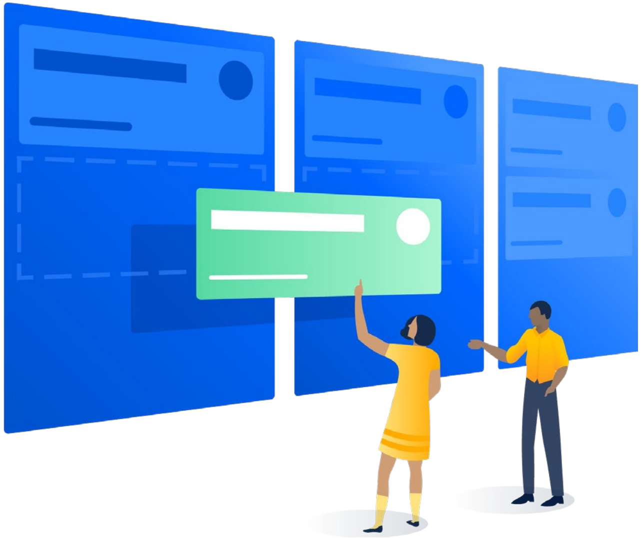
Divide y vencerás

Tema 3

Los contenidos de esta presentación han sido adaptados a partir del material creado por los profesores Javier Junquera Sánchez y Francisco Manuel Sáez de Adana Herrero.

Índice

- **Introducción**
  - Divide y vencerás
  - Búsqueda binaria
- **Ordenación eficiente**
  - Mergesort
  - Quicksort
- **Ejemplos**
  - Exponenciación binaria
  - Multiplicación de matrices



Ejemplo

Universidad de Alcalá

¿Cómo buscar un portal en una calle?

Ejemplo

¿Cómo buscar una palabra en el diccionario?

Divide y vencerás

Es una técnica de diseño de algoritmos:

La técnica divide y vencerás consiste en descomponer el caso a resolver en subcasos más pequeños del mismo problema.

Posteriormente, se resuelve independientemente cada subcaso y se combinan los resultados para construir la solución el caso original.

Este proceso se suele aplicar recursivamente y la eficiencia de esta técnica depende de cómo se resuelvan los subcasos.

Se trata, por tanto, de un esquema en 3 etapas.

Divide y vencerás

1. **Dividir**

   Descomponer el problema en subproblemas, hasta encontrar el caso base.

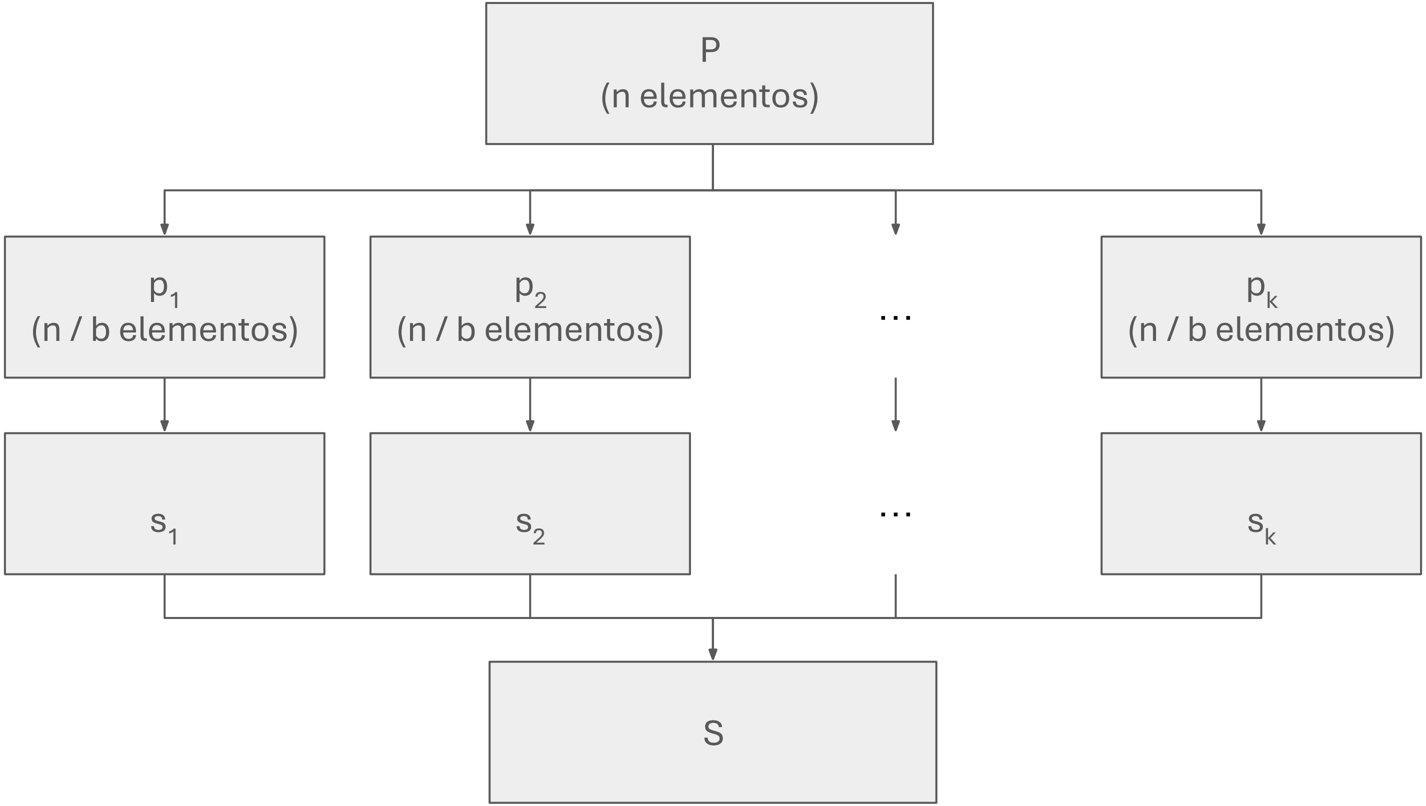
2. **Conquistar**

   Resolver los subproblemas, y obtener soluciones parciales.

3. **Vencer**

   Combinar las soluciones parciales para obtener la solución.

Divide y vencerás



Divide y vencerás

Para aplicar la estrategia divide y vencerás, debemos asegurarnos de que:

- Un subcaso concreto, sólo debe evaluarse una vez.
- Los subcasos deben ser de tamaño similar.
- Recomponer los subcasos debe permitir obtener la solución.

Hay casos que podremos resolver iterativamente (i.e., dividiendo el problema, y obteniendo la solución mezclando soluciones parciales), o recursivamente (i.e., profundizando en las particiones hasta encontrar la solución)

Divide y vencerás

Implementación en Python:

```
def divide_y_venceras(problema):

    # Caso base: si el problema es lo suficientemente pequeño, resolverlo directamente
    if es_solucion_simple(problema):
        return solucion_directa(problema)

    # Dividir el problema en subproblemas más pequeños
    subproblemas = dividir(problema)

    # Resolver cada subproblema recursivamente
    soluciones_parciales = []
    for p in subproblemas:
        soluciones_parciales += [divide_y_venceras(p)]

    # Combinar las soluciones de los subproblemas para obtener la solución
    solucion = combinar(soluciones_parciales)

    return solucion
```

Divide y vencerás

Determinación del umbral (caso base):

Cuando el problema sea lo suficientemente pequeño no se realizan más divisiones.

Este límite lo marca el umbral. El umbral será un n_0 tal que, cuando el tamaño del caso sea menor o igual que este, el problema se resolverá de forma directa, es decir, no se generarán más llamadas recursivas.

La determinación del umbral óptimo es un problema complejo.

Divide y vencerás

Determinación del umbral (caso base):

Dada una implementación particular, el umbral puede calcularse empíricamente mediante pruebas de rendimiento.

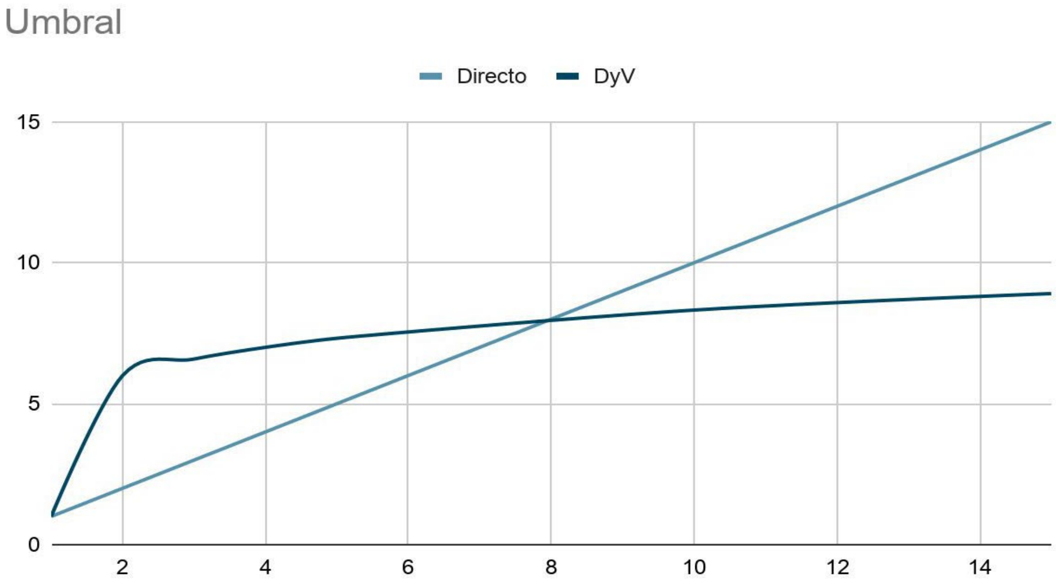
También puede calcularse usando la siguiente técnica:

- Obtener las ecuaciones de recurrencia del algoritmo.
- Determinar los valores de las constantes para una implementación concreta del algoritmo.
- El umbral óptimo se estima hallando el tamaño n del caso para el cual no hay diferencia entre aplicar directamente el algoritmo clásico o pasar a un nivel más de recursión.

Divide y vencerás

Determinación del umbral (caso base):

En todo caso, se utilizará el algoritmo directo cuando el tamaño del problema sea menor que el umbral elegido.



Divide y vencerás

Coste computacional (teorema maestro):

$$T_2(n) = \begin{cases} g(n) & \text{si } 0 \le n < c \\ a \cdot T_2(n/c) + b \cdot n^k & \text{si } c \le n \end{cases} \Rightarrow T_2(n) \in \begin{cases} \Theta(n^k) & \text{si } a < c^k \\ \Theta(n^k \cdot \log n) & \text{si } a = c^k \\ \Theta(n^{\log_c a}) & \text{si } a > c^k \end{cases}$$

Sea n el tamaño de nuestro caso original y sea a el número de subcasos; existe una constante c tal que el tamaño de los subcasos es aproximadamente n/c.

La función $g(n)$ representa el tiempo de resolver el problema de forma directa (caso base).

Los valores más frecuentes para k son $k = 0$ y $k = 1$, dependen de las operaciones adicionales.

Un ejemplo: Calcular x^n

Dado un número base x y un exponente n, se quiere calcular x^n de forma eficiente.

El método más simple es multiplicar x x por sí mismo n veces:

$$x^n = x \times x \times x \times \ldots \times x \quad (n \text{ veces})$$

- Este enfoque tiene un costo de $O(n)$
- Es ineficiente para exponentes grandes.

Un ejemplo: Calcular x^n

Solución: Exponentiación rápida.

Se usa recursión y dividir el problema en partes más pequeñas.

División del problema:

Sabiendo que:

- Si n es par:

$$x^n = (x^{n/2}) \times (x^{n/2})$$

- Si n es impar:

$$x^n = x \times (x^{(n-1)/2}) \times (x^{(n-1)/2})$$

- Este método reduce la cantidad de multiplicaciones
- Se pasa de $O(n)$ a $O(\log n)$

Un ejemplo: Calcular x^n

Código en Python:

```python
def exp_rapida(x, n):
    if n == 0:
        return 1  # Caso base: x^0 = 1
    mitad = exp_rapida(x, n // 2)  # Resuelve x^(n/2)
    resultado = mitad * mitad  # Solo una multiplicación
    if n % 2 != 0:  # Si n es impar, multiplicamos por x extra
        resultado *= x
    return resultado
```

Un ejemplo práctico: Calcular 2^{10}

Universidad de Alcalá

Con el enfoque se tiene:

Nivel	Operaciones en cada nivel
$2^{10} = 2^5 \times 2^5$	1 multiplicación
$2^5 = 2 \times (2^2 \times 2^2)$	1 multiplicación por lado (**2 en total**)
$2^2 = 2^1 \times 2^1$	1 multiplicación por lado (**4 en total**)
$2^1 = 2$ (Caso base)	**0 multiplicaciones**

✅ **Total: 1 + 2 + 1 = 4 multiplicaciones**

Con el método más simple se realizan 10 multiplicaciones

Un ejemplo: Calcular x^n

Universidad de Alcalá

Analizando la Recursión:

```
exp_rapida(2, 10)
 ├── exp_rapida(2, 5)
 │    ├── exp_rapida(2, 2)
 │    │    ├── exp_rapida(2, 1)
 │    │    │    ├── exp_rapida(2, 0) → Devuelve 1
 │    │    │    ├── Combina: 1 × 1 → Devuelve 2
 │    │    ├── Combina: 2 × 2 → Devuelve 4
 │    ├── exp_rapida(2, 5) → Toma 4 y lo usa: 4 × 4 = 16
 │    ├── Como 5 es impar, multiplica por 2: 16 × 2 = 32
 ├── Combina: 32 × 32 → Devuelve 1024 ✅
```

Búsqueda binaria

Universidad de Alcalá

Se trata de una de las aplicaciones más sencillas de divide y vencerás. Realmente no se va dividiendo el problema, sino que se va reduciendo su tamaño en cada paso. Es un algoritmo divide y vencerás de reducción o simplificación.

1. **Dividir:** El problema se descompone en subproblemas de menor tamaño (n / 2).
2. **Conquistar:** No hay soluciones parciales, la solución es única.
3. **Vencer:** No es necesario combinar las soluciones.

Búsqueda binaria

¿Está el número 77?

10	13	35	59	77	85	95	96

Búsqueda binaria

¿Está el número 77?

10	13	35	59	77	85	95	96

¿77 == 59?

¿77 > 59?

Búsqueda binaria

¿Está el número 77?

10	13	35	59	77	85	95	96

¿77 == 85?

¿77 > 85?

Búsqueda binaria

¿Está el número 77?

10	13	35	59	77	85	95	96

¿77 == 77?

Búsqueda binaria

Implementación en Python:

```python
def busqueda_binaria(array :list, item :int) -> bool:

    # Si el array no tiene elementos
    if len(array) == 0:
        return False

    # Si el elemento central coincide con el que buscamos
    mid = len(array) // 2
    if array[mid] == item:
        return True

    # Dividir y resolver recursivamente
    if item < array[mid]:
        return busqueda_binaria(array[:mid], item)
    else:
        return busqueda_binaria(array[mid + 1:], item)

    # No es necesario combinar las soluciones
```

Búsqueda binaria

Complejidad:

10	13	35	59	77	85	95	96	n/2

10	13	35	59	77	85	95	96	n/2

10	13	35	59	77	85	95	96	n/2

En cada paso, el tamaño del problema se reduce a la mitad.

Búsqueda binaria

Complejidad:

n = 100 →	50, 25, 13, 7, 4, 2, 1	→	7 pasos
n = 50 →	25, 13, 7, 4, 2, 1	→	6 pasos
n = 20 →	10, 5, 3, 2, 1	→	5 pasos
n = 10 →	5, 3, 2, 1	→	4 pasos
n = 8 →	4, 2, 1	→	3 pasos

Búsqueda binaria

Universidad de Alcalá

Complejidad:

n = 100	→	50, 25, 13, 7, 4, 2, 1	→	7 pasos	→	$2^7 = 128$	→	$\log(128)$
n = 50	→	25, 13, 7, 4, 2, 1	→	6 pasos	→	$2^6 = 64$	→	$\log(64)$
n = 20	→	10, 5, 3, 2, 1	→	5 pasos	→	$2^5 = 32$	→	$\log(32)$
n = 10	→	5, 3, 2, 1	→	4 pasos	→	$2^4 = 16$	→	$\log(16)$
n = 8	→	4, 2, 1	→	3 pasos	→	$2^3 = 8$	→	$\log(8)$

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

Búsqueda binaria

Universidad de Alcalá

Complejidad (teorema maestro):

$$T_2(n) = \begin{cases} g(n) & \text{si } 0 \le n < c \\ a \cdot T_2(n/c) + b \cdot n^k & \text{si } c \le n \end{cases} \Rightarrow T_2(n) \in \begin{cases} \Theta(n^k) & \text{si } a < c^k \\ \Theta(n^k \cdot \log n) & \text{si } a = c^k \\ \Theta(n^{\log_c a}) & \text{si } a > c^k \end{cases}$$

- $a = 1$ → Porque al dividir, solo nos quedamos con un caso.
- $c = 2$ → Porque dividimos el tamaño del problema entre 2.
- $k = 0$ → Porque las operaciones adicionales son constantes.

$$T(n) \in O(n^0 \log(n)) \in O(\log(n))$$

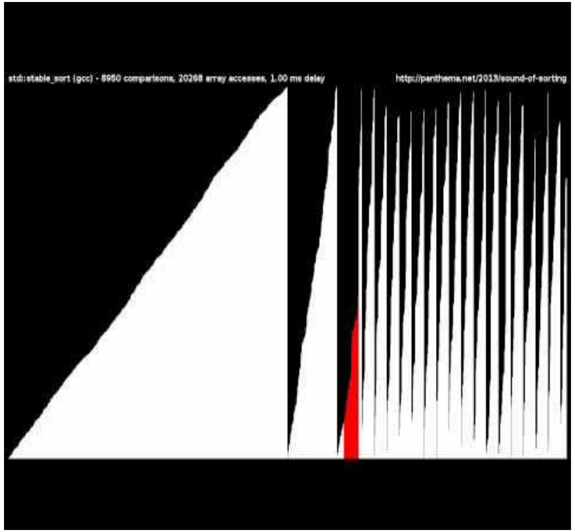
Ordenación eficiente

Dado un vector T[1..n], inicialmente desordenado, se ordenará aplicando la técnica de divide y vencerás, partiendo el vector inicial en dos subvectores más pequeños.

Se utilizarán dos técnicas:

- Ordenación por mezcla (Mergesort)
- Ordenación rápida (Quicksort)

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



MergeSort

Pasos:

1. Se divide el vector en dos mitades de forma recursiva.
2. Cada subconjunto pertenece a la mitad izquierda o a la mitad derecha de un subconjunto más grande.
3. Se combinan los subconjuntos de forma ordenada para obtener la solución.

4	1	7	3	2	9	5

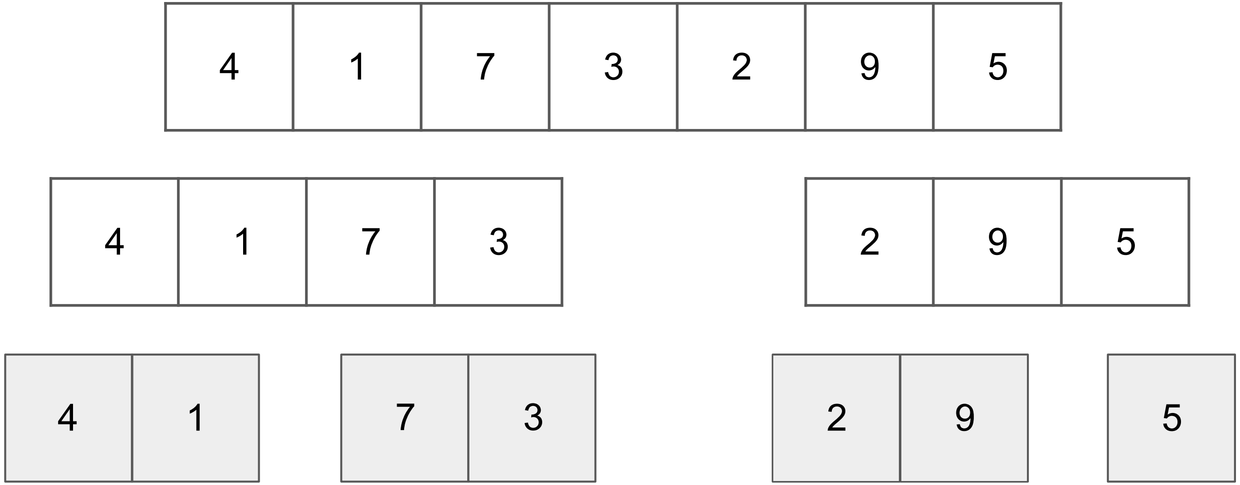
MergeSort

4	1	7	3	2	9	5

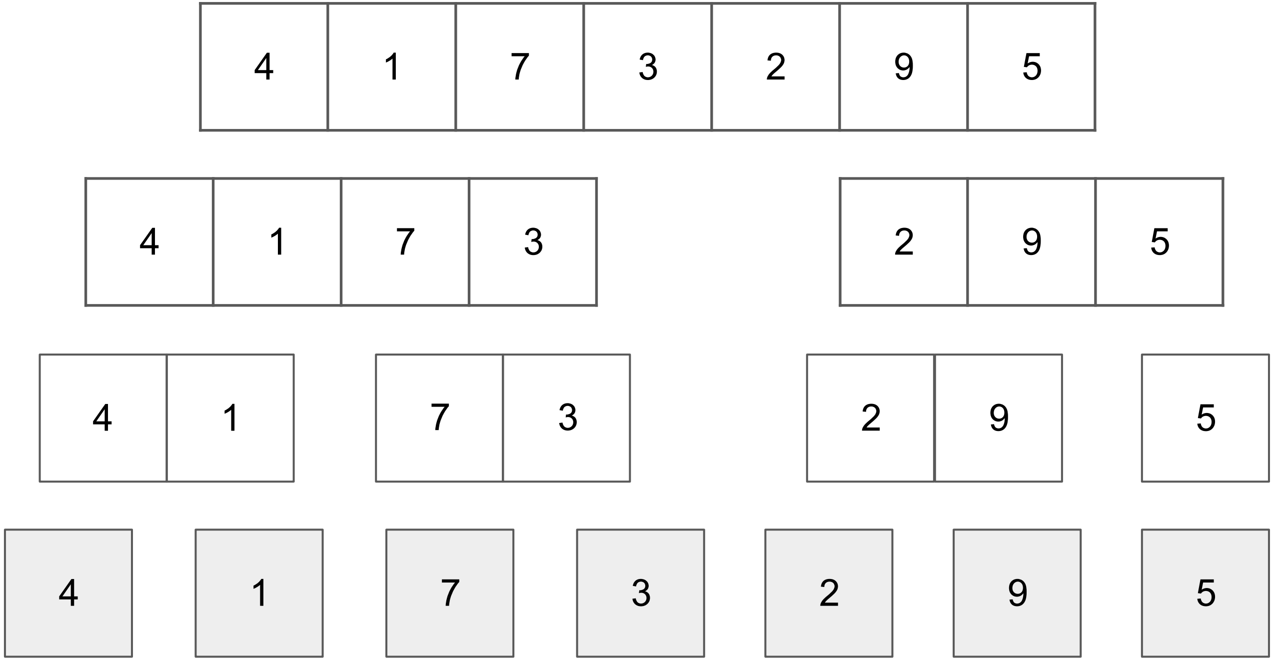
4	1	7	3

2	9	5

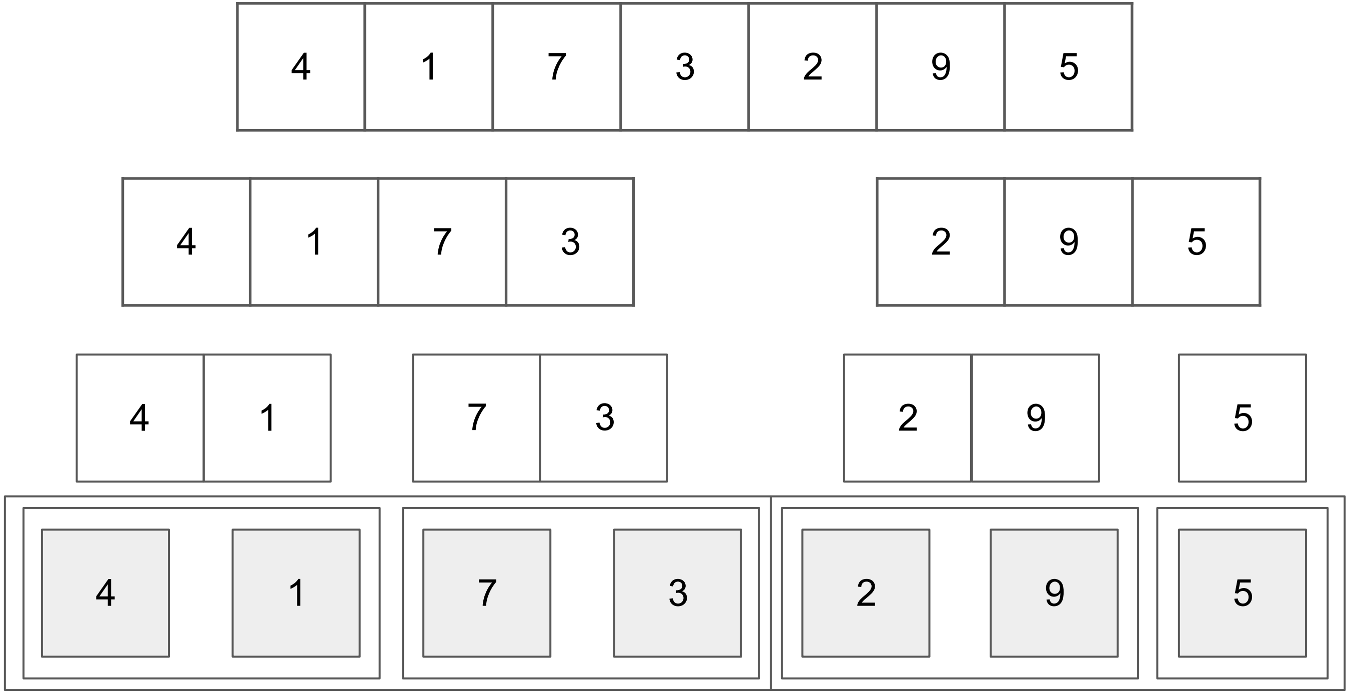
MergeSort



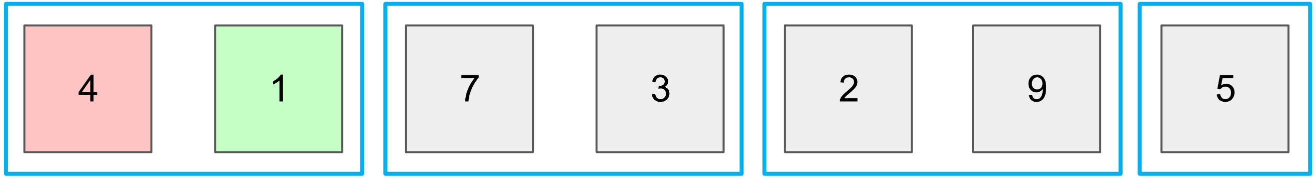
MergeSort



MergeSort



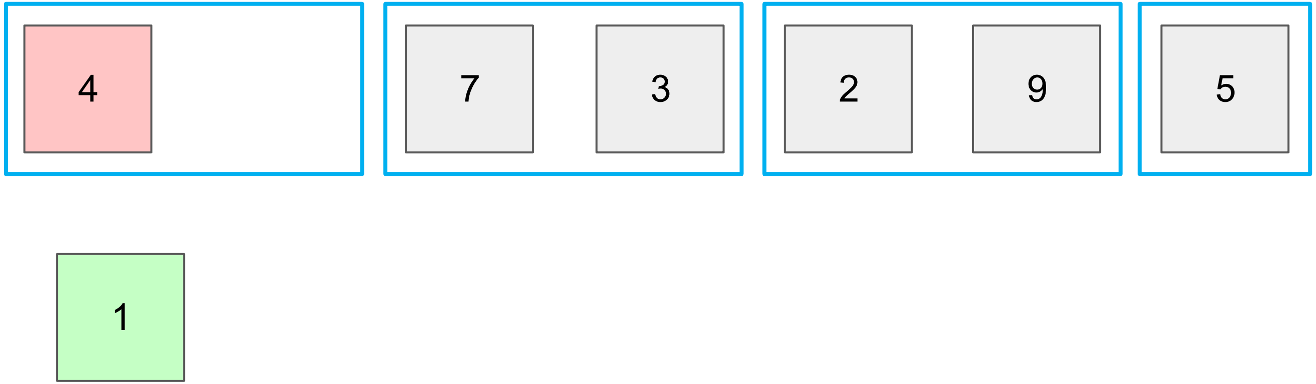
MergeSort



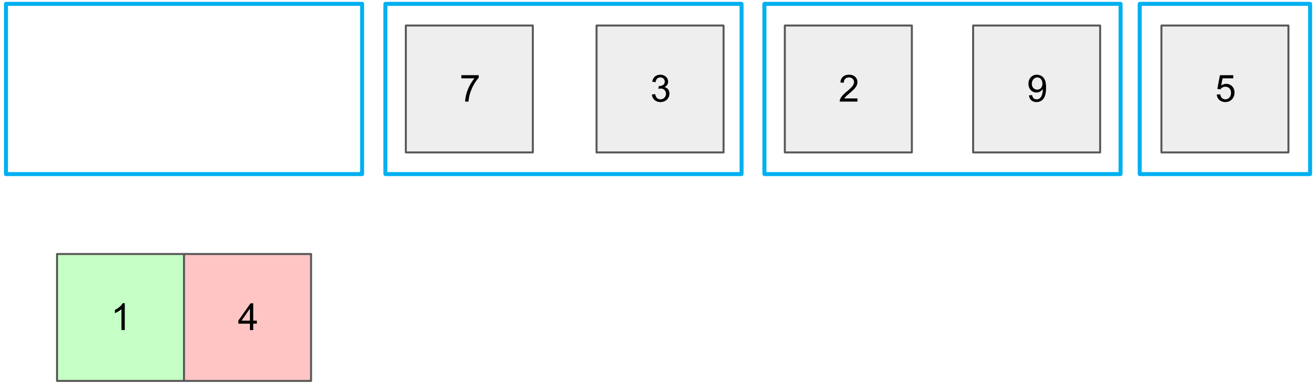
Para combinar:

- Se compara el primer elemento de la izquierda con el primero de la derecha y se escoge el menor hasta que una de las partes se quede sin elementos.
- Se añaden los elementos restantes de la otra parte.

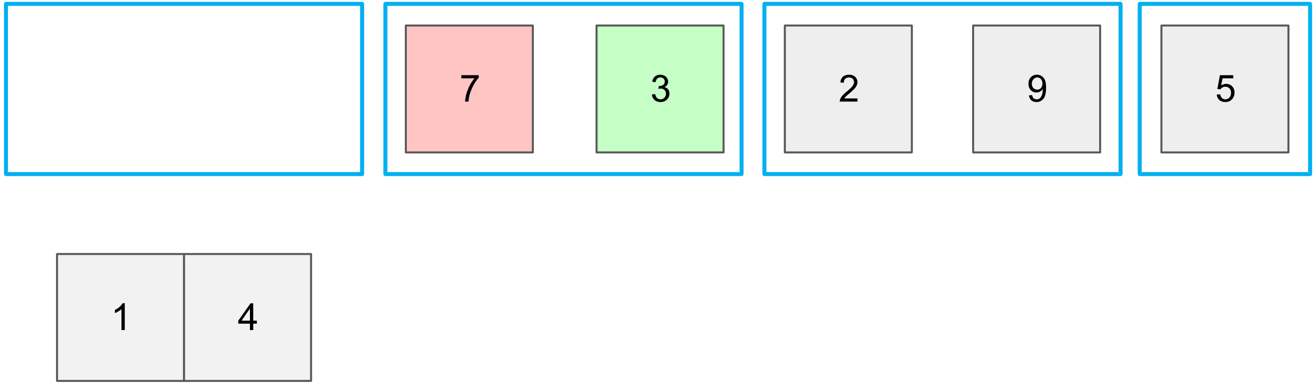
MergeSort



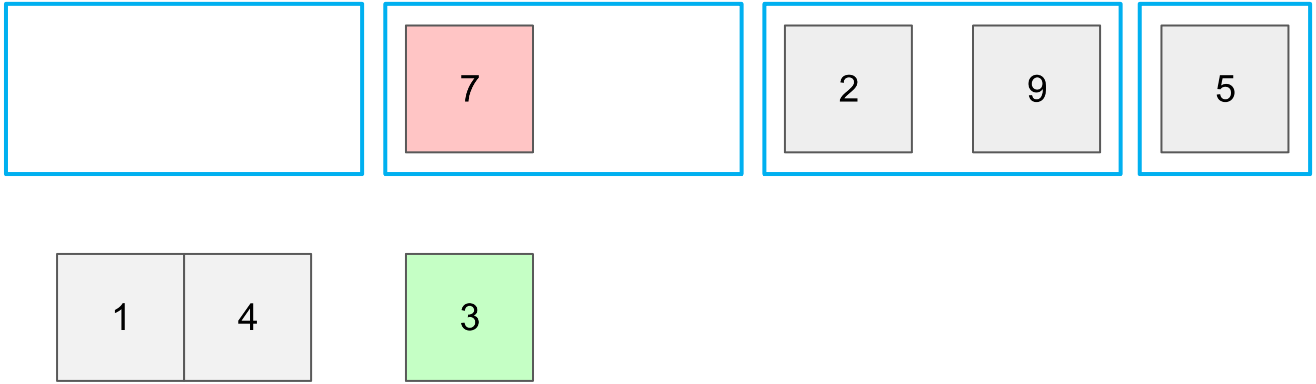
MergeSort



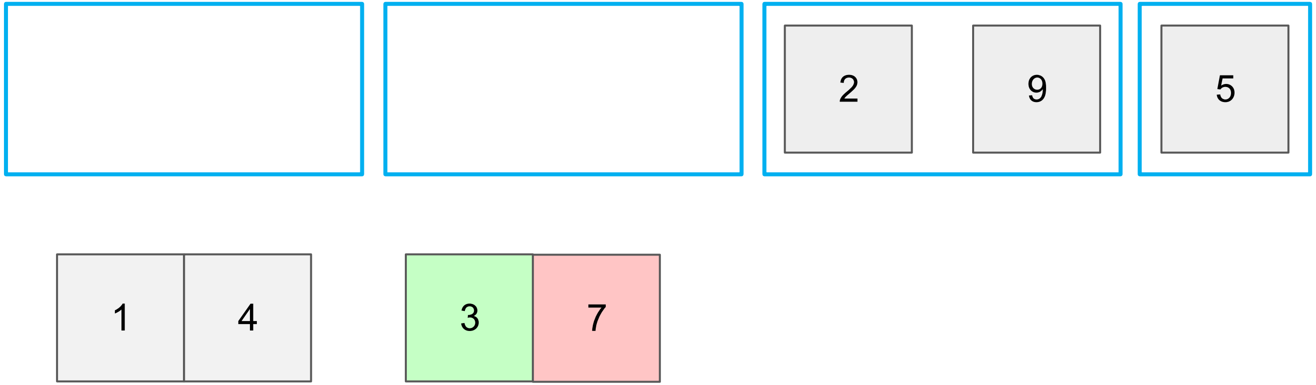
MergeSort



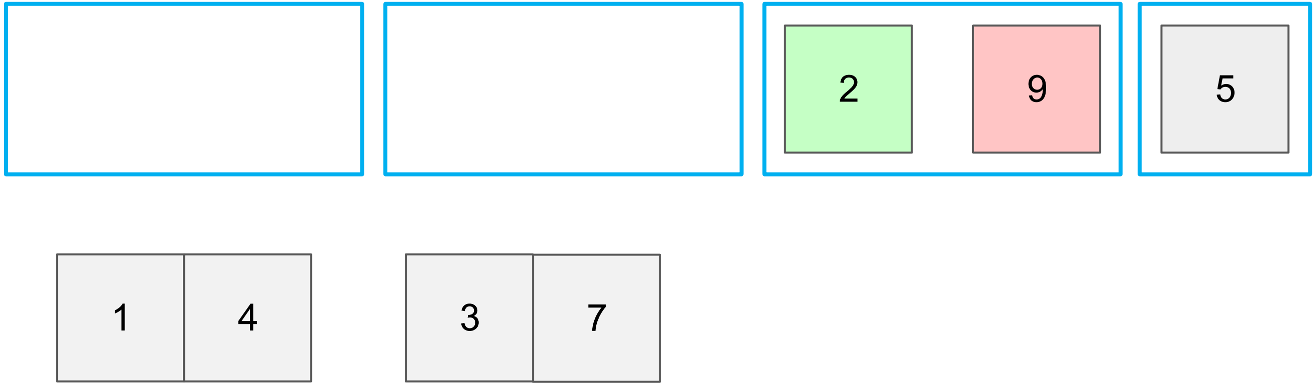
MergeSort



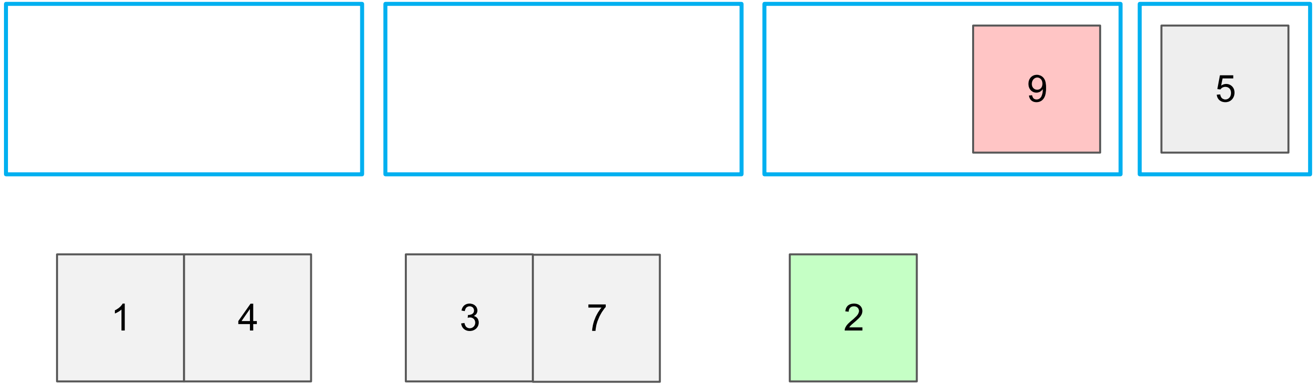
MergeSort



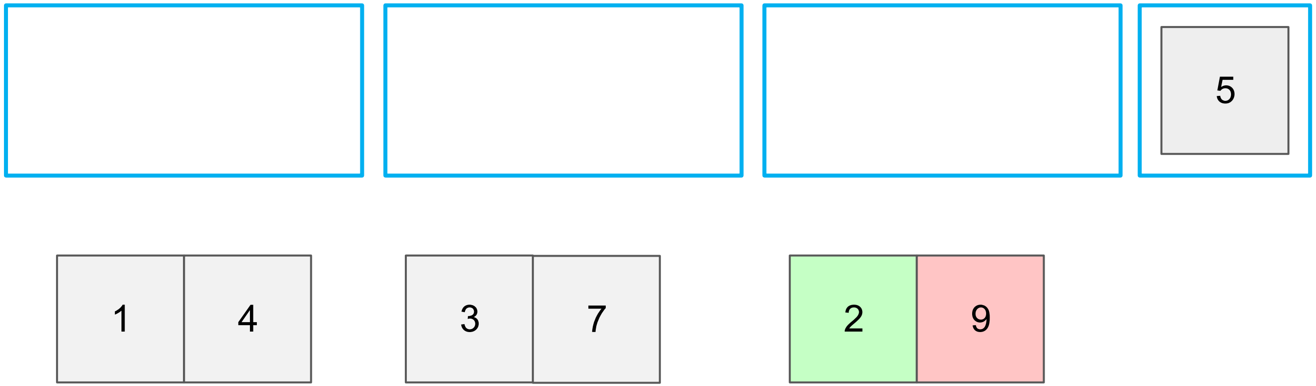
MergeSort



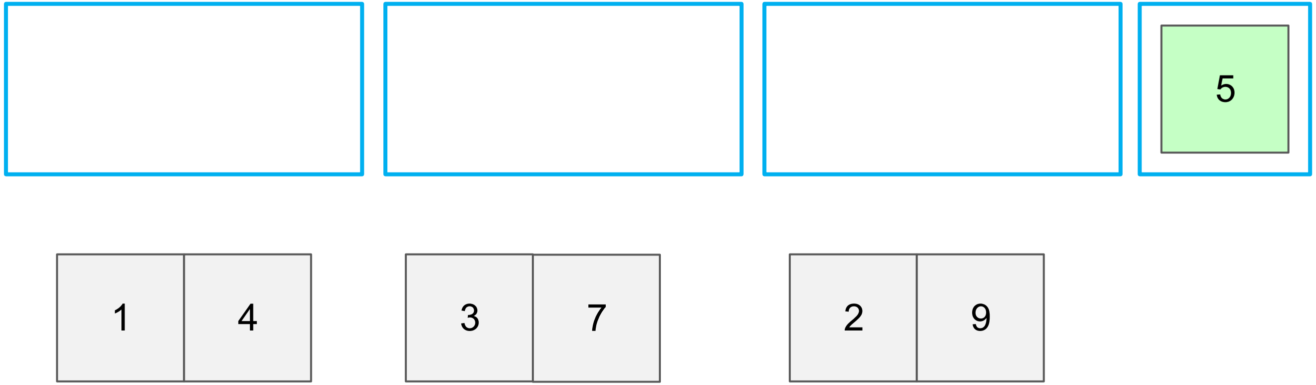
MergeSort



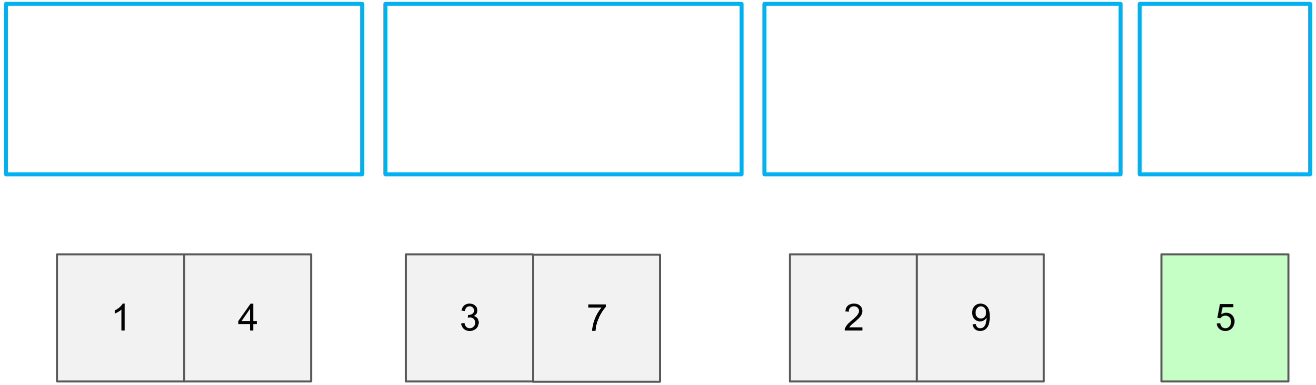
MergeSort



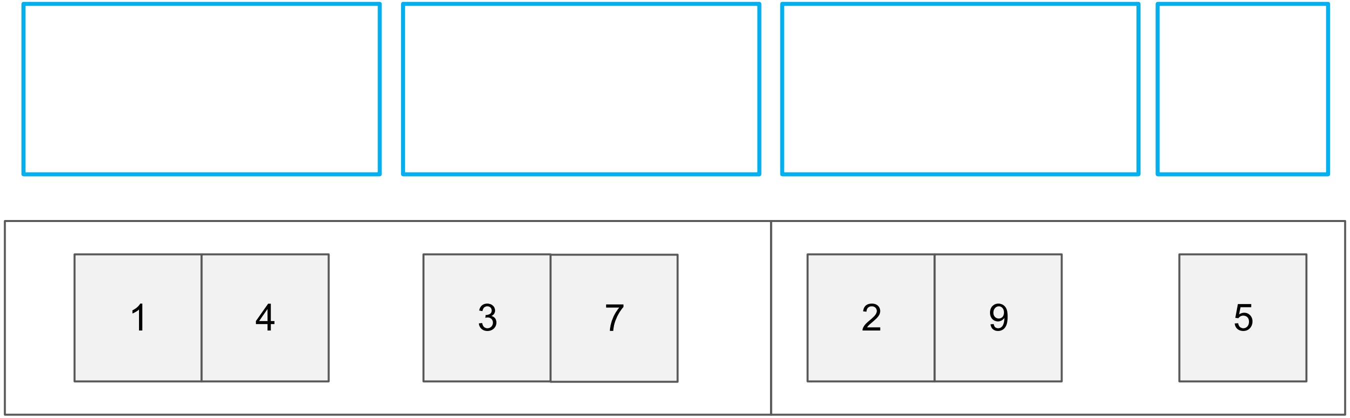
MergeSort



MergeSort

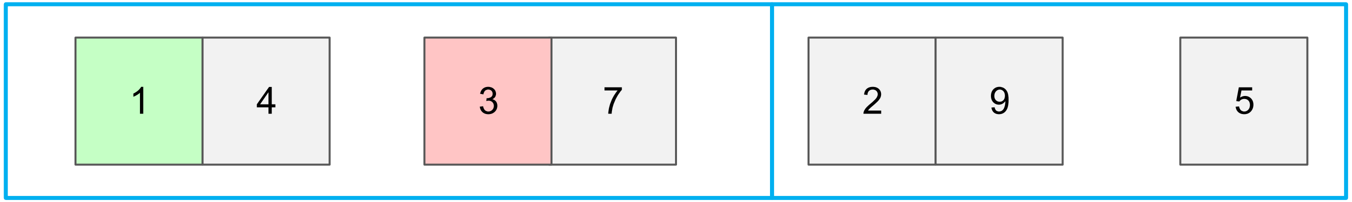


MergeSort



Combinamos los siguientes subconjuntos de la misma forma.

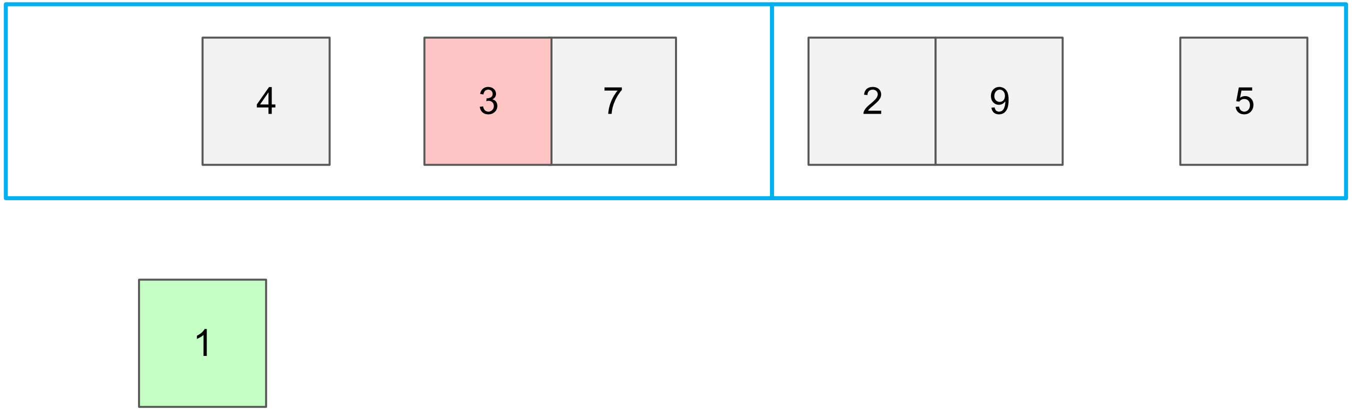
MergeSort



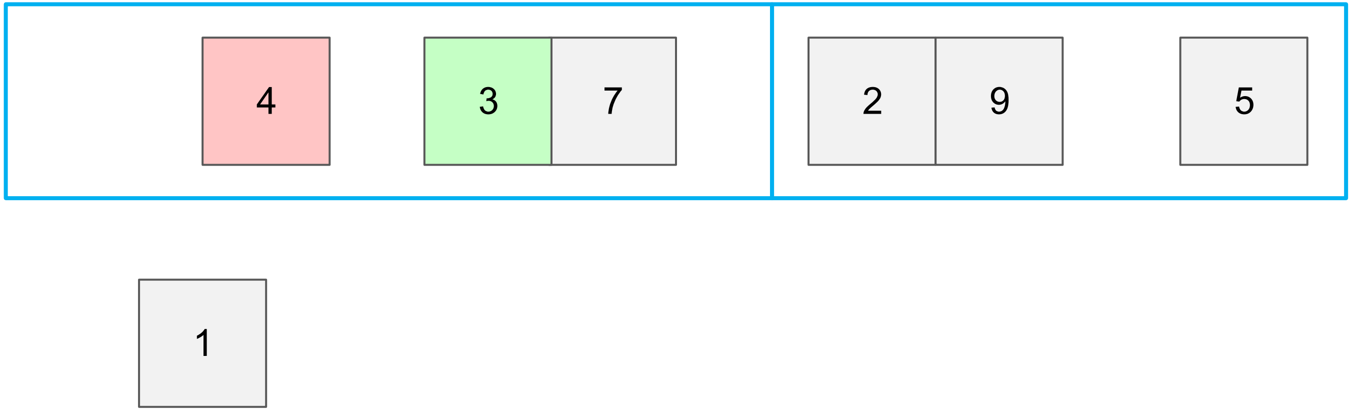
Recuerda, para combinar:

- Se compara el primer elemento de la izquierda con el primero de la derecha y se escoge el menor hasta que una de las partes se quede sin elementos.
- Se añaden los elementos restantes de la otra parte.

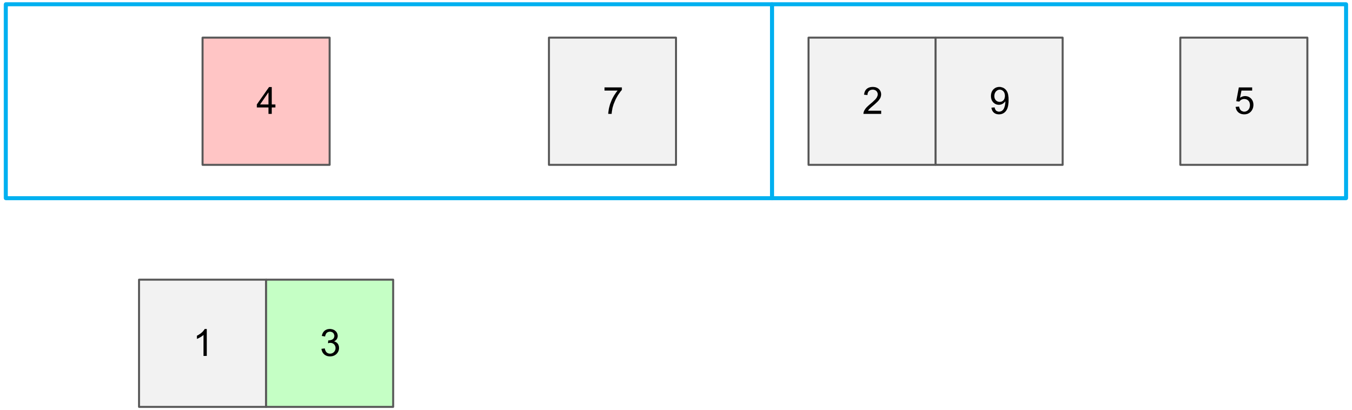
MergeSort



MergeSort

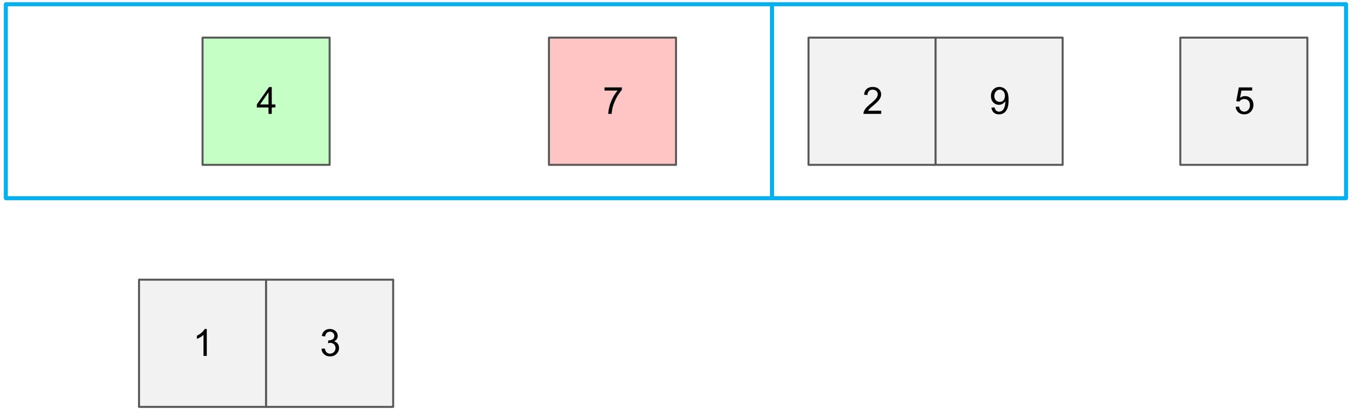


MergeSort

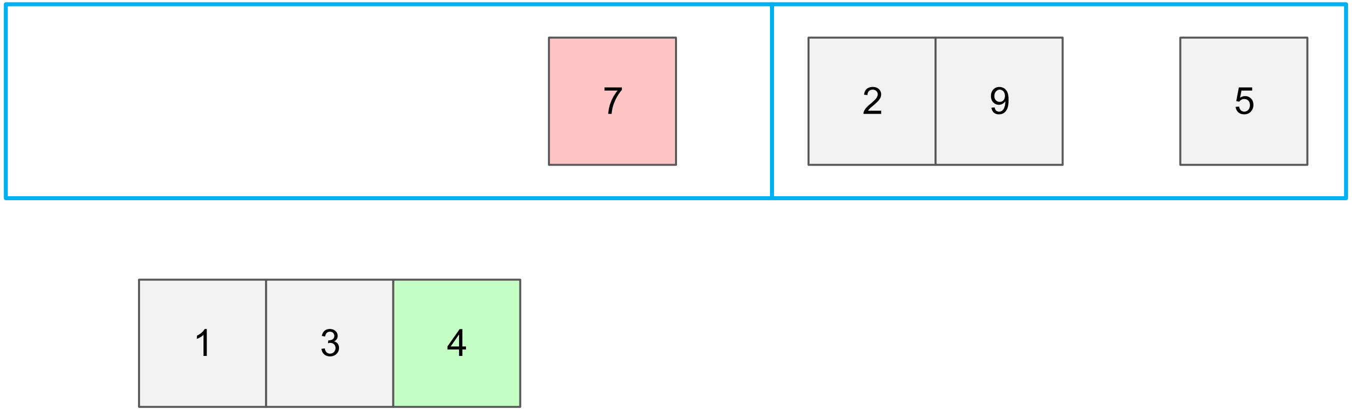


MergeSort

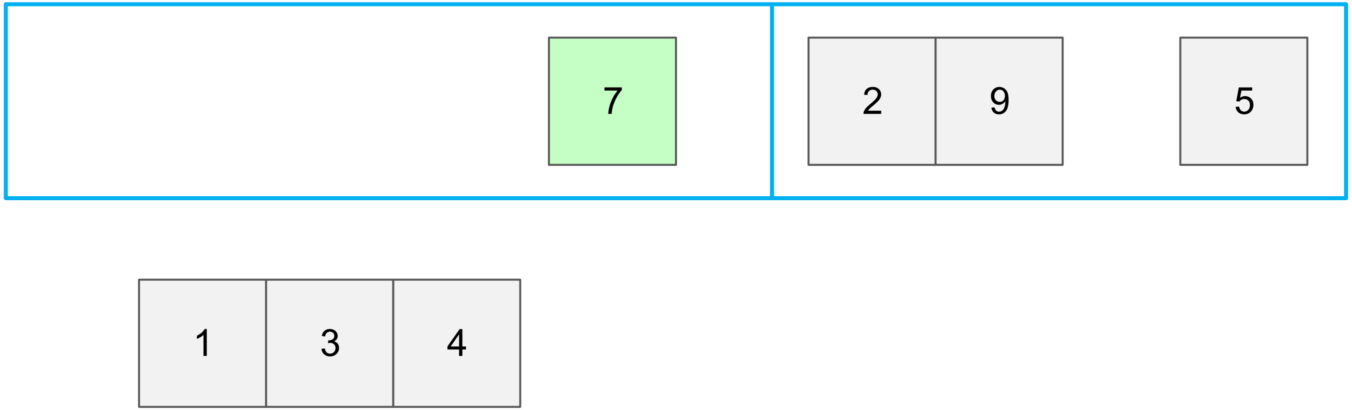
Universidad de Alcalá



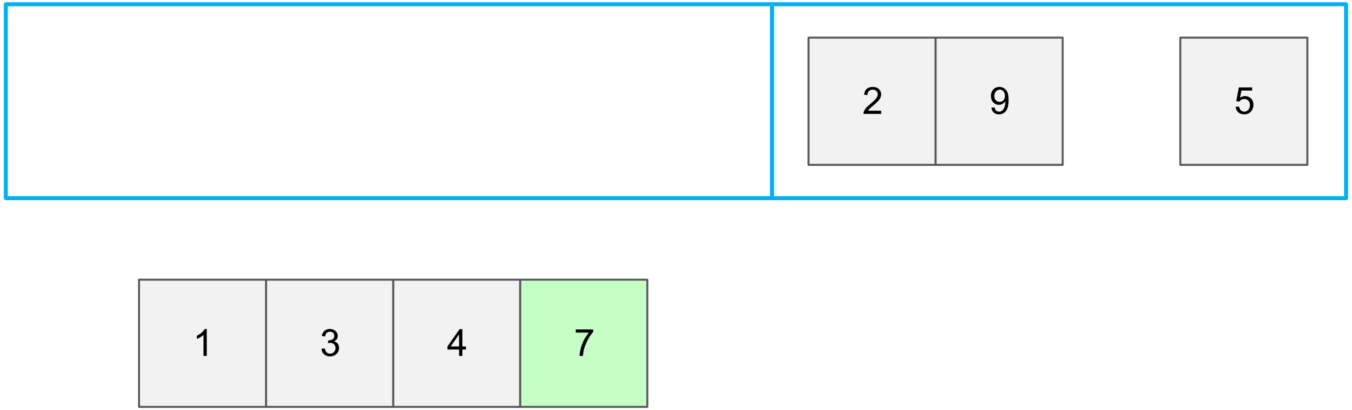
MergeSort



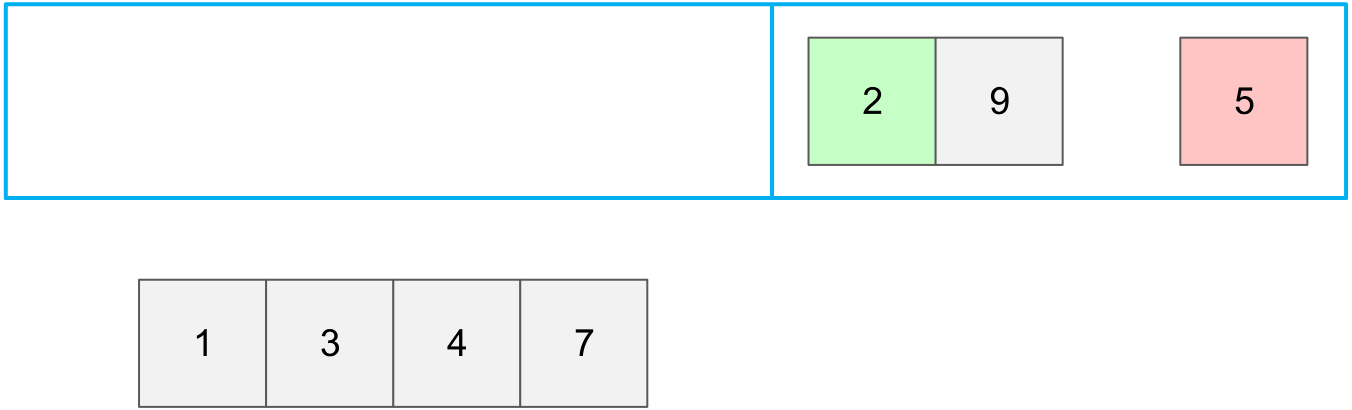
MergeSort



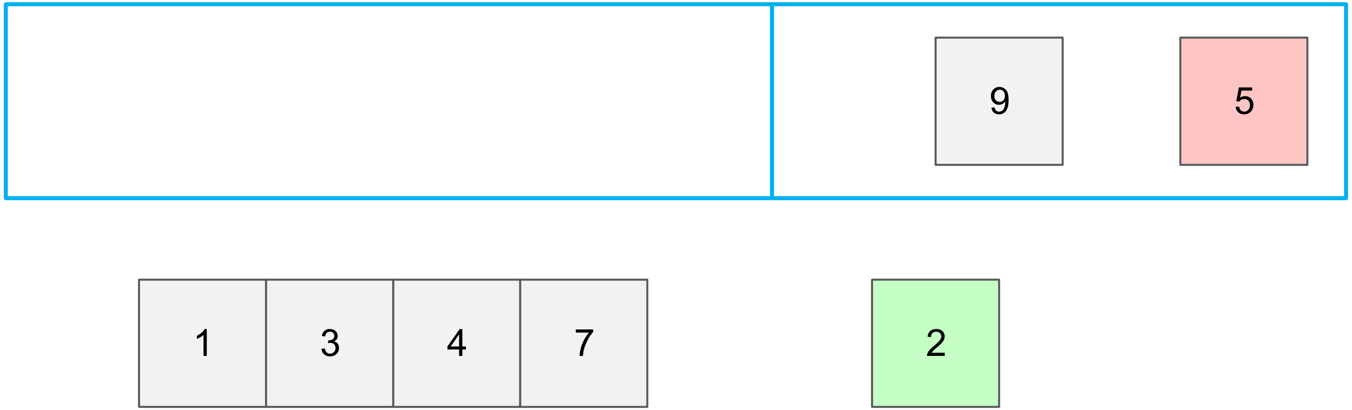
MergeSort



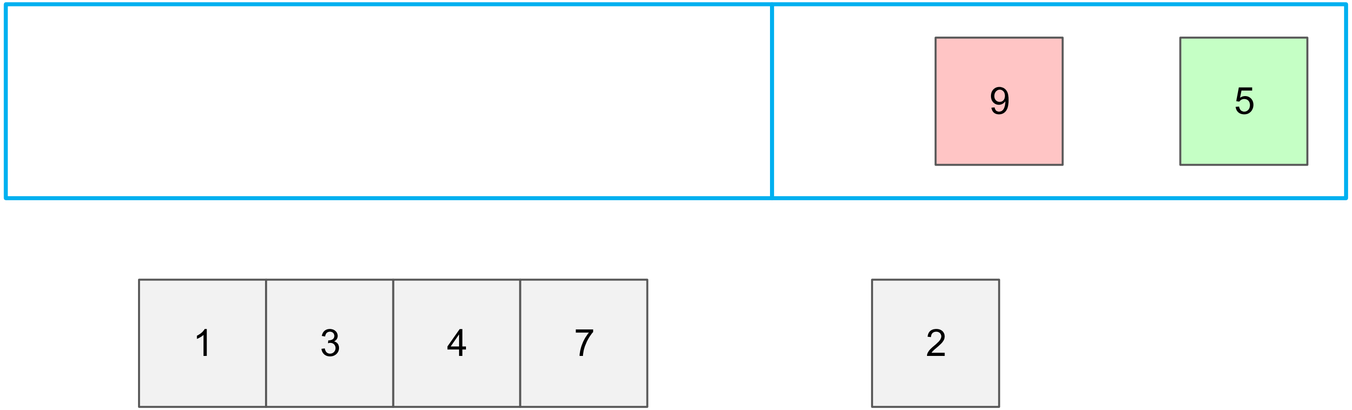
MergeSort



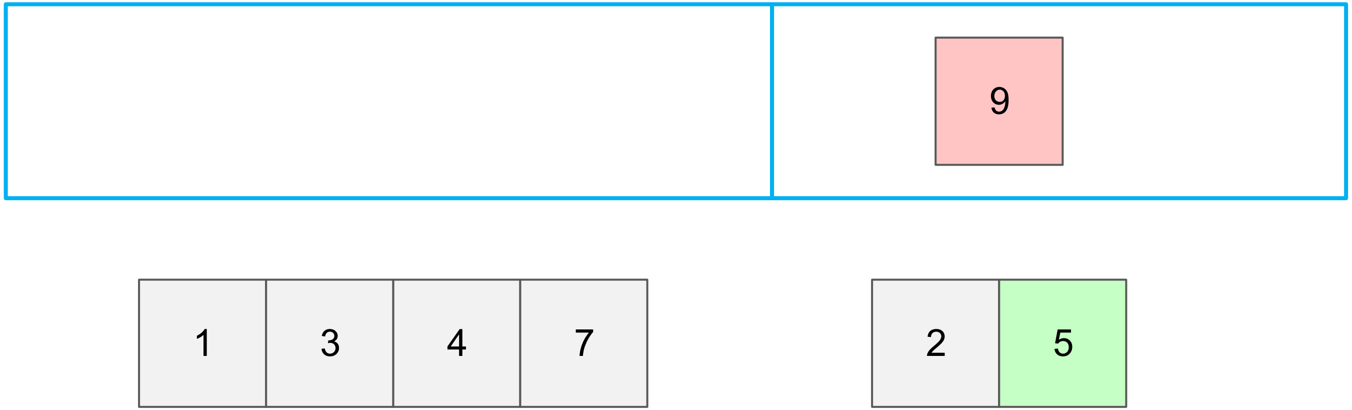
MergeSort



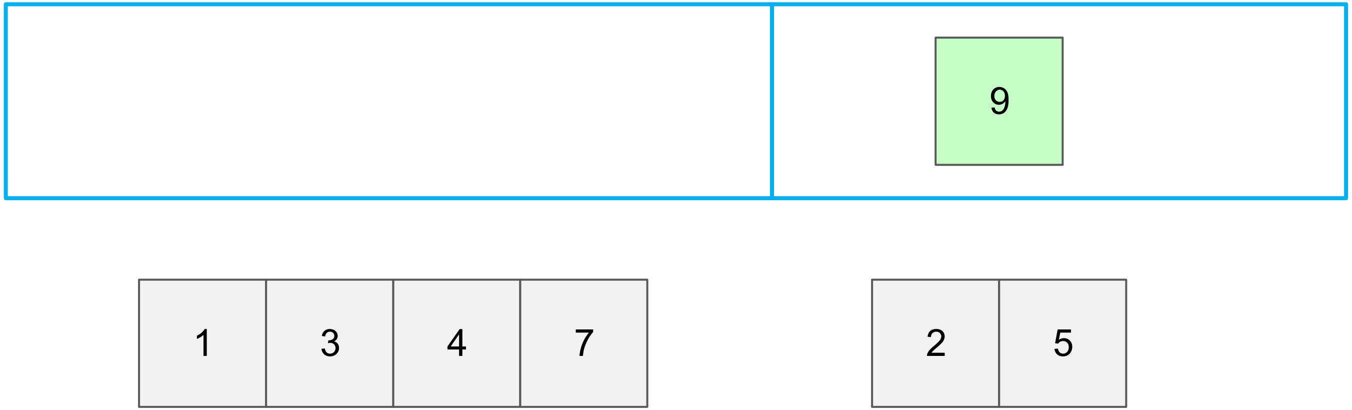
MergeSort



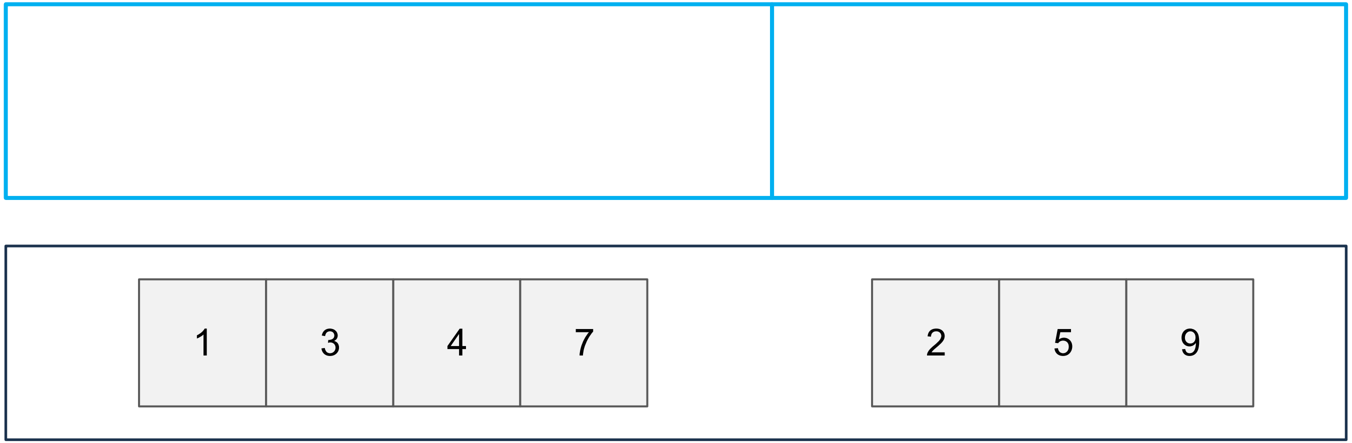
MergeSort



MergeSort



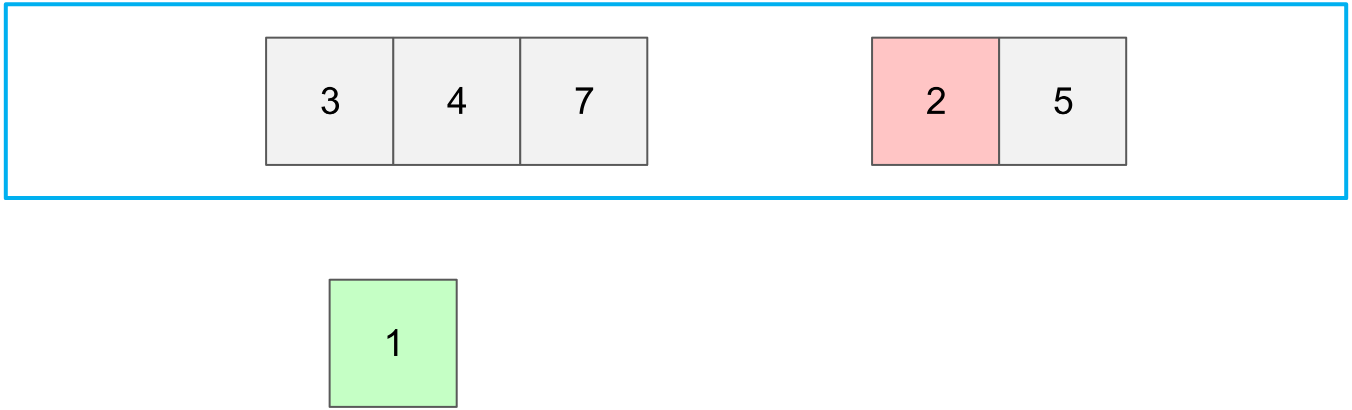
MergeSort



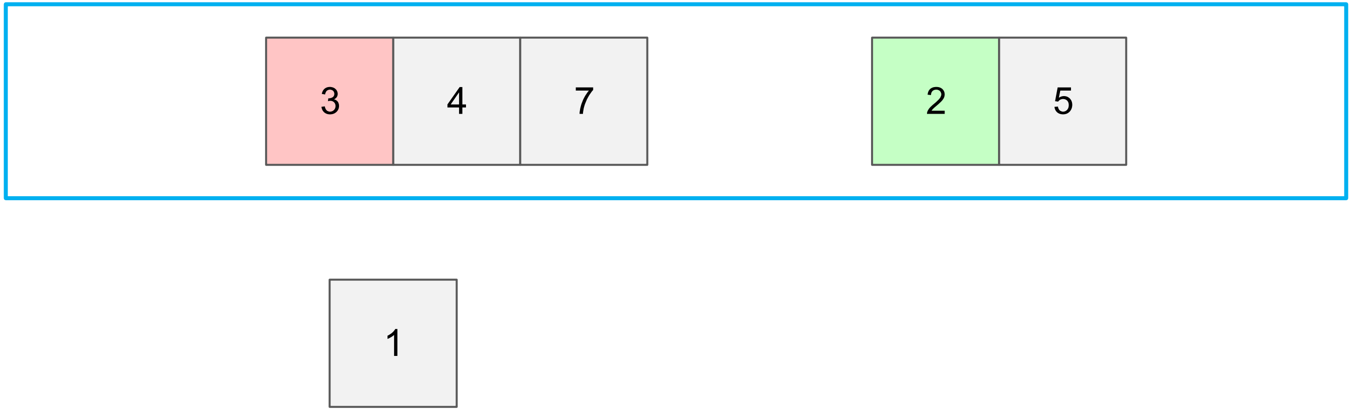
Combinamos el último conjunto de la misma forma.

MergeSort

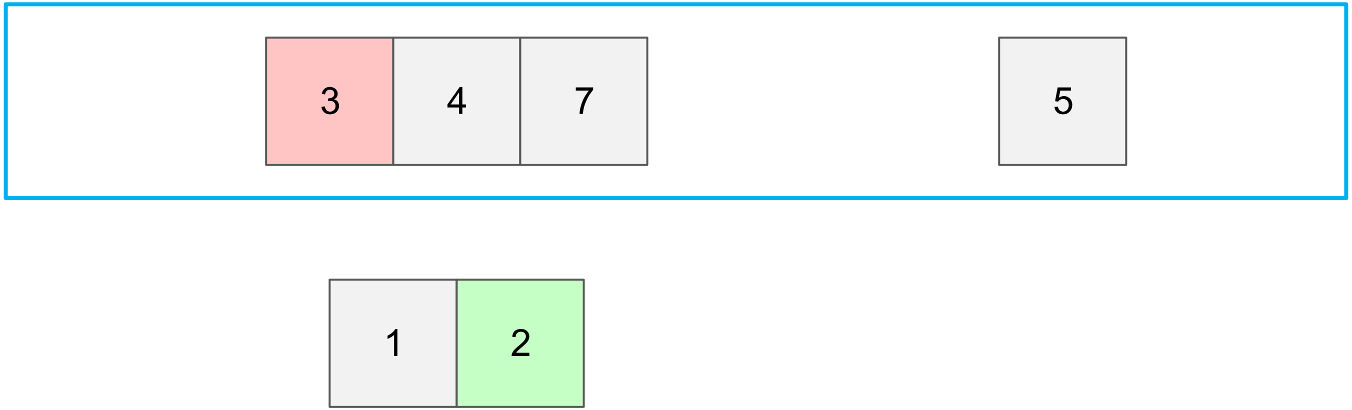
MergeSort



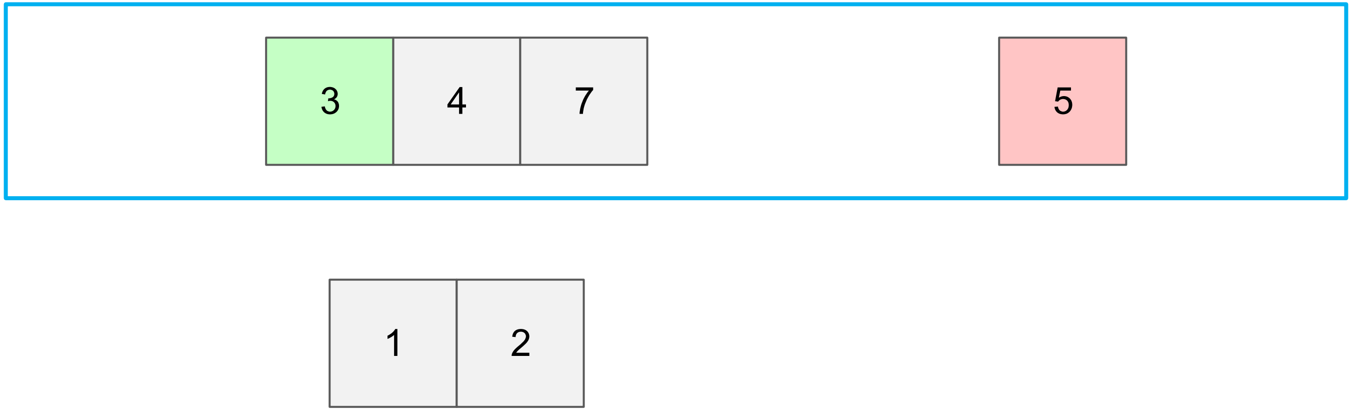
MergeSort



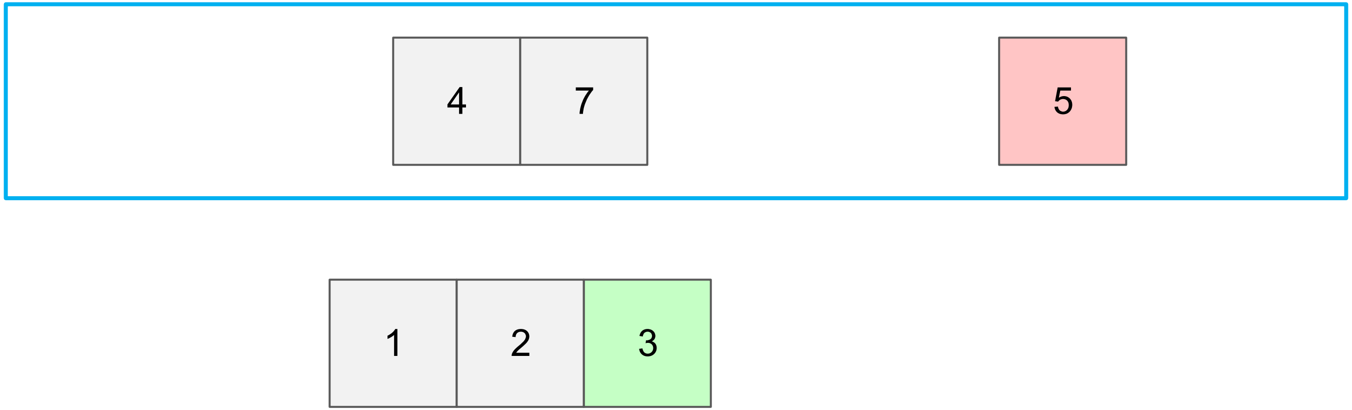
MergeSort



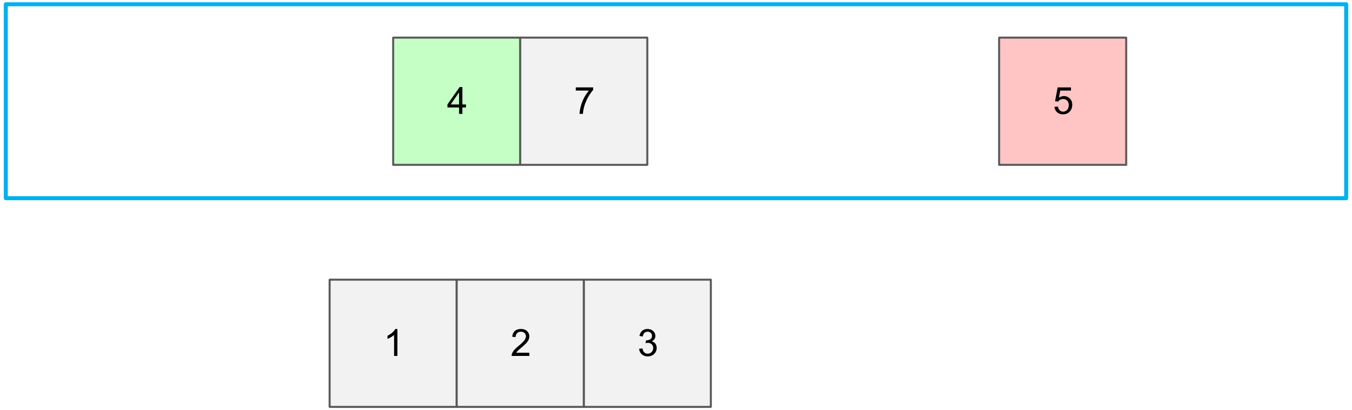
MergeSort



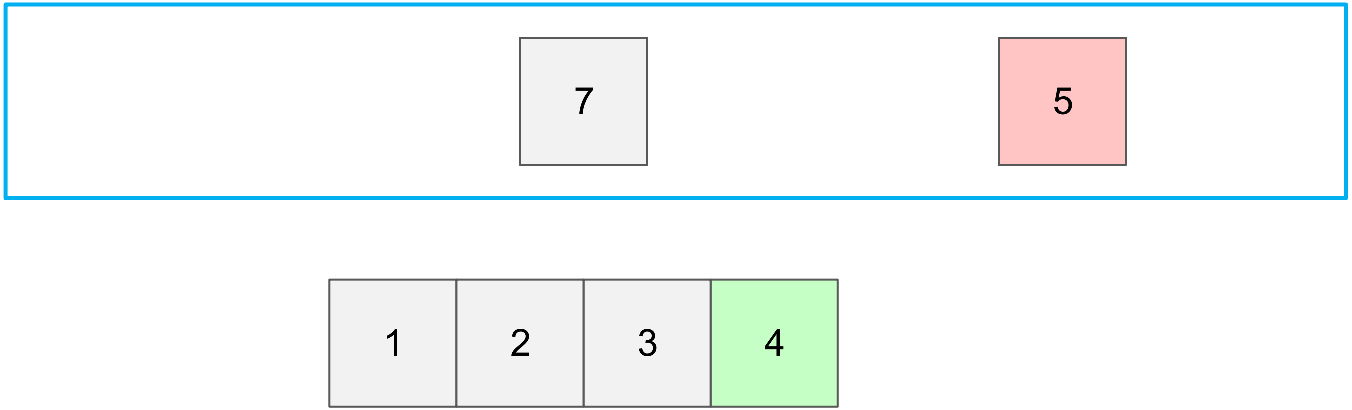
MergeSort



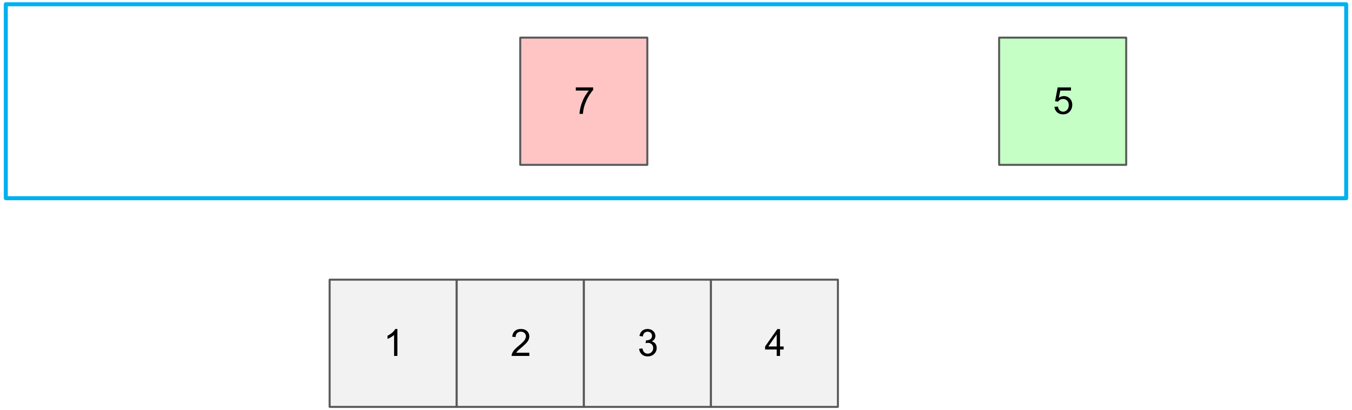
MergeSort



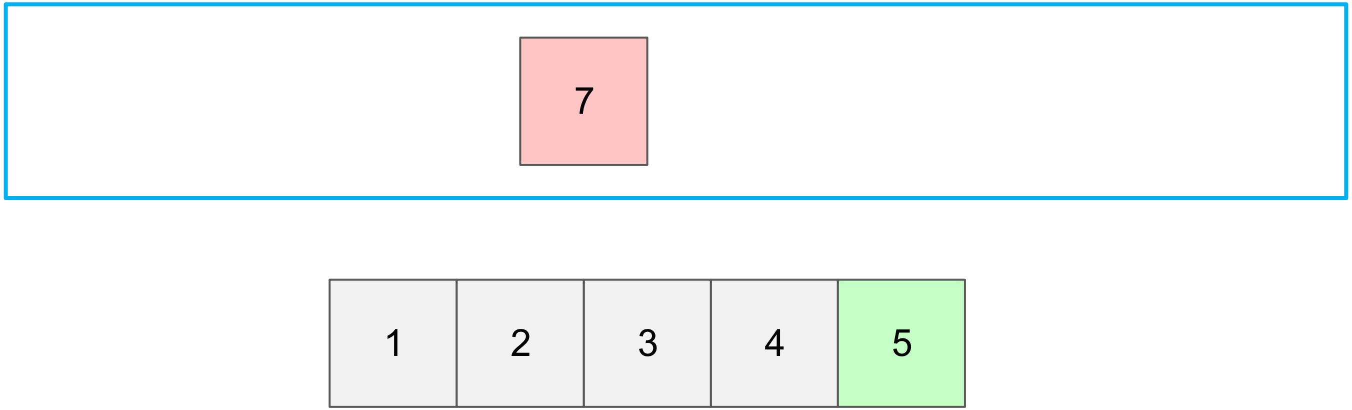
MergeSort



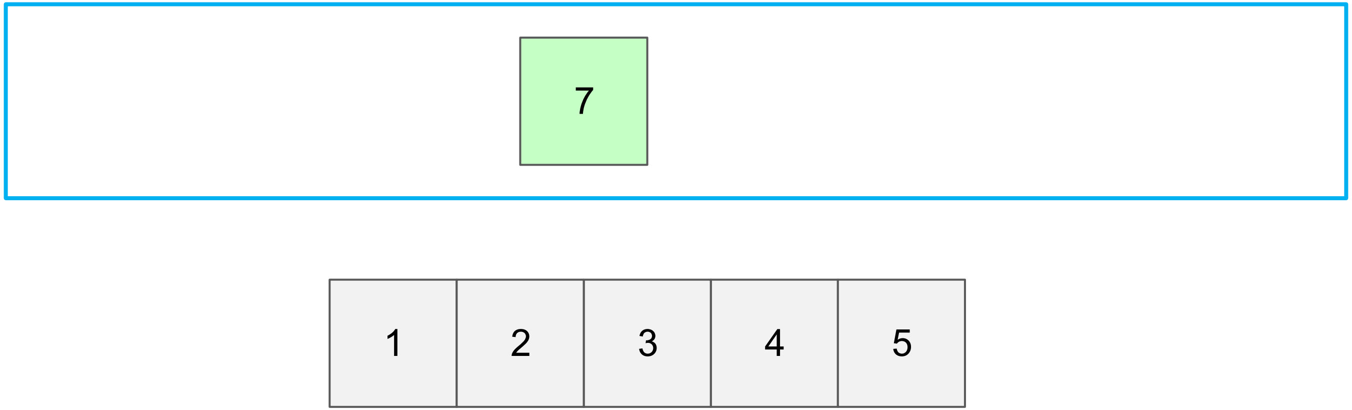
MergeSort



MergeSort



MergeSort



MergeSort

1	2	3	4	5	7

MergeSort

Universidad de Alcalá

```python
def merge_sort(array :list) -> list:

    # Si el array solo tiene un elemento
    n = len(array)
    if n <= 1:
        return array
    # Dividir el array en dos mitades y resolver recursivamente
    mid = n // 2
    left = merge_sort(array[:mid])
    right = merge_sort(array[mid:])
    # Combinar los arrays
    sorted_array = []
    while len(left) > 0 and len(right) > 0:
        if left[0] <= right[0]:
            sorted_array.append(left[0])
            left = left[1:]
        else:
            sorted_array.append(right[0])
            right = right[1:]
    # Añadir los elementos restantes
    sorted_array += left + right
    return sorted_array
```

MergeSort

Complejidad (teorema maestro):

$$T_2(n) = \begin{cases} g(n) & \text{si } 0 \le n < c \\ a \cdot T_2(n/c) + b \cdot n^k & \text{si } c \le n \end{cases} \Rightarrow T_2(n) \in \begin{cases} \Theta(n^k) & \text{si } a < c^k \\ \Theta(n^k \cdot \log n) & \text{si } a = c^k \\ \Theta(n^{\log_c a}) & \text{si } a > c^k \end{cases}$$

- $a = 2$ → Porque siempre se hacen dos llamadas recursivas.
- $c = 2$ → Porque dividimos el tamaño del problema entre 2.
- $k = 1$ → Porque combinar tiene un coste lineal.

$$T(n) \in O(n^1 \log(n))$$

QuickSort

Pasos:

1. Se escoge un elemento del array como pivote.
2. Los elementos mayores que el pivote se almacenan a la derecha y los menores a la izquierda.
3. Se combina la solución de ordenar la parte izquierda (usando el mismo algoritmo de forma recursiva), añadir el pivote y ordenar la parte derecha (de la misma forma).

4	1	7	3	2	9	5

QuickSort

Pasos:

1. Se escoge un elemento del array como pivote.
2. Los elementos mayores que el pivote se almacenan a la derecha y los menores a la izquierda.
3. Se combina la solución de ordenar la parte izquierda (usando el mismo algoritmo de forma recursiva), añadir el pivote y ordenar la parte derecha (de la misma forma).

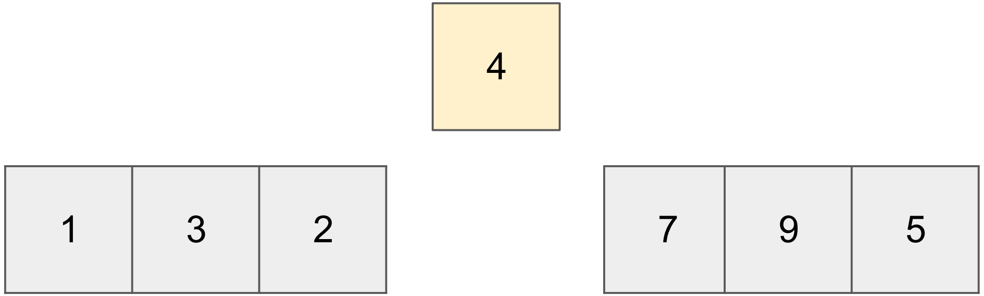
4	1	7	3	2	9	5

QuickSort

4	1	7	3	2	9	5

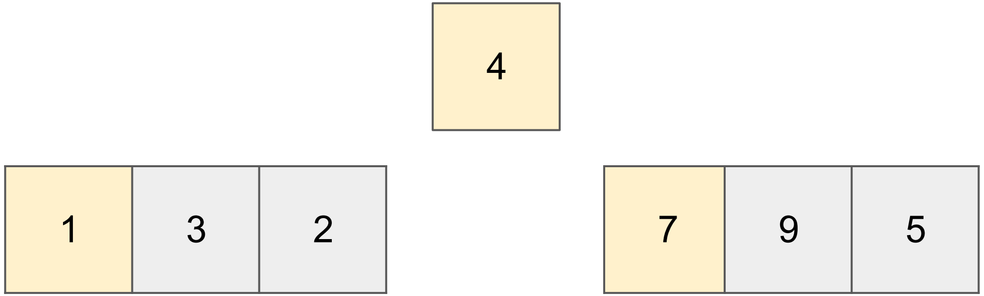
Se escoge un elemento del array como pivote.

QuickSort



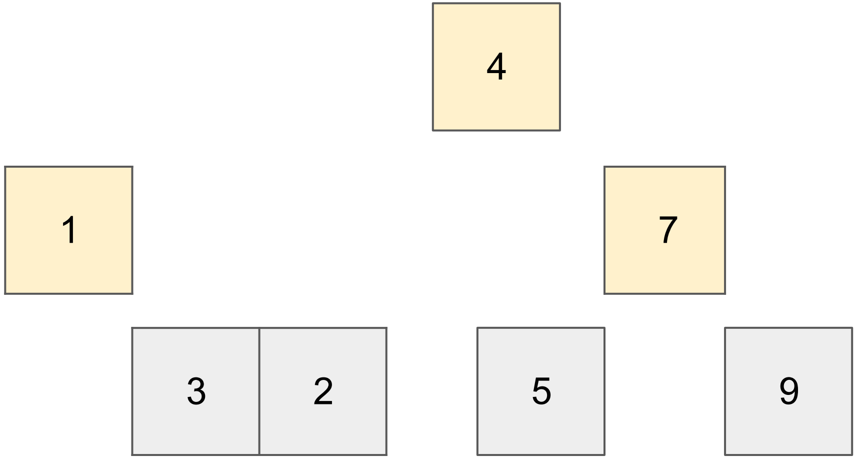
Se colocan los menores a la izquierda y los mayores a la derecha.

QuickSort



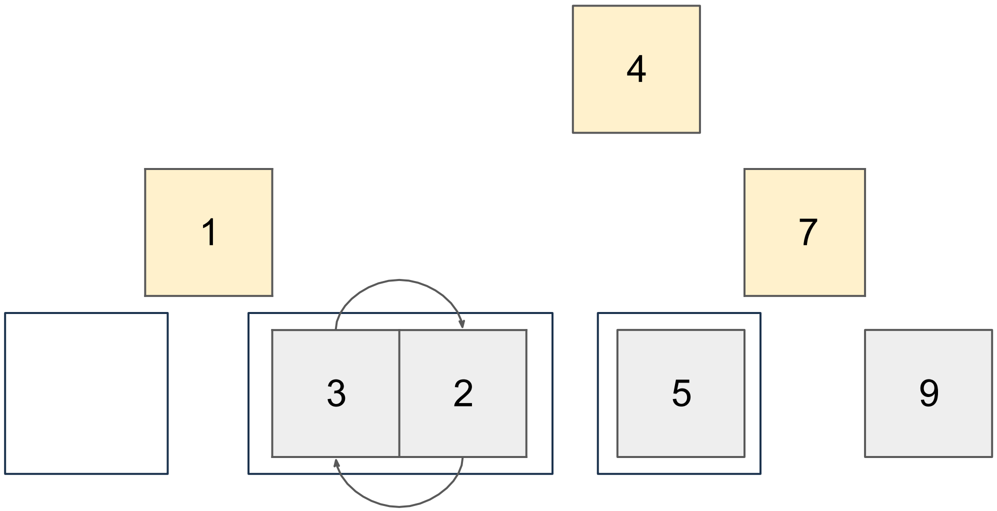
Se repite el proceso con cada parte.

QuickSort



Se repite el proceso con cada parte.

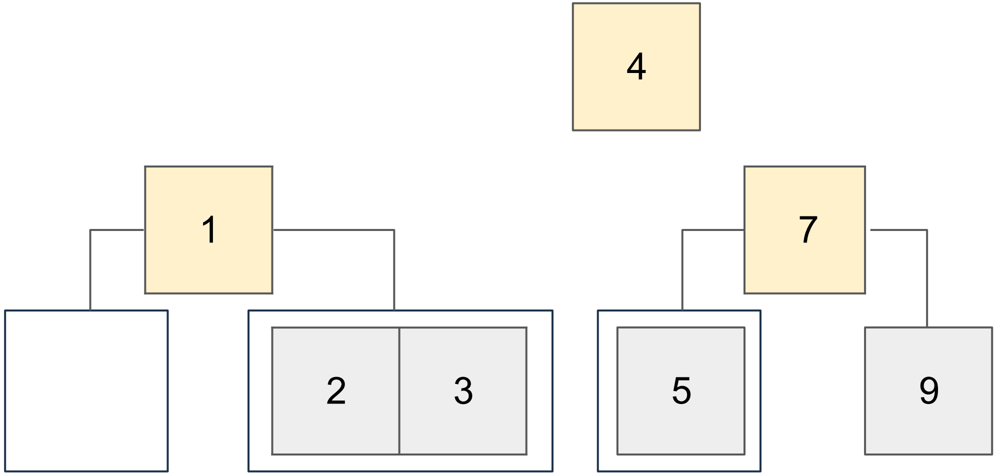
QuickSort



Hasta llegar a un caso base.

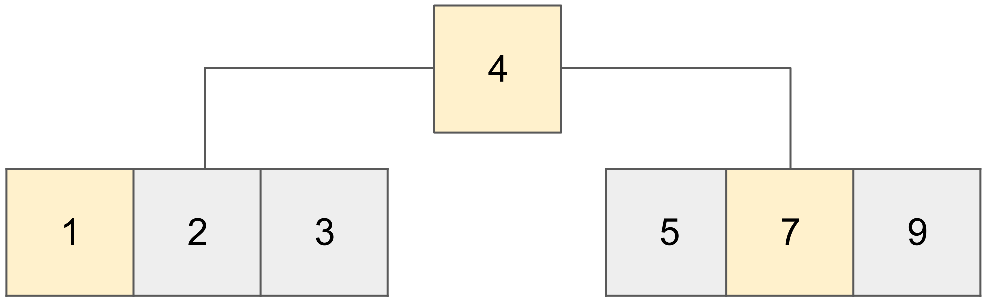
a) Conjunto vacío → [ ]
b) Conjunto con un elemento → [a]
c) Conjunto con dos elementos → [a, b] (en orden)

QuickSort



Se combinan a la vuelta de la recursión.

QuickSort



Se combinan a la vuelta de la recursión.

QuickSort

1	2	3	4	5	7	9

QuickSort


```python
def quicksort(array :list) -> list:

    # Si el array solo tiene un elemento
    if len(array) <= 1:
        return array

    # Seleccionar el pivote
    pos = random.randint(0, len(array) - 1)
    pivote = array[pos]

    # Dividir el array en dos partes
    left = []
    right = []
    for num in array[:pos] + array[pos + 1:]:
        if num <= pivote:
            left.append(num)
        else:
            right.append(num)

    # Resolver recursivamente y combinar las soluciones
    return quicksort(left) + [pivote] + quicksort(right)
```

QuickSort

Complejidad (teorema maestro):

$$T_2(n) = \begin{cases} g(n) & \text{si } 0 \le n < c \\ a \cdot T_2(n/c) + b \cdot n^k & \text{si } c \le n \end{cases} \Rightarrow T_2(n) \in \begin{cases} \Theta(n^k) & \text{si } a < c^k \\ \Theta(n^k \cdot \log n) & \text{si } a = c^k \\ \Theta(n^{\log_c a}) & \text{si } a > c^k \end{cases}$$

- $a = 2$ → Porque siempre se hacen dos llamadas recursivas.
- $c = 2$ → Porque dividimos el tamaño del problema entre 2.
- $k = 1$ → Porque dividir el array en dos partes tiene coste lineal.

$$T(n) \in O(n^1 \log(n))$$