrac{T(n/4)}{n/4} + 1 \\
 & & \cdots & \\
\text{ecuación } k & \dfrac{T(2)}{2} & = & \dfrac{T(1)}{1} + 1
\end{array}$$

Como el lado derecho de la i-ésima ecuación considera la potencia 2^i, de la k-ésima ecuación deducimos

$$1 = \frac{n}{2^k} \quad \Rightarrow \quad 2^k = n \quad \Rightarrow \quad k = \log(n)$$

Complejidad de `MergeSort`

Sumamos las $\log(n)$ ecuaciones y simplificamos los términos que aparecen a ambos lados

$$
\begin{array}{lcccc}
\text{ecuación } 1 & \dfrac{T(n)}{n} & = & \dfrac{T(n/2)}{n/2} & +1 \\
\text{ecuación } 2 & \dfrac{T(n/2)}{n/2} & = & \dfrac{T(n/4)}{n/4} & +1 \\
 & & \cdots & & \\
\text{ecuación } k & \dfrac{T(2)}{2} & = & \dfrac{T(1)}{1} & +1 \\
\hline
\end{array}
$$

Complejidad de `MergeSort`

Sumamos las $\log(n)$ ecuaciones y simplificamos los términos que aparecen a ambos lados

$$
\begin{array}{rcccl}
\text{ecuación } 1 & \dfrac{T(n)}{n} & = & \dfrac{T(n/2)}{n/2} & +1 \\
\text{ecuación } 2 & \dfrac{T(n/2)}{n/2} & = & \dfrac{T(n/4)}{n/4} & +1 \\
 & & \cdots & & \\
\text{ecuación } k & \dfrac{T(2)}{2} & = & \dfrac{T(1)}{1} & +1 \\
\hline
\text{suma} & \dfrac{T(n)}{n} & = & \dfrac{T(1)}{1} & +\log(n)
\end{array}
$$

Complejidad de `MergeSort`

Sumamos las $\log(n)$ ecuaciones y simplificamos los términos que aparecen a ambos lados

$$\begin{array}{rccccc}
\text{ecuación } 1 & \dfrac{T(n)}{n} & = & \dfrac{T(n/2)}{n/2} & +1 \\
\text{ecuación } 2 & \dfrac{T(n/2)}{n/2} & = & \dfrac{T(n/4)}{n/4} & +1 \\
 & & \dots & & \\
\text{ecuación } k & \dfrac{T(2)}{2} & = & \dfrac{T(1)}{1} & +1 \\
\hline
\text{suma} & \dfrac{T(n)}{n} & = & \dfrac{T(1)}{1} & +\log(n)
\end{array}$$

Despejando, obtenemos $T(n) = n\log(n) + n$

Complejidad de `MergeSort`

Sumamos las $\log(n)$ ecuaciones y simplificamos los términos que aparecen a ambos lados

$$
\begin{array}{lccc}
\text{ecuación } 1 & \dfrac{T(n)}{n} & = \dfrac{T(n/2)}{n/2} & +1 \\
\text{ecuación } 2 & \dfrac{T(n/2)}{n/2} & = \dfrac{T(n/4)}{n/4} & +1 \\
 & & \dots & \\
\text{ecuación } k & \dfrac{T(2)}{2} & = \dfrac{T(1)}{1} & +1 \\
\hline
\text{suma} & \dfrac{T(n)}{n} & = \dfrac{T(1)}{1} & +\log(n)
\end{array}
$$

Despejando, obtenemos $T(n) = n\log(n) + n$

La complejidad de tiempo de `MergeSort` es $\mathcal{O}(n\log(n))$

Complejidad de `MergeSort`

En términos de memoria adicional

```
MergeSort (A):
    if $|A| = 1$ : return $A$
    Dividir $A$ en $A_1$ y $A_2$
    $B_1 \leftarrow$ MergeSort($A_1$)
    $B_2 \leftarrow$ MergeSort($A_2$)
    $B \leftarrow$ Merge($B_1, B_2$)
    return $B$
```

Complejidad de `MergeSort`

En términos de memoria adicional

```
MergeSort (A):
    if $|A| = 1$ : return $A$
    Dividir $A$ en $A_1$ y $A_2$
    $B_1 \leftarrow$ MergeSort($A_1$)
    $B_2 \leftarrow$ MergeSort($A_2$)
    $B \leftarrow$ Merge($B_1, B_2$)
    return $B$
```

- El caso base no ocupa memoria adicional

Complejidad de `MergeSort`

En términos de memoria adicional

```
MergeSort $(A)$:
1     if $|A| = 1$ : return $A$
2     Dividir $A$ en $A_1$ y $A_2$
3     $B_1 \leftarrow$ MergeSort($A_1$)
4     $B_2 \leftarrow$ MergeSort($A_2$)
5     $B \leftarrow$ Merge($B_1, B_2$)
6     return $B$
```

- El caso base no ocupa memoria adicional
- Para $|A| = n$, la línea 5 ocupa $\mathcal{O}(n)$

Complejidad de `MergeSort`

En términos de memoria adicional

```
MergeSort ($A$):
1     if $|A| = 1$ : return $A$
2     Dividir $A$ en $A_1$ y $A_2$
3     $B_1 \leftarrow$ MergeSort($A_1$)
4     $B_2 \leftarrow$ MergeSort($A_2$)
5     $B \leftarrow$ Merge($B_1, B_2$)
6     return $B$
```

- El caso base no ocupa memoria adicional
- Para $|A| = n$, la línea 5 ocupa $\mathcal{O}(n)$
- Ojo! Los llamados recursivos no van acumulando memoria reservada, por lo que **no sumamos** $\mathcal{O}(n)$ por llamado

Complejidad de `MergeSort`

En términos de memoria adicional

```
MergeSort (A):
1    if $|A| = 1$ : return $A$
2    Dividir $A$ en $A_1$ y $A_2$
3    $B_1 \leftarrow$ MergeSort($A_1$)
4    $B_2 \leftarrow$ MergeSort($A_2$)
5    $B \leftarrow$ Merge($B_1, B_2$)
6    return $B$
```

- El caso base no ocupa memoria adicional
- Para $|A| = n$, la línea 5 ocupa $\mathcal{O}(n)$
- Ojo! Los llamados recursivos no van acumulando memoria reservada, por lo que **no sumamos** $\mathcal{O}(n)$ por llamado

La memoria adicional **se puede reciclar:**
La complejidad de memoria de `MergeSort` es $\mathcal{O}(n)$

Complejidad de algoritmos de ordenación

Resumimos los resultados de complejidad por caso hasta el momento

Algoritmo	Mejor caso	Caso promedio	Peor caso	Memoria adicional
Selection Sort	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(1)$
Insertion Sort	$\mathcal{O}(n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(1)$
Merge Sort	$\mathcal{O}(n\log(n))$	$\mathcal{O}(n\log(n))$	$\mathcal{O}(n\log(n))$	$\mathcal{O}(n)$
Quick Sort	?	?	?	?
Heap Sort	?	?	?	?

Complejidad de algoritmos de ordenación

Resumimos los resultados de complejidad por caso hasta el momento

Algoritmo	Mejor caso	Caso promedio	Peor caso	Memoria adicional
Selection Sort	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(1)$
Insertion Sort	$\mathcal{O}(n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(1)$
Merge Sort	$\mathcal{O}(n\log(n))$	$\mathcal{O}(n\log(n))$	$\mathcal{O}(n\log(n))$	$\mathcal{O}(n)$
Quick Sort	?	?	?	?
Heap Sort	?	?	?	?

Notemos la mejora en tiempo con `MergeSort`
a cambio de memoria adicional

Sumario

Objetivos de la clase

Objetivos de la clase

☐ Comprender el algoritmo `Merge` para combinar secuencias ordenadas

Objetivos de la clase

☐ Comprender el algoritmo `Merge` para combinar secuencias ordenadas

☐ Determinar complejidad de `Merge` y el *trade off* de ejecutarlo *in place*

Objetivos de la clase

☐ Comprender el algoritmo `Merge` para combinar secuencias ordenadas

☐ Determinar complejidad de `Merge` y el *trade off* de ejecutarlo *in place*

☐ Demostrar correctitud de `Merge`

Objetivos de la clase

☐ Comprender el algoritmo `Merge` para combinar secuencias ordenadas

☐ Determinar complejidad de `Merge` y el *trade off* de ejecutarlo *in place*

☐ Demostrar correctitud de `Merge`

☐ Comprender el uso de `Merge` como algoritmo de ordenación general en `MergeSort`

Objetivos de la clase

☐ Comprender el algoritmo `Merge` para combinar secuencias ordenadas

☐ Determinar complejidad de `Merge` y el *trade off* de ejecutarlo *in place*

☐ Demostrar correctitud de `Merge`

☐ Comprender el uso de `Merge` como algoritmo de ordenación general en `MergeSort`

☐ Determinar la complejidad de `MergeSort`