ESTRATEGIAS DE PROGRAMACIÓN

Algoritmos y Programación

Javier Miranda

Escuela de Ingeniería Informática
Universidad de Las Palmas de Gran Canaria

18 de octubre de 2023

Estrategias

- Fuerza bruta (*brute force*)
- Vuelta atrás (*backtracking*)
- Voráz (*greedy*)

- Divide y vencerás
  - Reduce y vencerás

Técnica

- Programación Dinámica

Estrategia Divide y Vencerás

A divide-and-conquer algorithm recursively breaks down a problem into two or more sub-problems of the same or related type, until these become simple enough to be solved directly.

The solutions to the sub-problems are then combined to give a solution to the original problem.

- *Para demostrar que el algoritmo es correcto se utiliza inducción matemática*
- *Para analizar su coste se resuelve la recurrencia (lo veremos en otro tema)*

https://en.wikipedia.org/wiki/Divide-and-conquer_algorithm

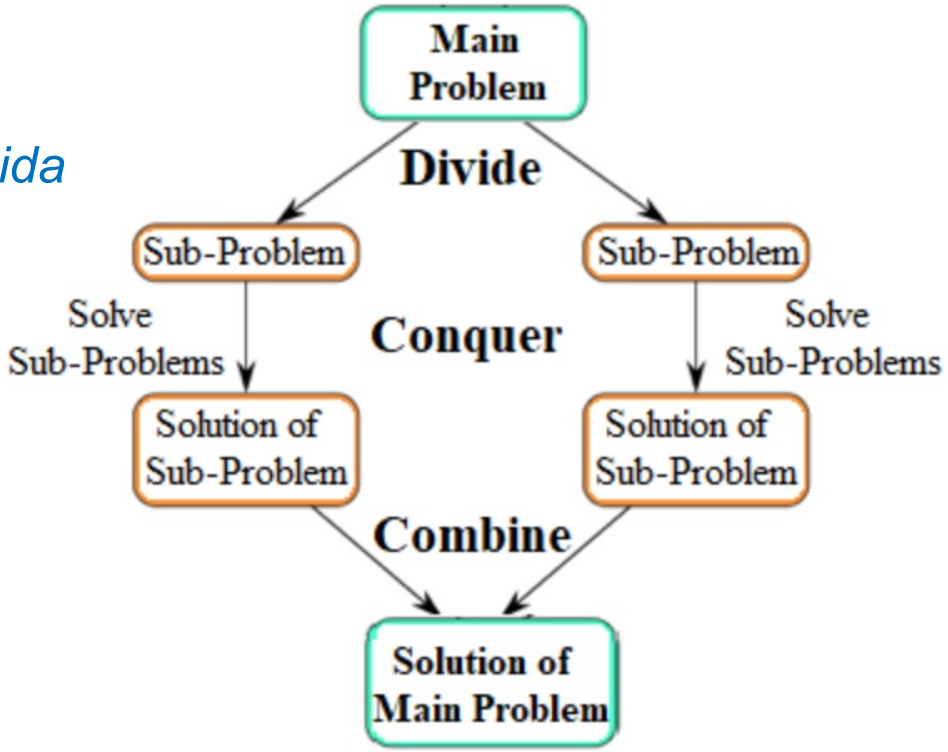
Estrategia Divide y Vencerás

- Consta de 3 pasos:

1. **Dividir** el ejemplar en dos o más subproblemas
2. **Resolver** recursivamente los subproblemas
3. **Combinar** las soluciones para obtener la solución completa

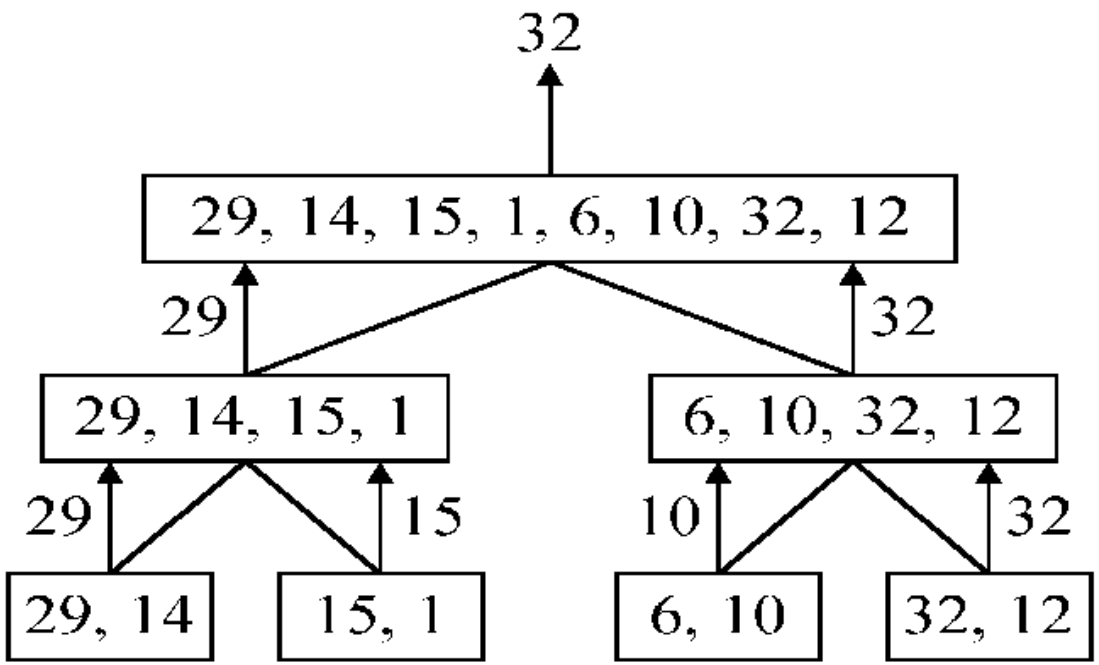
Probablemente la estrategia más conocida

Generalmente se programa de forma recursiva, pero puede programarse de forma iterativa



Ejemplo (1/3)

- Buscar el máximo de un conjunto de números



Características Comunes

1) Factor de reducción grande (1/2, 1/3, 1/4 …)

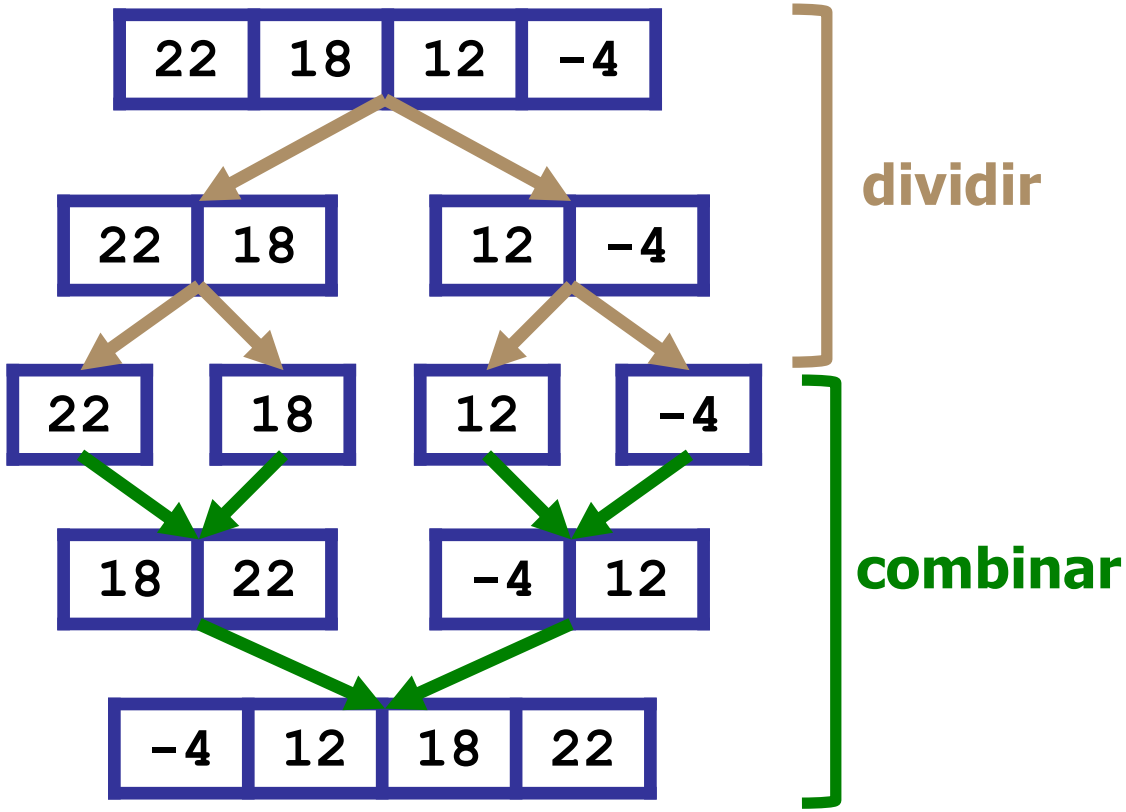
2) No hay solapamiento entre los subproblemas

3) Subproblemas independientes que pueden resolverse en paralelo

1945

Ejemplo (2/3): Merge Sort

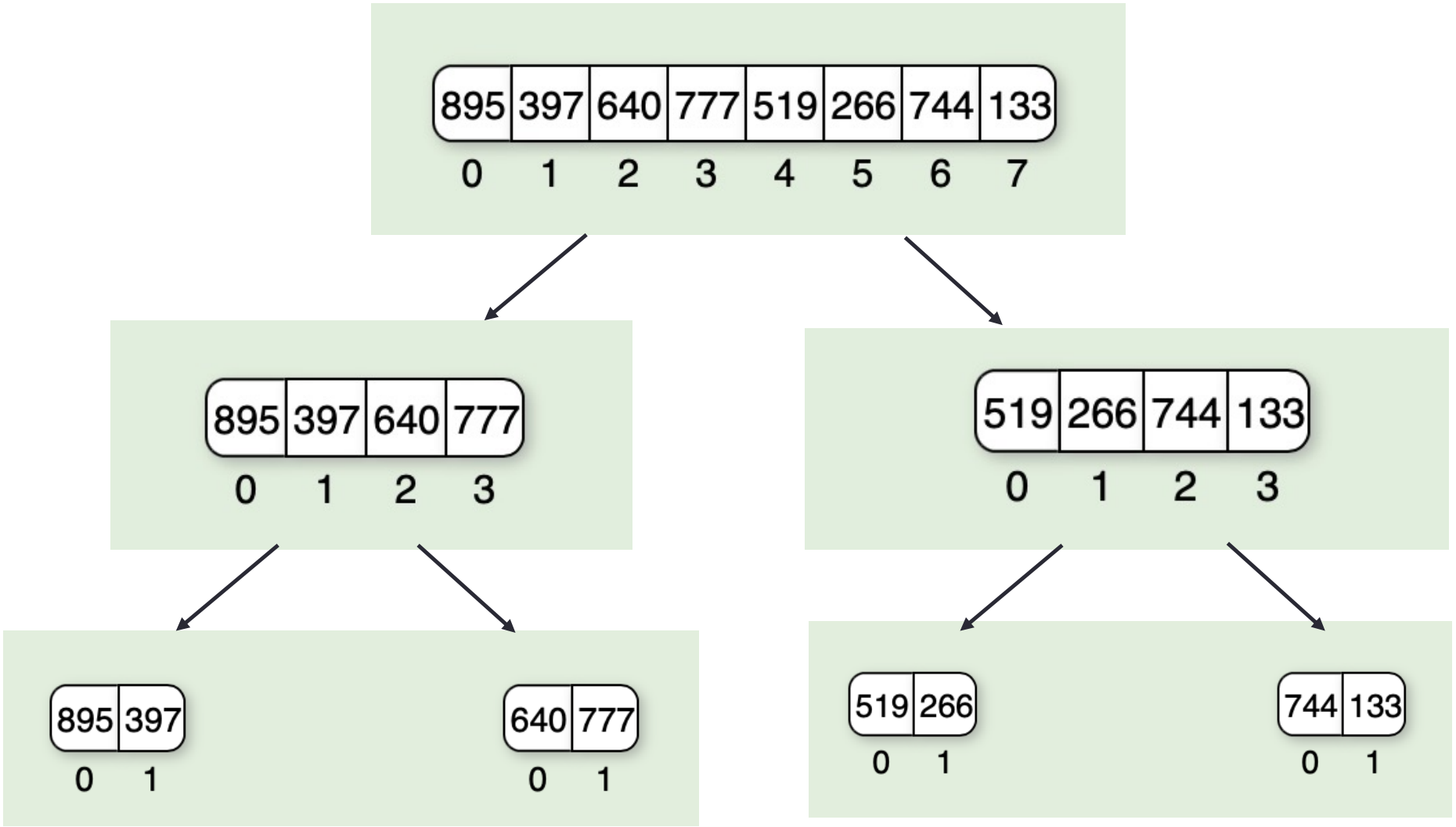
- Dividimos el array en dos partes
- Nos llamamos recursivamente para ordenar la primera mitad
- Nos llamamos recursivamente para ordenar la segunda mitad
- Unimos las dos partes ordenadas (*merge*)



John von Neumann

Ejemplo

Merge Sort: a) División



https://opendsa-server.cs.vt.edu/ODSA/Books/Everything/html/Mergesort.html

Pseudo-código

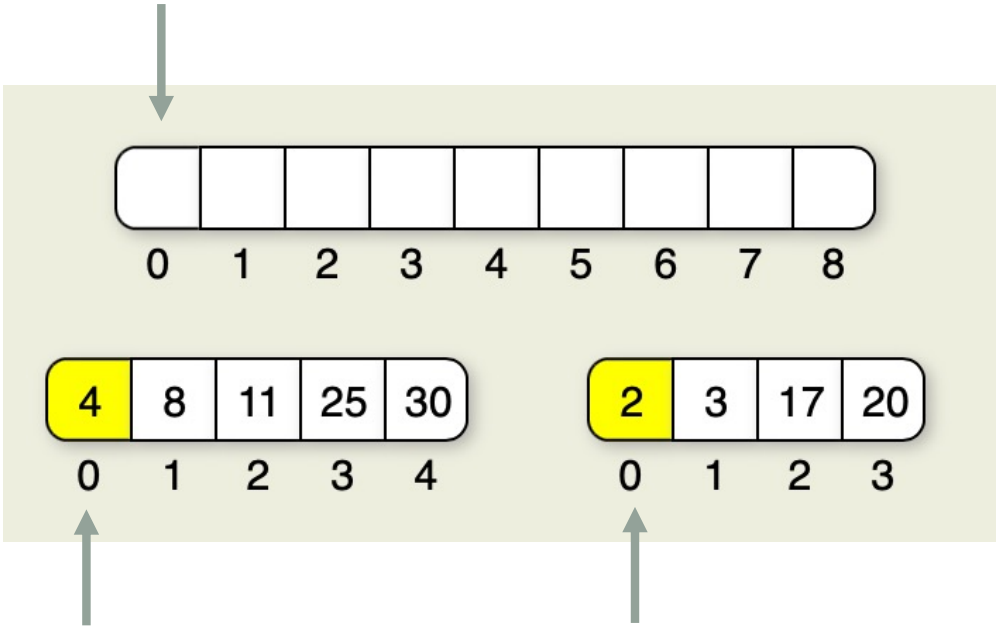
```
MergeSort (data, left, right) {
    if (left < right) {
       mid = divide (left, right)
       MergeSort (data, left, mid)
       MergeSort (data, mid+1, right)

       ...

    }
}
```

Fase de división

Fase de combinación

Ejemplo

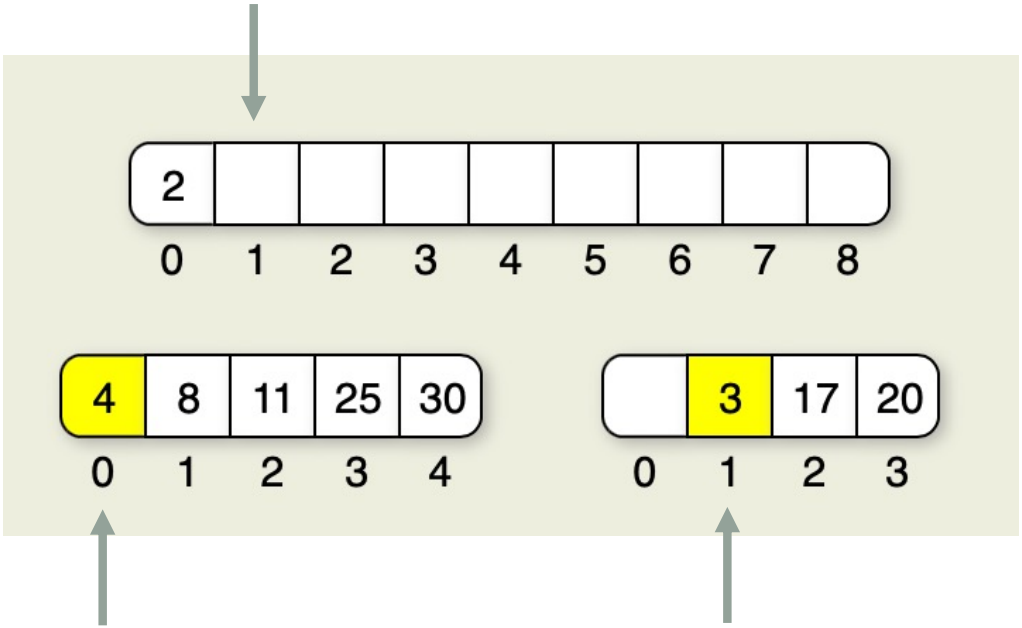
Merge Sort: b) Fusión (*merge*)



https://opendsa-server.cs.vt.edu/ODSA/Books/Everything/html/Mergesort.html

Ejemplo

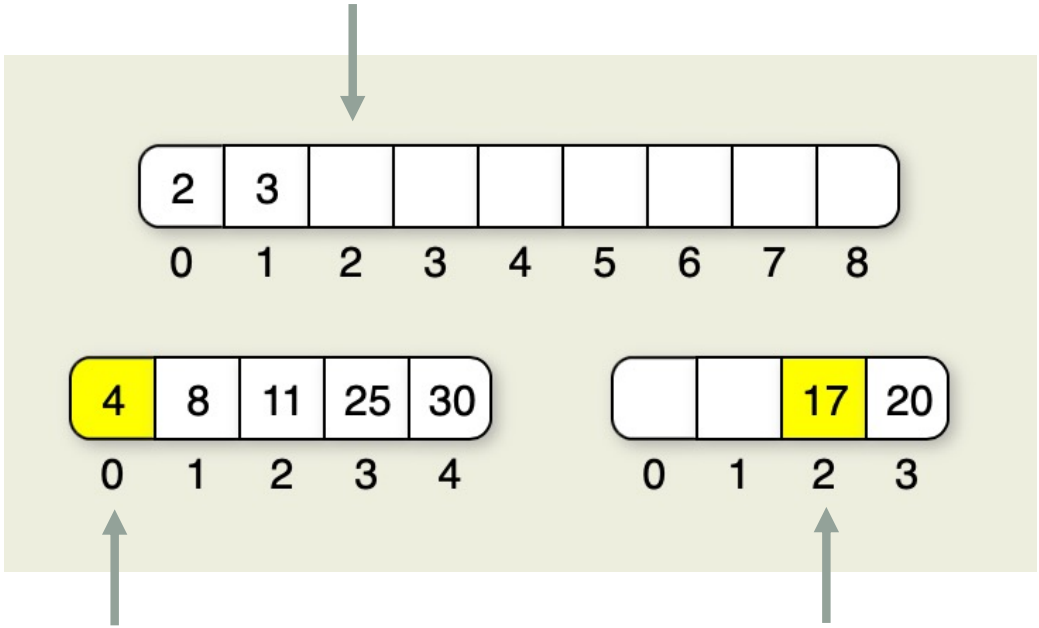
Merge Sort: b) Fusión (*merge*)



https://opendsa-server.cs.vt.edu/ODSA/Books/Everything/html/Mergesort.html

Ejemplo

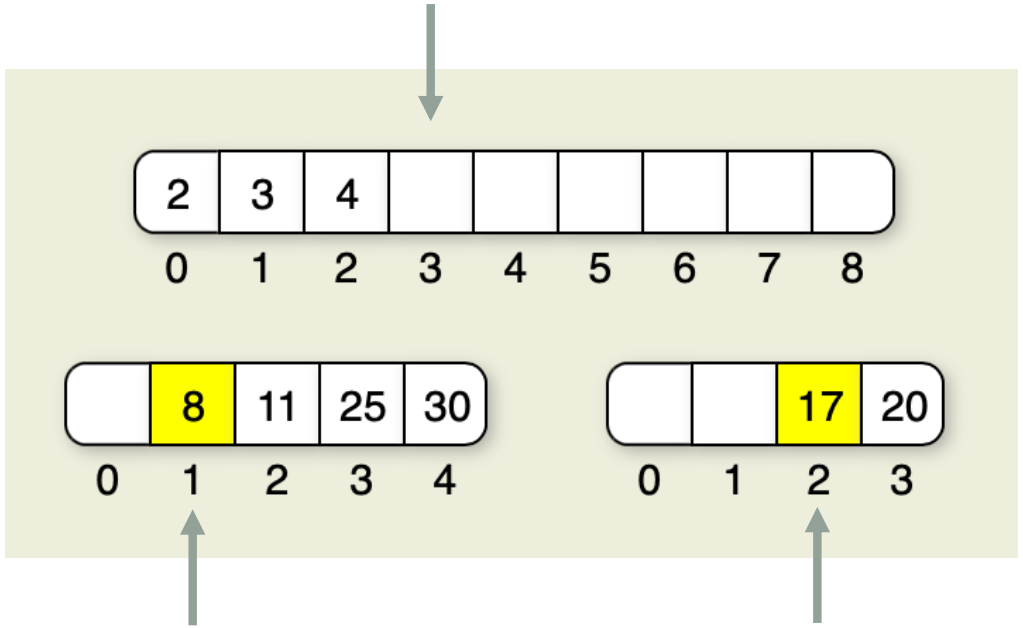
Merge Sort: b) Fusión (*merge*)



https://opendsa-server.cs.vt.edu/ODSA/Books/Everything/html/Mergesort.html

Ejemplo

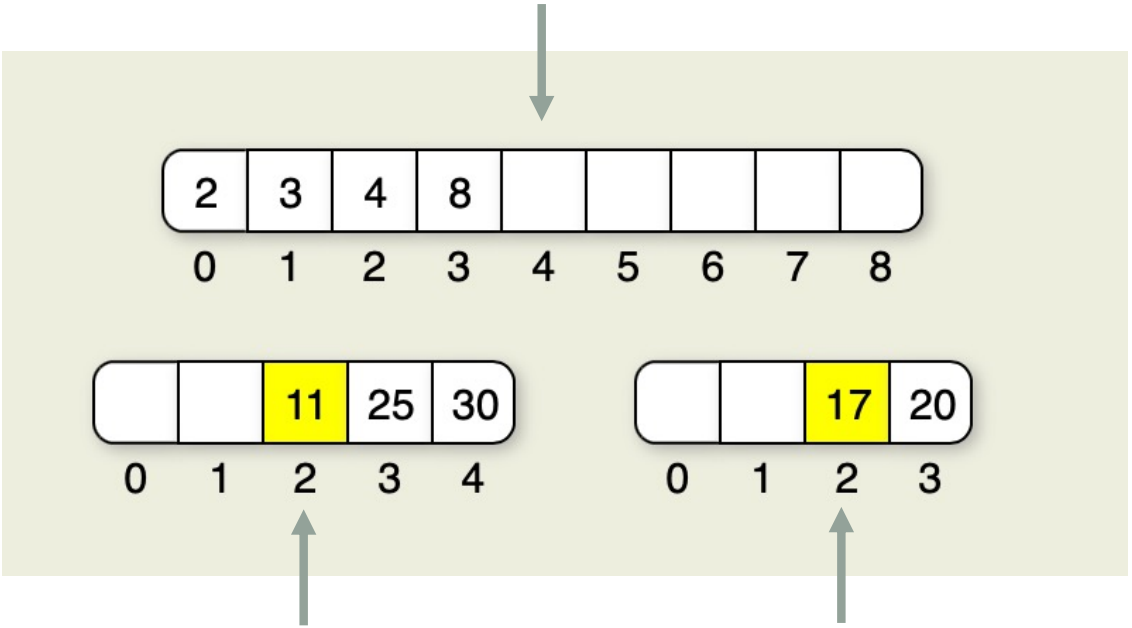
Merge Sort: b) Fusión (*merge*)



https://opendsa-server.cs.vt.edu/ODSA/Books/Everything/html/Mergesort.html

Ejemplo

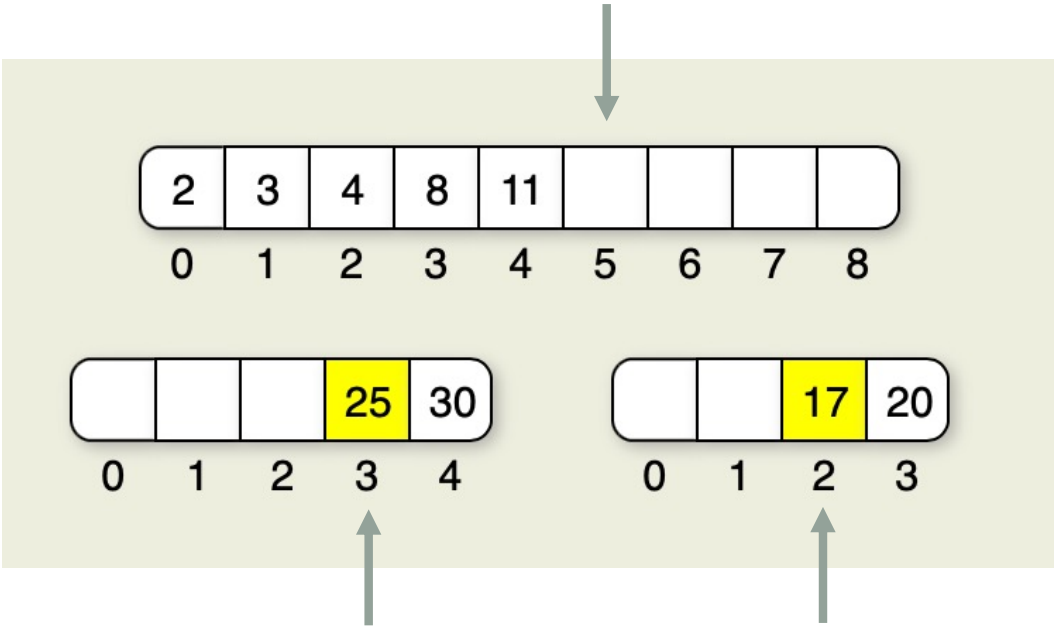
Merge Sort: b) Fusión (*merge*)



https://opendsa-server.cs.vt.edu/ODSA/Books/Everything/html/Mergesort.html

Ejemplo

Merge Sort: b) Fusión (*merge*)

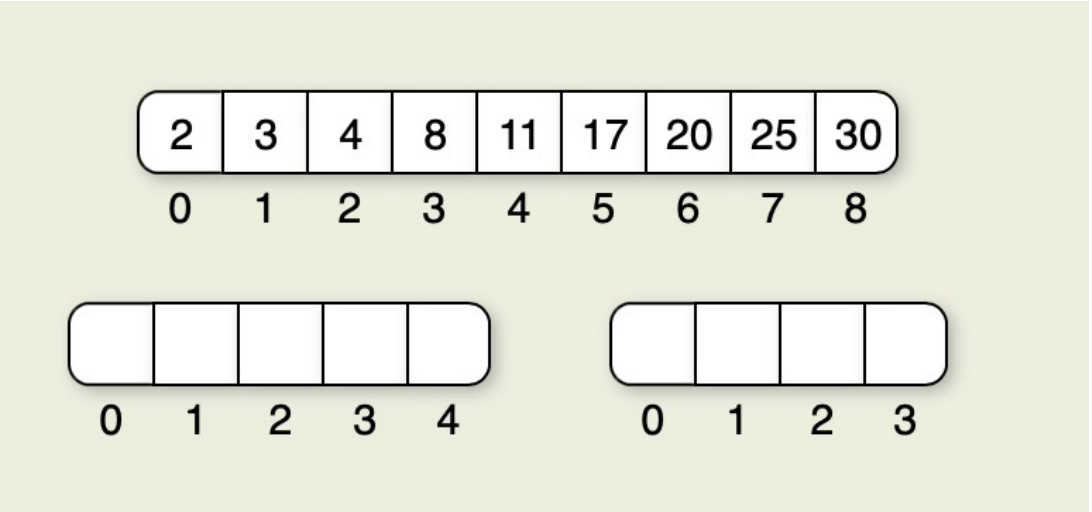


https://opendsa-server.cs.vt.edu/ODSA/Books/Everything/html/Mergesort.html

Ejemplo

Merge Sort: b) Fusión (*merge*)

Tras la fusión todos los elementos están ordenados



Como los subarrays están ordenados, para ordenar N elementos sólo hemos recorrido los N elementos una vez

https://opendsa-server.cs.vt.edu/ODSA/Books/Everything/html/Mergesort.html

Pseudo-código

```
MergeSort (data, left, right) {
    if (left < right) {
       mid = divide (left, right)
       MergeSort (data, left, mid)
       MergeSort (data, mid+1, right)
       Merge(data, left, mid+1, right)
    }
}
```

Pseudocode for Merge:

```
C = output [length = n]
A = 1st sorted array [n/2]
B = 2nd sorted array [n/2]
i = 1
j = 1
```

```
for k = 1 to n
        if A(i) < B(j)
                C(k) = A(i)
                i++
        else [B(j) < A(i)]
                C(k) = B(j)
                j++
end
```

(ignores end cases)

https://opendsa-server.cs.vt.edu/ODSA/Books/Everything/html/Mergesort.html

Resultados Empíricos (MergeSort)

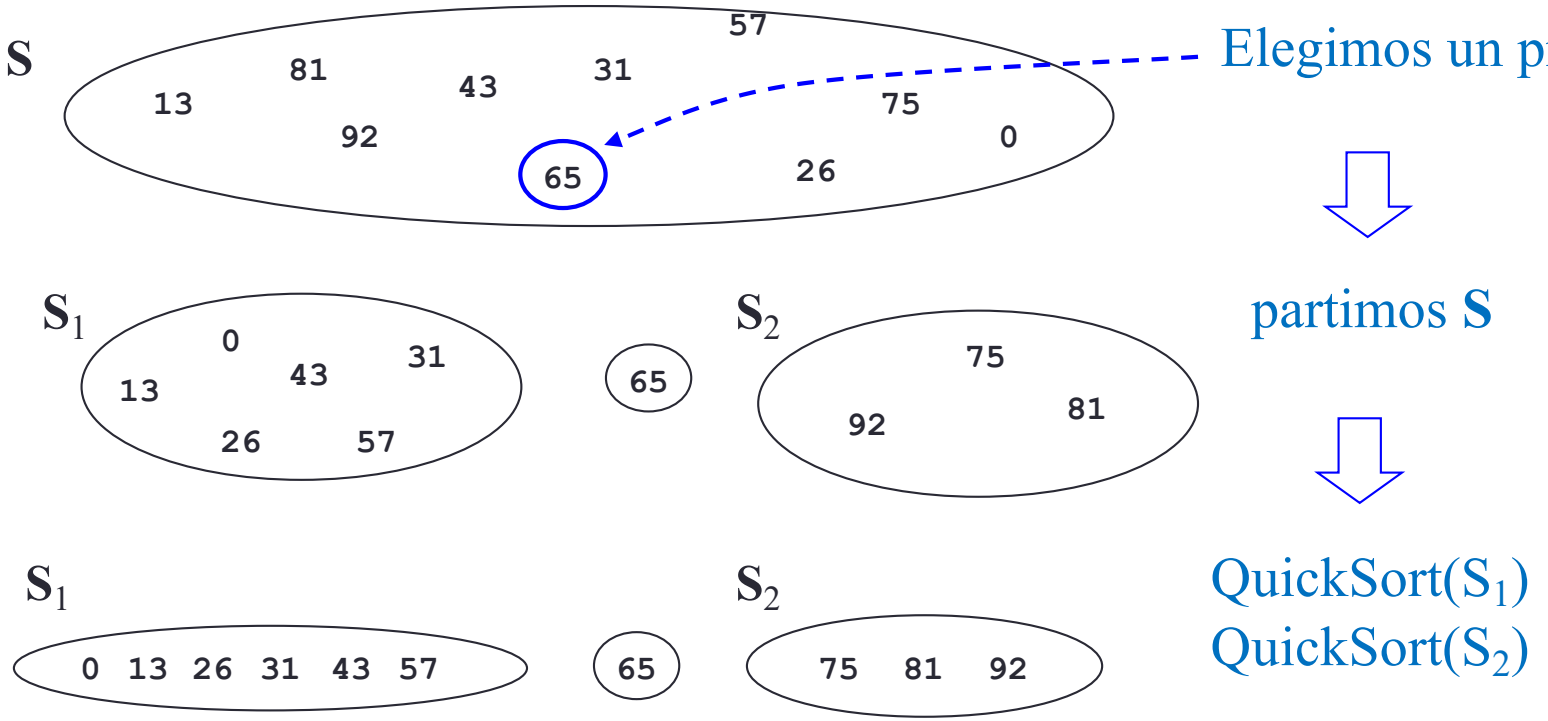
1.000 elementos	→	0.003 seconds
10.000	→	0.035
100.000	→	0.407
1.000.000	→	4.897
10.000.000	→	54.133

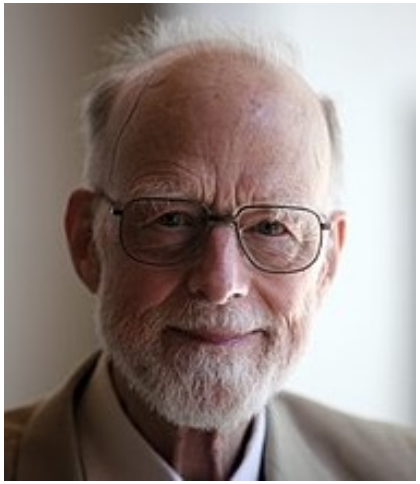
Ordenando listas que contienen números enteros elegidos aleatoriamente

1959

Ejemplo (3/3): Quicksort

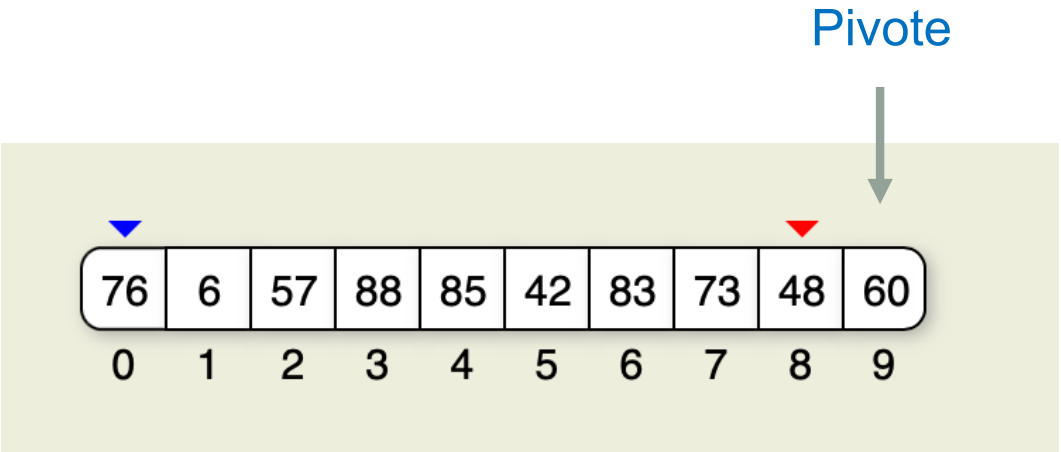
- Elegimos un elemento como ***pivote***, situando a su izquierda los elementos que sean más pequeños que él y los mayores a su derecha.
- Al final de este proceso el elemento que se ha usado como *pivote* queda en su posición definitiva, y ordenamos recursivamente las dos mitades.





Tony Hoare

Partition



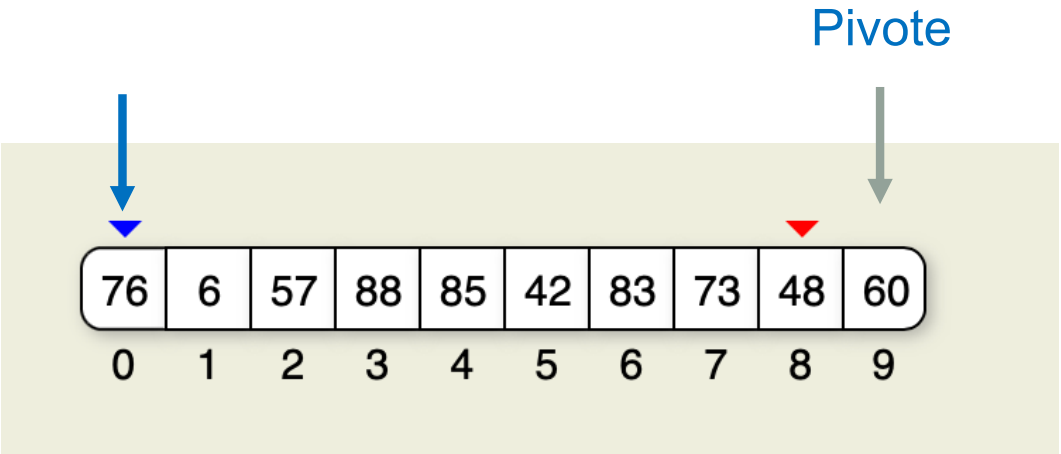
Elegimos el último elemento como nuestro pivote

Colocamos un indice al principio (izquierda) y al final (derecha) del resto de los elementos

https://opendsa-server.cs.vt.edu/ODSA/Books/Everything/html/Quicksort.html#partition

Paso 1

Partition



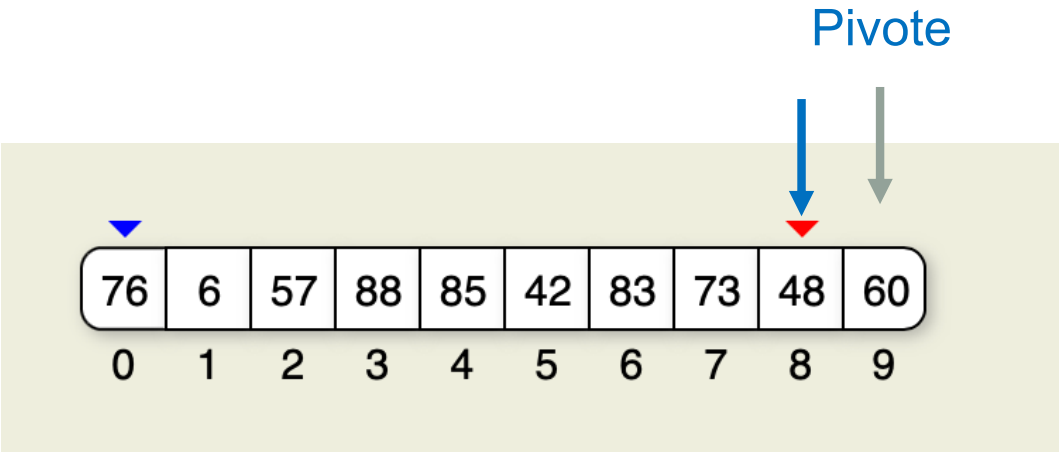
Paso 1: Avanzamos el puntero de la izquierda mientras apunte a elementos que sean menores que el pivote

En este ejemplo: No puedo avanzarlo

https://opendsa-server.cs.vt.edu/ODSA/Books/Everything/html/Quicksort.html#partition

Paso 2

Partition



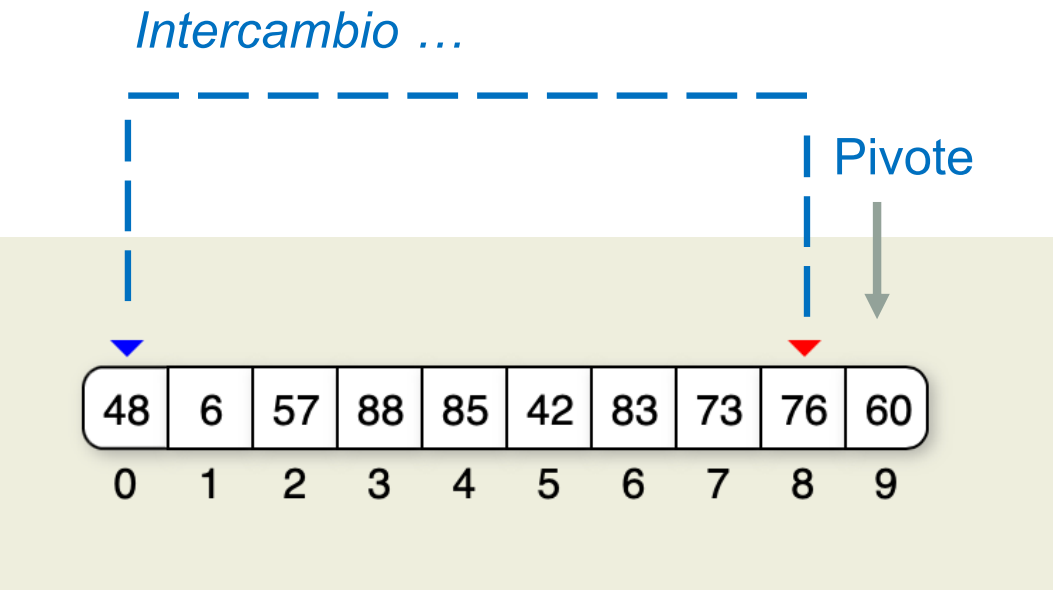
Paso 2: Retrocedemos el puntero de la derecha mientras apunte a elementos que sean mayores que el pivote

En este ejemplo: No puedo retrocederlo

https://opendsa-server.cs.vt.edu/ODSA/Books/Everything/html/Quicksort.html#partition

Paso 3

Partition

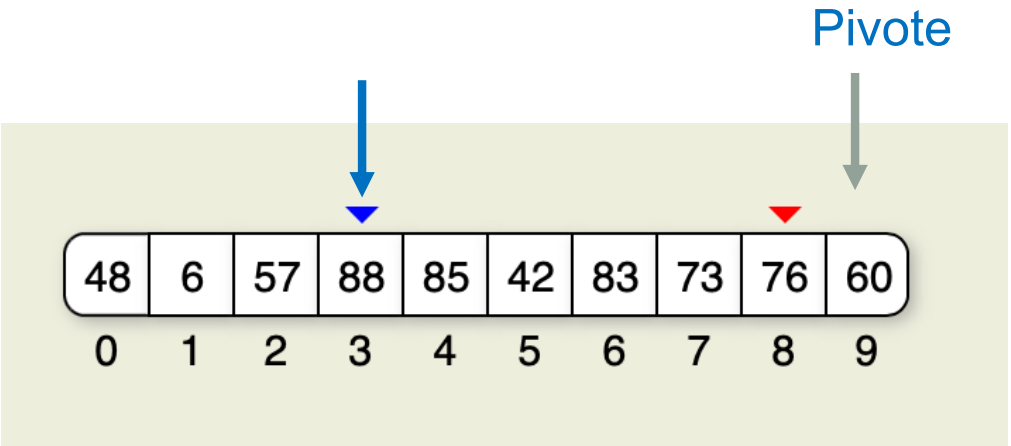


Paso 3: Intercambiamos porque, con respecto al pivote, estos elementos no estaban en la partición que les corresponde.

https://opendsa-server.cs.vt.edu/ODSA/Books/Everything/html/Quicksort.html#partition

Paso 1

Partition

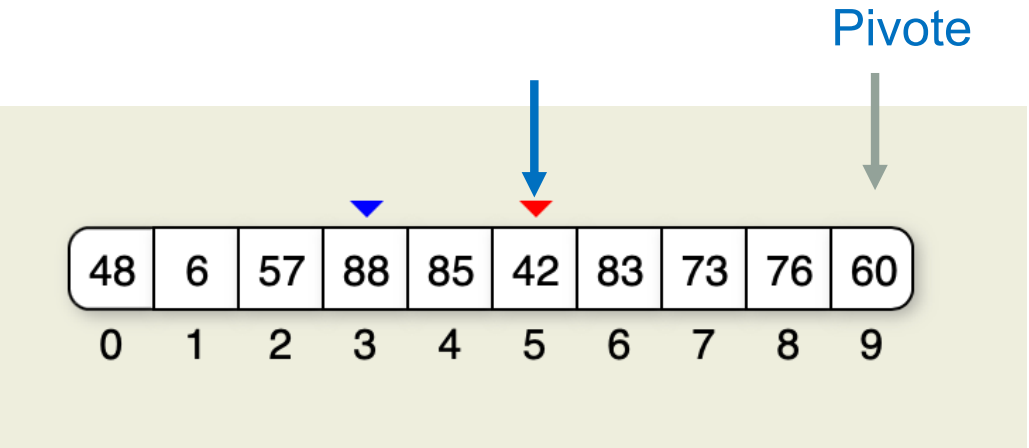


Paso 1: Avanzamos el puntero de la izquierda mientras apunte a elementos que sean menores que el pivote

https://opendsa-server.cs.vt.edu/ODSA/Books/Everything/html/Quicksort.html#partition

Paso 2

Partition

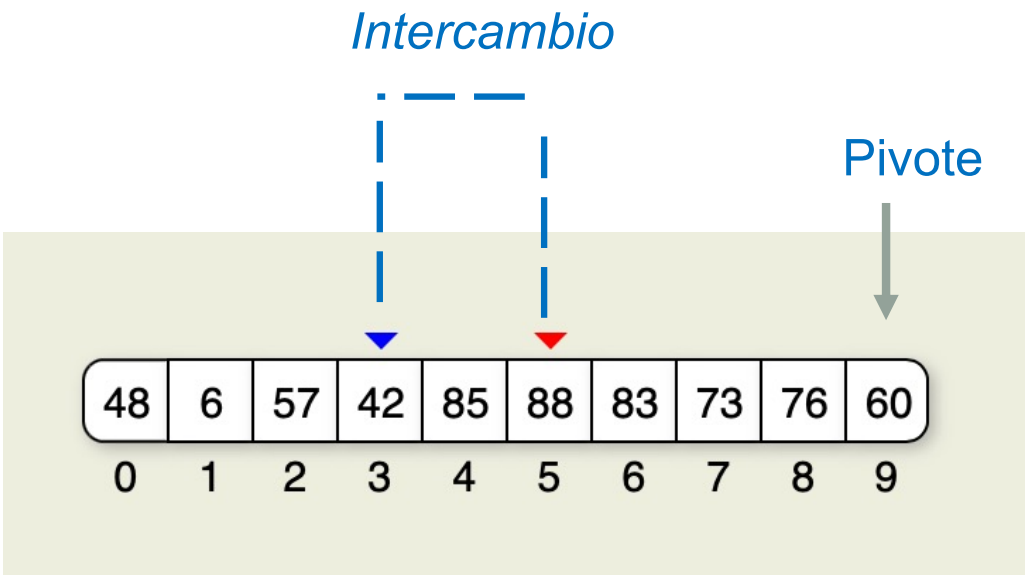


Paso 2: Retrocedemos el puntero de la derecha mientras apunte a elementos que sean mayores que el pivote

https://opendsa-server.cs.vt.edu/ODSA/Books/Everything/html/Quicksort.html#partition

Paso 3

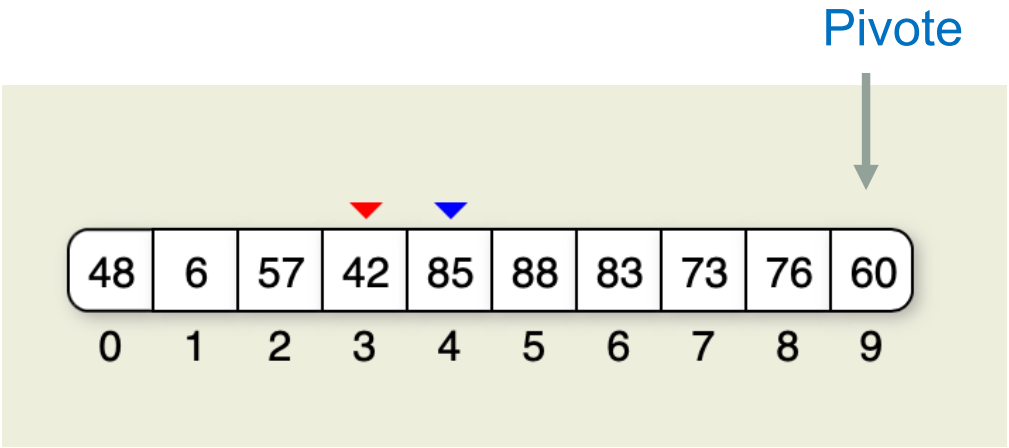
Partition



¿Cuando paramos ?

Parada

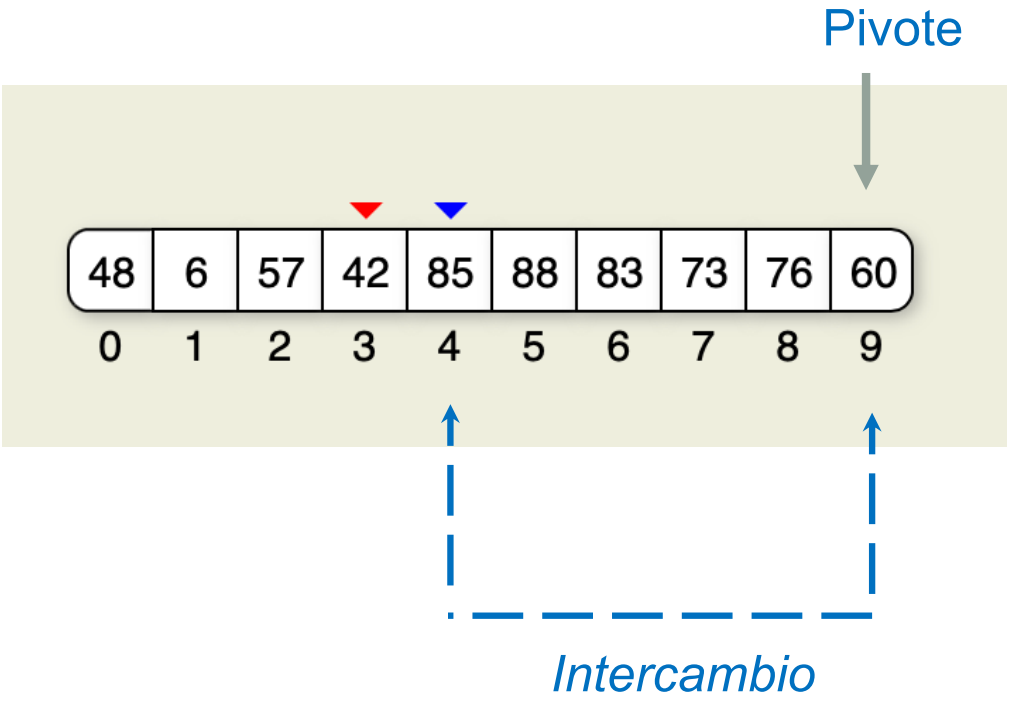
Partition



Paramos cuando se cruzan los índices

Paso 3

Partition



Paramos cuando se cruzan los índices
… y ahora ya podemos colocar el pivote en su posición definitiva

Resultado = 48 6 57 42 60 88 83 73 76 85
Posición del pivote = 4

Pseudo-código

```
QuickSort (data, left, right)
   if (left < right)
      pivot_index = Partition (data, left, right)
      Quicksort (data, left, pivot_index-1)
      Quicksort (data, pivot_index+1, right)
```

Pseudo-código

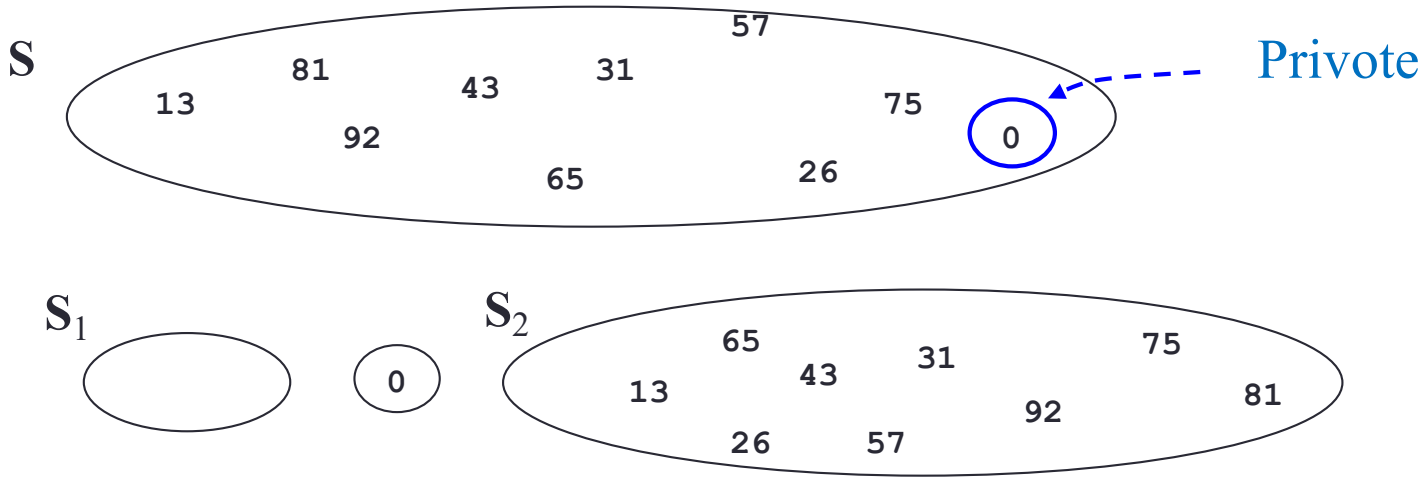
```
QuickSort (data, left, right)
   if (left < right)
      pivot_index = Partition (data, left, right)
      Quicksort (data, left, pivot_index-1)
      Quicksort (data, pivot_index+1, right)
```

Compáralo con MergeSort ….

```
MergeSort (data, left, right)
    if (left < right)
       mid = divide (left, right)
       MergeSort (data, left, mid-1)
       MergeSort (data, mid, right)
       Merge(data, left, mid, right)
```

Estrategias de Elección del Pivote en QuickSort

1. Utilizar siempre el primer (o el último) elemento
   - Funciona bien cuando los números están desordenados
   - Rendimiento malo (y potencial desbordamiento de pila) si los números están ordenados (todos los elementos van a S1 o a S2)

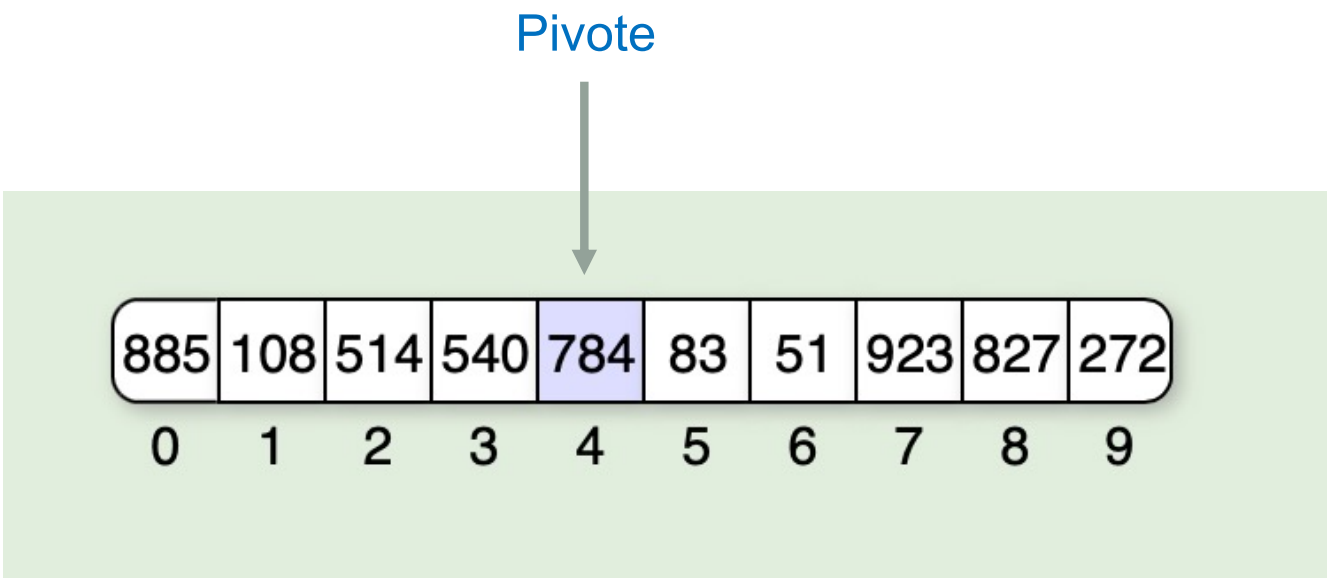


Estrategias de Elección del Pivote en QuickSort

1. Utilizar siempre el primer (o el último) elemento
   - Funciona bien cuando los números están desordenados
   - Rendimiento malo (y potencial desbordamiento de pila) si los números están ordenados (todos los elementos van a S1 o a S2)

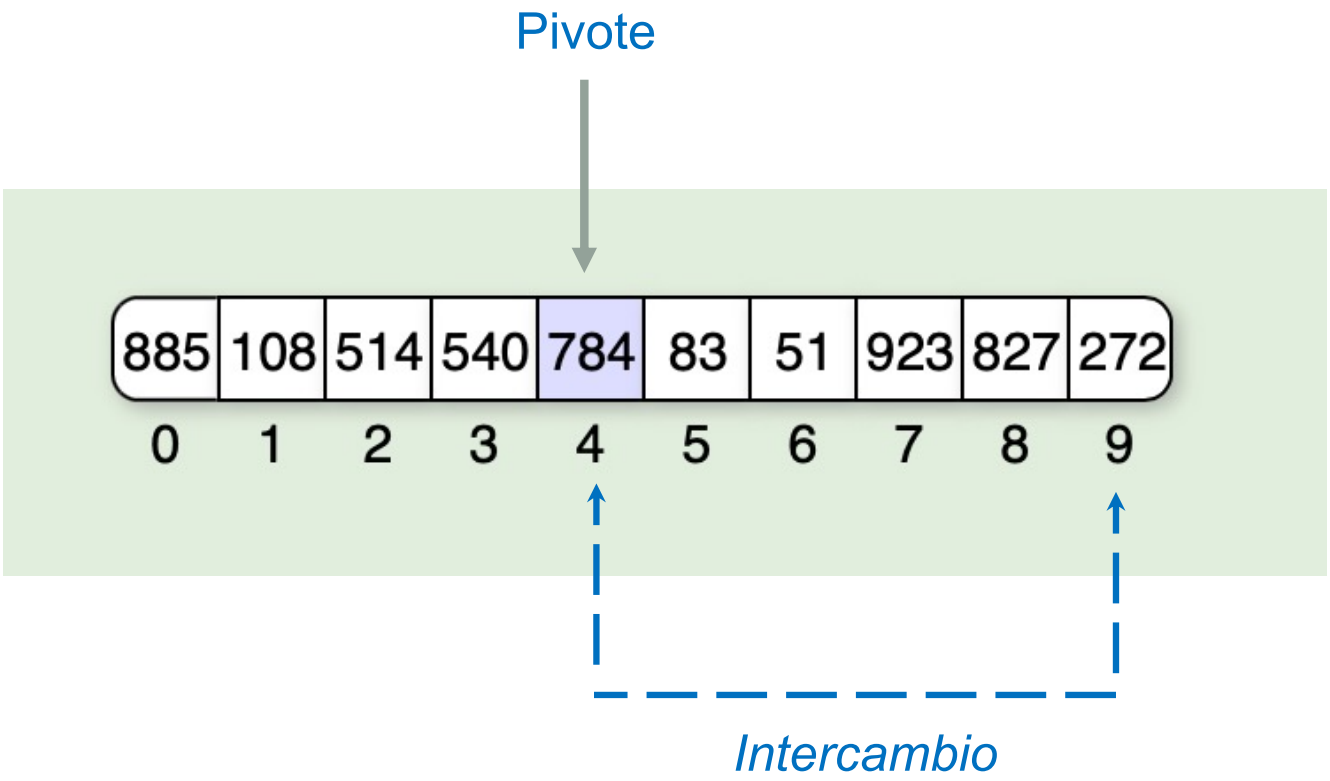
2. Elegir el pivote aleatoriamente (*Randomized Algorithm*)
   - Evita el problema de desbordamiento (es generalmente más seguro)
   - Tiene un coste adicional: elegir números aleatorios

Partition (detalle adicional cuando se elige el pivote aleatoriamente)



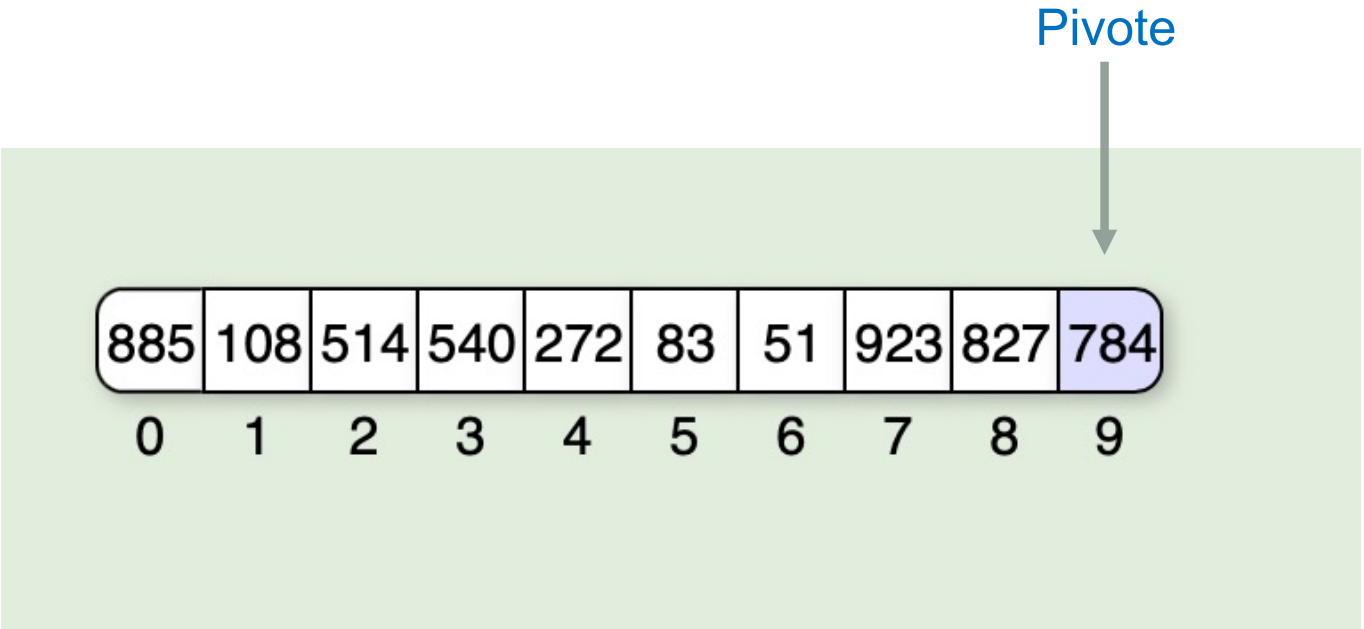
1) Elegimos un elemento cualquiera como pivote

Partition (detalle adicional cuando se elige el pivote aleatoriamente)



2) Movemos el pivote al final de la lista

Partition (detalle adicional cuando se elige el pivote aleatoriamente)



3) Continuamos como si hubésemos elegido como pivote el último elemento de la lista

Estrategias de Elección del Pivote en QuickSort

1. Utilizar siempre el primer (o el último) elemento
   - Funciona bien cuando los números están desordenados
   - Rendimiento malo (y potencial desbordamiento de pila) si los números están ordenados (todos los elementos van a S1 o a S2)

2. Elegir el pivote aleatoriamente (*Randomized Algorithm*)
   - Es generalmente más seguro
   - Tiene el coste adicional de elegir números aleatorios

3. Desordenar el array y aplicar la primera estrategia
   - Permite acotar el coste extra de desordenar el array, eliminar el coste de elección de números aleatorios, y reducir el potencial problema de desbordamiento de pila.

Resultados Empíricos (pivote fijo; datos aleatorios)

		MergeSort	QuickSort
1.000 elementos	→	0.003 seconds	0.003 seconds
10.000	→	0.036	0.030
100.000	→	0.401	0.356
1.000.000	→	4.897	4.380
10.000.000	→	57.342	56.847

Ordenando listas idénticas que contienen números enteros elegidos aleatoriamente.

MergeSort hace siempre el mismo trabajo independientemente de los datos de entrada; QuickSort depende del pivote elegido (con resultados que pueden ser un poco mejor o mucho peor que MergeSort)

Resultados Empíricos (pivote fijo; datos ordenados)

		MergeSort	QuickSort
1.000 elementos	→	0.003 seconds	Stack Overflow
10.000	→	0.036	
100.000	→	0.401	
1.000.000	→	4.897	
10.000.000	→	57.342	

Ordenando listas idénticas que contienen números enteros ordenados.

Resultados Empíricos QuickSort
(Pivote fijo, pivote aleatorio; datos aleatorios)

		Pivote Fijo	Pivote Aleatorio
1.000 elementos	→	0.003 seconds	0.003 seconds
10.000	→	0.030	0.036
100.000	→	0.356	0.431
1.000.000	→	4.380	5.435
10.000.000	→	57.342	65.848

Ordenando listas idénticas que contienen números enteros elegidos aleatoriamente; QuickSort eligiendo pivote fijo y eligiendo pivote aleatorio.

Estrategia Divide y Vencerás

A divide-and-conquer algorithm recursively breaks down a problem into two or more sub-problems of the same or related type, until these become simple enough to be solved directly.

The solutions to the sub-problems are then combined to give a solution to the original problem.

Hasta donde debemos subdividir (en MergeSort o en QuickSort): casos base

1. Hasta que tengamos 0 o 1 elemento
   - Realiza muchas llamadas recursivas con pocos elementos

2. Hasta tamaños que se puedan ordenar rápidamente con algorimos iterativos (*solución híbrida*)
   - Reduce el coste de estas llamadas recursivas
   - Evita problemas de desbordamiento de pila

QuickSort suele combinarse con InsertionSort (que es iterativo) para evitar problemas de desbordamiento de pila

Resultados Empíricos

		QuickSort	InsertionSort
10 elementos	→	0.000 seconds	0.000 seconds
50	→	0.000	0.000
U → 100	→	0.000	0.000
150	→	0.000	0.001
200	→	0.000	0.002
300	→	0.001	0.004

Ordenando listas idénticas que contienen números enteros elegidos aleatoriamente.

Dependiendo del algoritmo iterativo que utilicemos para nuestra versión hibrida obtendremos nuestro valor umbral (U)

Pseudo-Código (soluciones híbridas)

```
MergeSort (data, left, right) {
    if abs(left - right) <= U:
       InsertionSort (data, left, right)
    else
       mid = divide (left, right)
       MergeSort (data, left, mid)
       MergeSort (data, mid+1, right)
       Merge(data, left, mid+1, right)
    }
}
```

Valor umbral

```
QuickSort (data, left, right)
   if abs(left - right) <= U:
        InsertionSort (data, left, right)
   else
        pivot_index = Partition (data, left, right)
        Quicksort (data, left, pivot_index-1)
        Quicksort (data, pivot_index+1, right)
```

Algoritmo Iterativo; podemos elegir otro

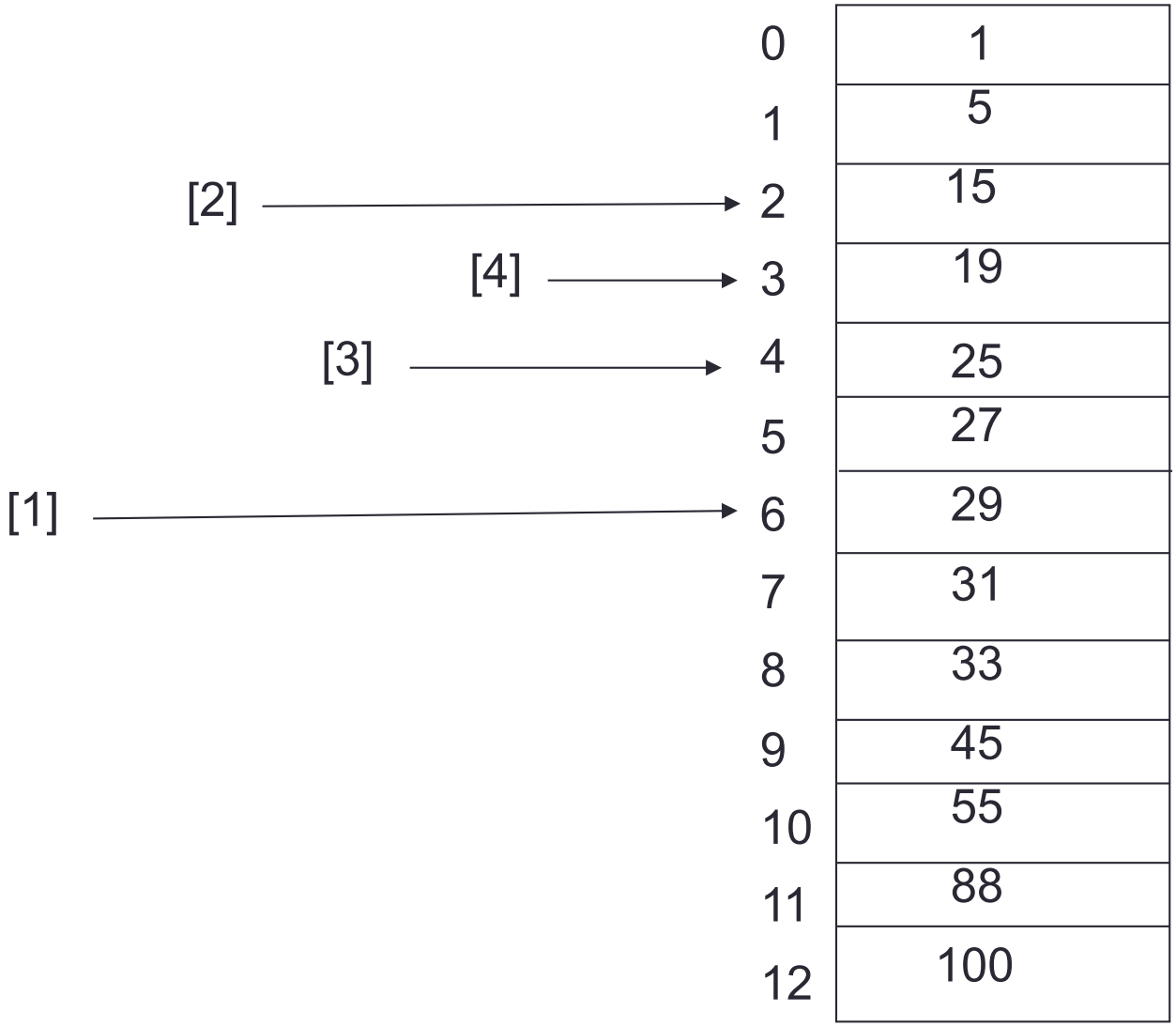
Decrease and Conquer

Se utiliza este nombre en algoritmos que dividen el problema en varios subproblemas pero **sólamente necesitan resolver uno** de los subproblemas

- Ejemplos
  - Búsqueda binaria en un array ordenado
  - Búsqueda en un árbol
  - Busqueda de la mediana

Ejemplo (1/2): Búsqueda binaria

search key = 19



[1]: (0 + 12) / 2 = 6
[2]: (0 + 5) / 2 = 2
[3]: (3 + 5) / 2 = 4
[4]: (3 + 3) / 2 = 3

Ejemplo (1/2): Búsqueda binaria

- Es realmente muy rápida porque, el número máximo de comparaciones es $\log_2(N)$

Tamaño de tabla	Número máximo de comparaciones
64	6
256	8
1.024	10
4.096	12
16.384	14
65.536	16
262.144	18
1.048.576	20
4.194.304	22
16.777.216	24
67.108.864	26
268.435.456	28

Ejemplo (2/2): Búsqueda de la Mediana

- Dado un vector no ordenado con elementos distintos, sin ordenar los elementos, calcular el valor del elemento que va en la mitad del vector (posición $\lceil n/2 \rceil$) si el vector estuviese ordenado.

- Vamos a ver una solución más general que nos permite calcular el valor del elemento que va en cualquier posición k que necesitemos!
  - Se resuelve fácilmente reutilizando el método de partición que vimos en QuickSort

https://en.wikipedia.org/wiki/Quickselect

Ejemplo: Búsqueda de la Mediana

```
function QuickSelect(data, left, right, k)
    if left = right
       return data[left]

    pivot_index = partition(data, left, right)
    // El pivote está en su posición definitiva

    if k = pivot_index
        return data[k]
    else if k < pivot_index
        return QuickSelect(data, left, pivot_index - 1, k)
    else
        return QuickSelect(data, pivot_index + 1, right, k)
```

Solución Recursiva

https://en.wikipedia.org/wiki/Quickselect

Ejemplo: Búsqueda de la Mediana

```
function QuickSelect(data, left, right, k)
    loop
        if left = right
            return data[left]

        pivot_index = partition(data, left, right)
        // El pivote está en su posición definitiva

        if k = pivot_index
            return data[k]
        else if k < pivot_index
            right = pivot_index - 1
        else
            left = pivot_index + 1
```

Solución Iterativa

Evita la recursividad de cola!