UNIVERSIDAD NACIONAL DE INGENIERÍA

Facultad de Ciencias

Escuela Profesional de Ciencia de la Computación

- **Curso: Fundamentos de Programación**
- **Docente: Américo Chulluncuy Reynoso**

2025-II

Sesión 3:

Algoritmos de Ordenamiento y Búsqueda II

Contenido

1. Eficiencia de Algoritmos: tiempo - espacio
   - Notación O grande.
2. Algoritmo de Mezcla (Merge Sort)
3. Algoritmo Rápido (Quick Sort)
4. Algoritmo de Búsqueda Binaria (Binary Search)

1. Eficiencia de algoritmos

- Una forma fundamental de comparar algoritmos es analizar **cuánto esfuerzo computacional requieren** para resolver una tarea (**eficiencia**).

- Para describir este esfuerzo utilizamos la notación Big O (O grande) que nos permite expresar como crece el **tiempo de ejecución** (o el **uso de memoria**) en función del tamaño de la entrada, considerando los diferentes casos de desempeño:

  ✓ **Peor caso**: El rendimiento del algoritmo bajo las condiciones más desfavorables. Big O describe, en general, el peor caso posible del comportamiento de un algoritmo.

  ✓ **Caso promedio**: El rendimiento esperado en condiciones típicas o promedio de entrada.

  ✓ **Mejor caso**: El rendimiento del algoritmo en las condiciones más favorables

Ejemplos comunes de la notación O grande

- **O(1) orden constante:** indica que tendremos el mismo rendimiento sin importar el tamaño de entrada.
- **O(log(n)) orden logarítmico:** indica que el tiempo de ejecución aumenta linealmente mientras que el tamaño n crece de forma exponencial.
- **O(n) orden lineal:** indica que la complejidad del algoritmo aumenta de manera proporcional al tamaño del arreglo.
- **O(n log(n)) orden lineal-logarítmico:** indica que el tiempo de ejecución crece proporcional a n veces el logaritmo de n.
- **O(n^2) orden cuadrático:** el tiempo de ejecución crece proporcional al cuadrado del tamaño de entrada.
- **O(2^n) orden exponencial:** el tiempo se duplica con cada incremento en el tamaño n.

Ejemplos:

Dado un arreglo de tamaño n, estimar la eficiencia de un algoritmo para:

- **Determinar si el primer elemento de un arreglo es igual al segundo**
  - ✓ Requiere solo una comparación. Es independiente de n. $O(1)$
- **Determinar si el primer elemento de un arreglo es igual a cualquiera de los otros elementos.** $O(n)$
  - ✓ Requiere n-1 comparaciones (n domina)

    término de mayor crecimiento
- **Determinar si algún elemento está duplicado en el arreglo** $O(n^2)$
  - ✓ Requiere n(n-1)/2 comparaciones (n^2 domina)

    $\frac{n^2-n}{2}$

    término de mayor crecimiento

¿Qué mide O grande?

El **crecimiento del tiempo de ejecución** de un algoritmo **en relación con la cantidad de elementos procesados**.

Un algoritmo que requiere n^2 comparaciones:

- Si n = 4, el algoritmo requerirá 16 comparaciones
- Si n = 8, 64 comparaciones.

x4

Un algoritmo que requiere $\frac{n^2}{2}$ comparaciones

- Si n = 4, el algoritmo requerirá 8 comparaciones
- Si n = 8, 32 comparaciones.

x4

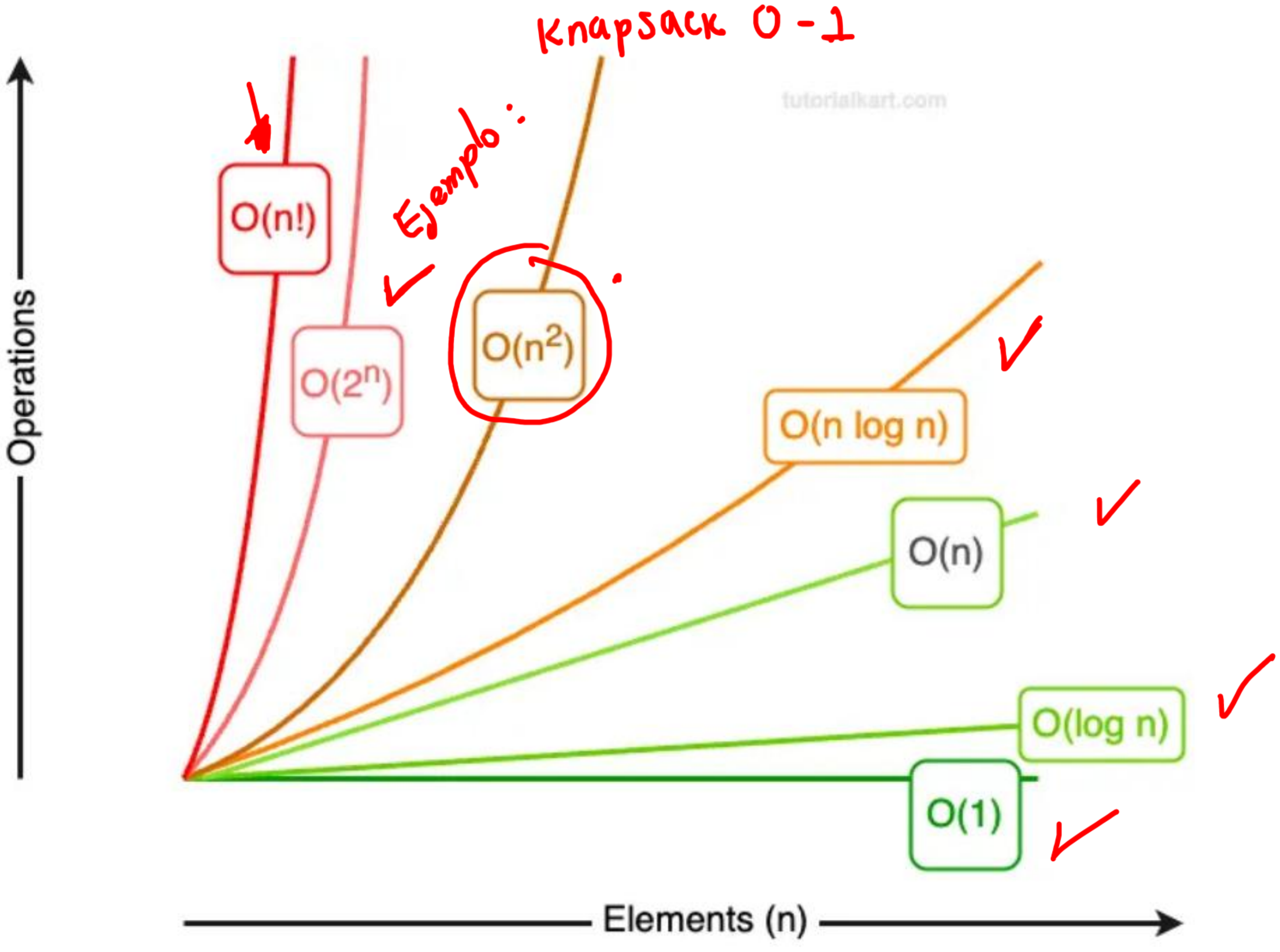
En ambos algoritmos al **duplicar el número de elementos** se **cuadriplica el número de comparaciones**.

Ambos algoritmos crecen como el cuadrado de n, por lo que **O grande ignora la constante** y **ambos algoritmos se consideran $O(n^2)$**.

Ejercicios

- Eficiencia del algoritmo de Búsqueda Lineal. $O(n)$
- Eficiencia del algoritmo de Burbuja $O(n^2)$
- Eficiencia del algoritmo de Ordenamiento por selección $O(n^2)$
- Eficiencia del algoritmo de Ordenamiento por Inserción $O(n^2)$

Comparativa de los órdenes de complejidad



Método Divide y Vencerás

- Este enfoque consiste en descomponer un problema complejo en varios subproblemas más pequeños y manejables, que son versiones simplificadas del problema original.

- Estos subproblemas se resuelven recursivamente y, una vez resueltos, se combinan sus soluciones para obtener la solución completa del problema original.
  - ✓ Caso base: Se resuelve directamente sin necesidad de recurrencia.
  - ✓ Caso recursivo: Se lleva a cabo en tres etapas clave:
    - 1 División: El problema se descompone en uno o más subproblemas de tamaño reducido.
    - 2 Conquista: Se resuelven los subproblemas de forma recursiva.
    - 3 Combinación: Se combinan las soluciones de los subproblemas para construir la solución al problema original.

Ejercicio: 3

Implementar una función que reciba dos arreglos ordenados y los combine en un solo arreglo ordenado. Ejemplo:

Input:

```
arr1 = {1, 3, 5, 7};
arr2 = {2, 4, 6, 8};
```

Output:

```
arr = {1, 2, 3, 4, 5, 6, 7, 8}
```

2. Ordenamiento por mezcla (Merge sort)

- Es un algoritmo basado en el enfoque de Divide y Vencerás.

- Es un algoritmo estable (mantiene el orden relativo de los elementos con valores iguales).

- Tiene una complejidad temporal de O(n log n), lo que lo convierte en uno de los algoritmos más eficientes para ordenar grandes cantidades de datos.

¿Cómo Funciona?

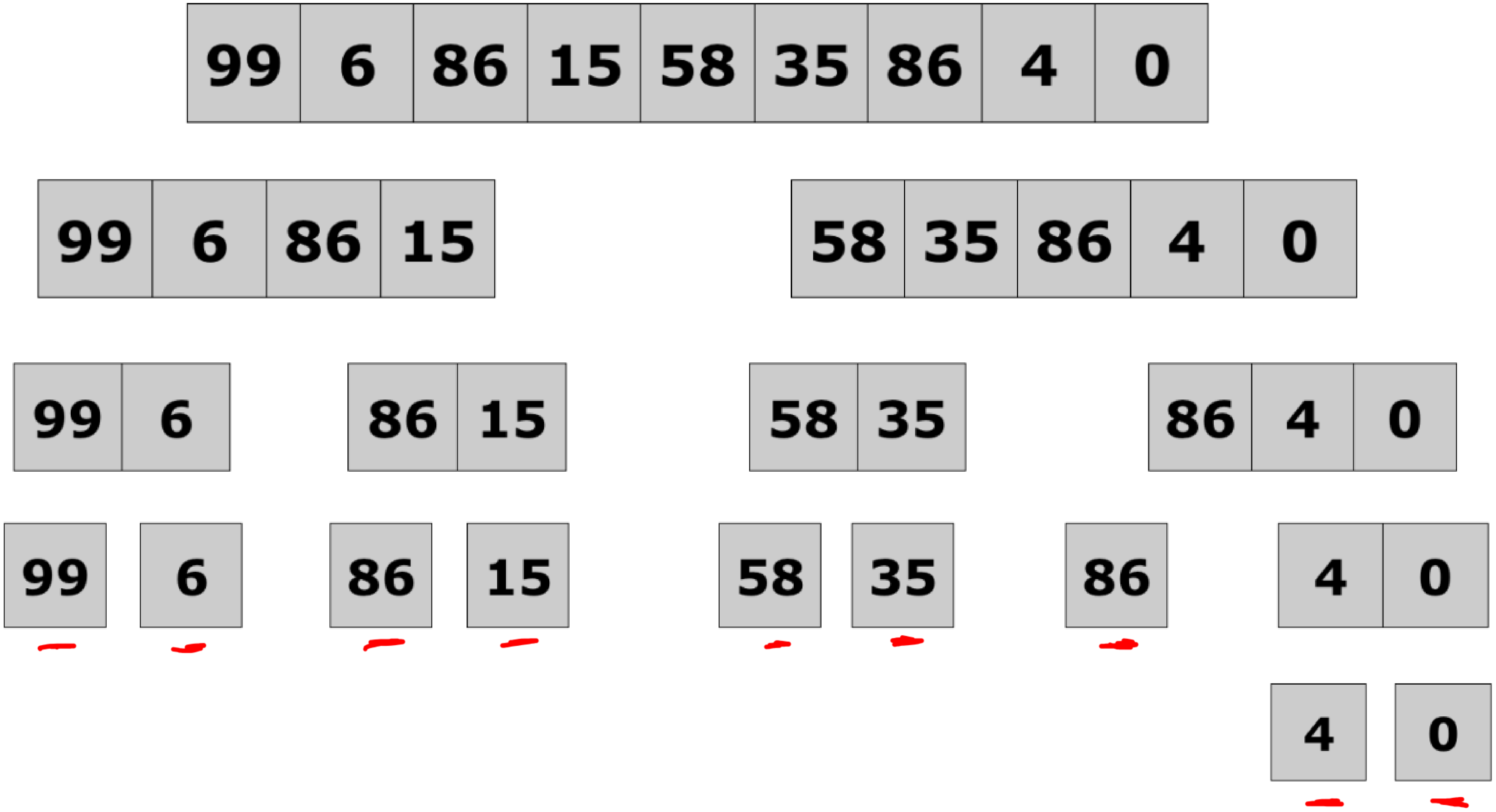
- Dividir: El arreglo se divide recursivamente en dos mitades.
- Ordenar: Cada mitad se ordena de manera recursiva.
- Mezclar: Se fusionan las dos mitades ordenadas para obtener el arreglo completo ordenado.

Ejemplo: Merge Sort

Arreglo a ordenar (en forma creciente):

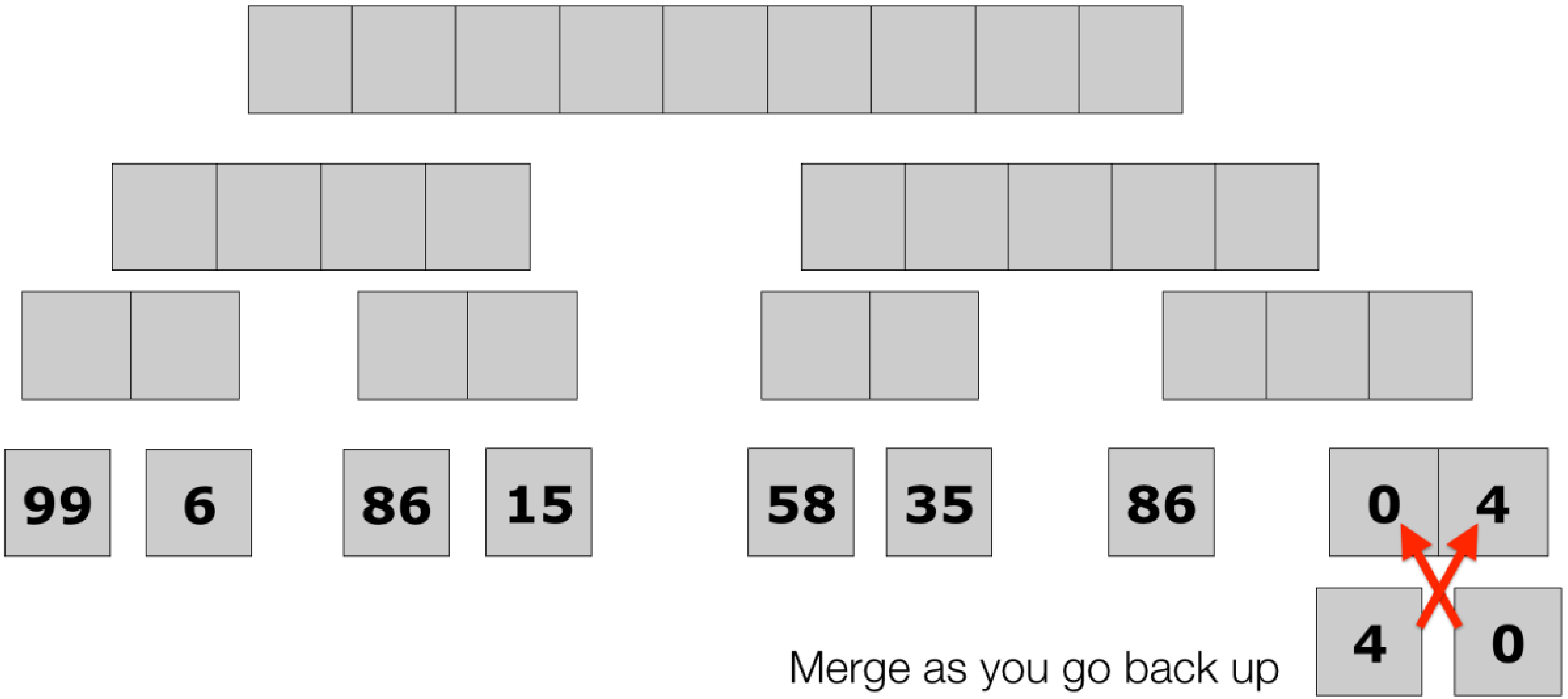
99	6	86	15	58	35	86	4	0

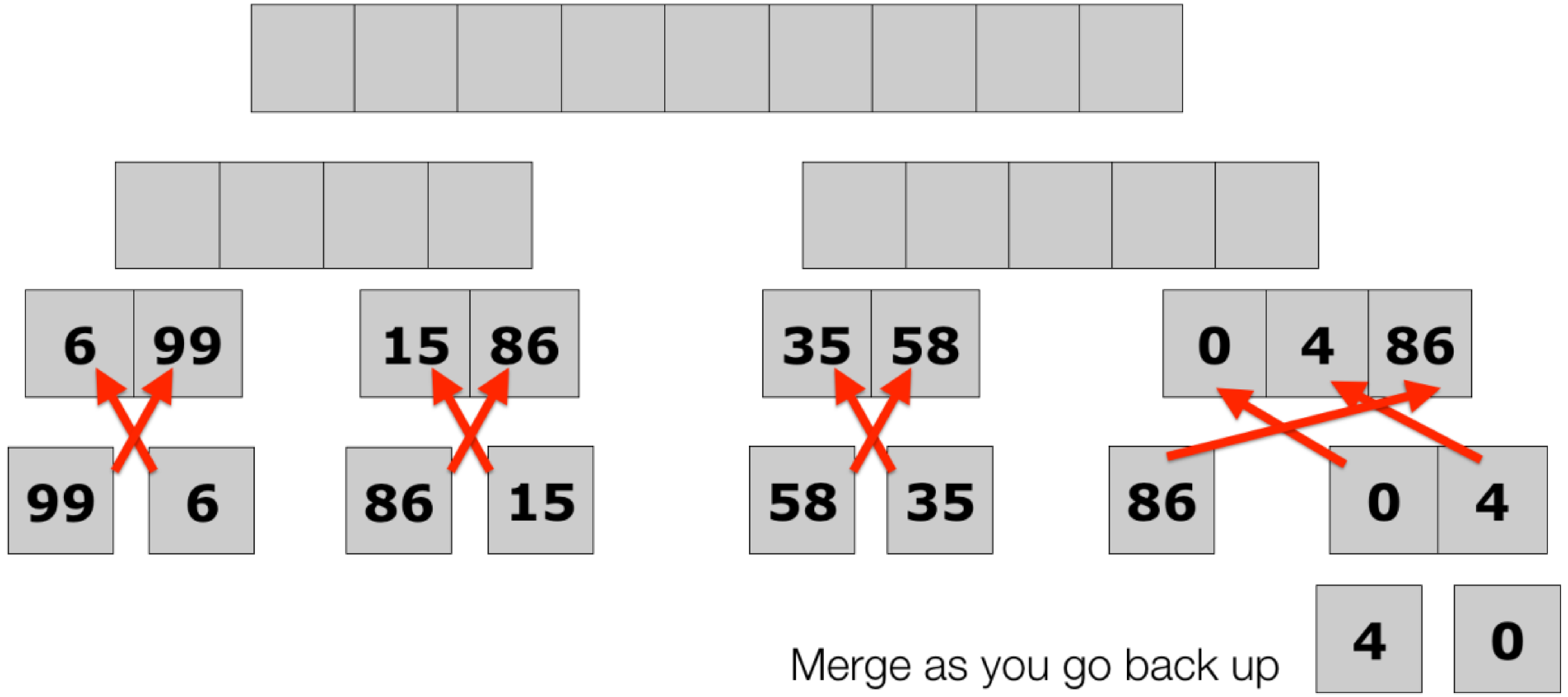
Dividir



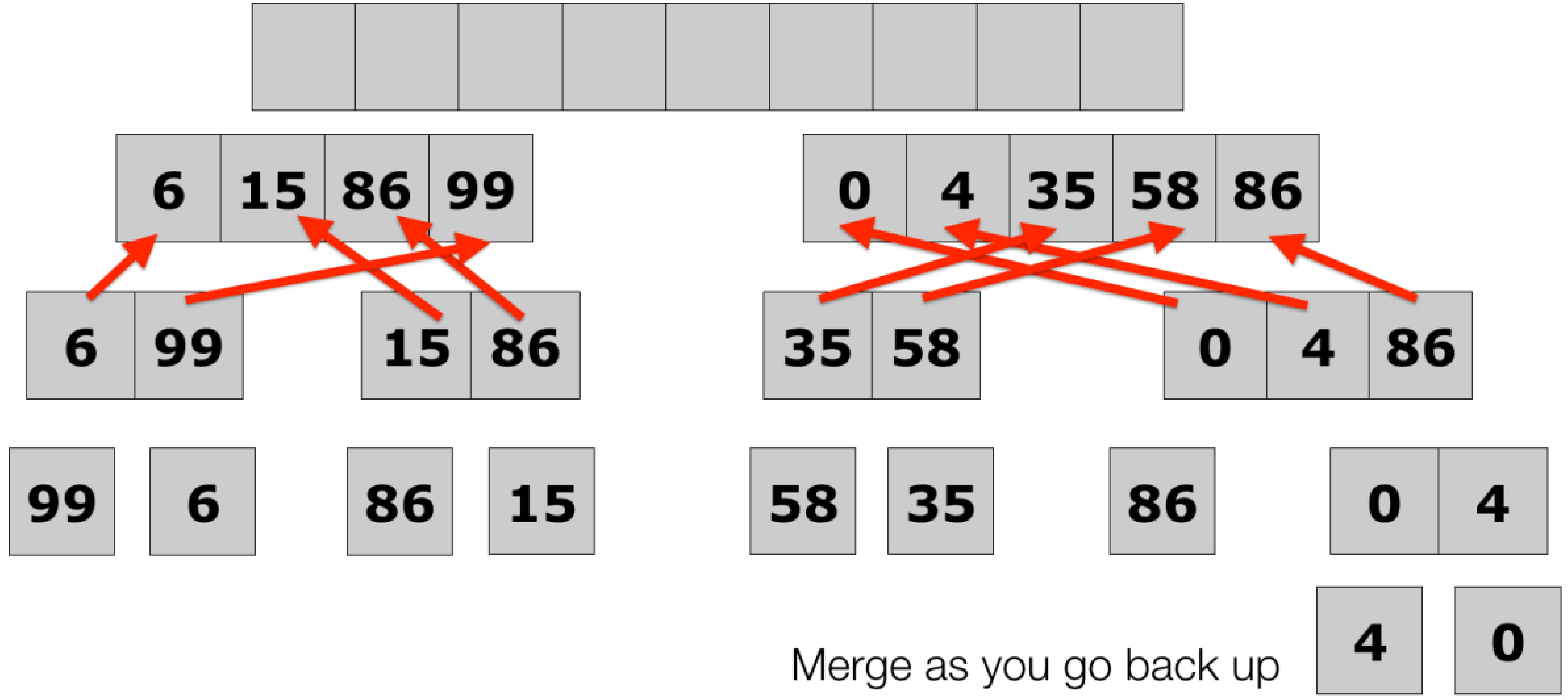
99 6 86 15 58 35 86 4 0
99 6 86 15
58 35 86 4 0
99 6
86 15
58 35
86 4 0
99
6
86
15
58
35
86
4 0
4
0

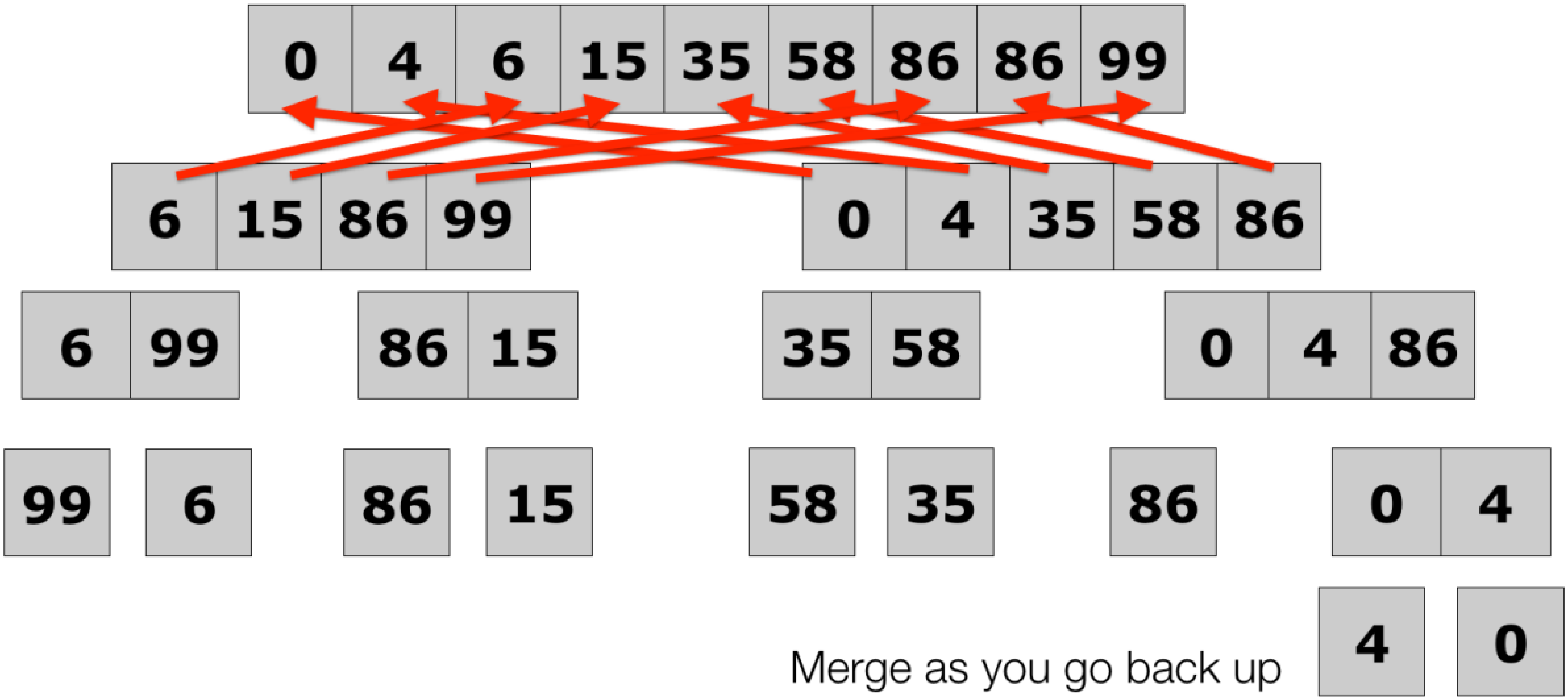
Ordenar y fusionar:





6
99
15
86
35
58
0
4
86
99
6
86
15
58
35
86
0
4
Merge as you go back up
4
0


6 15 86 99
0 4 35 58 86
6 99
15 86
35 58
0 4 86
99
6
86
15
58
35
86
0 4
4
0
Merge as you go back up



Merge as you go back up

Merge Sort recursivo en C++

Ver repositorio de la semana 3

3. Ordenamiento rápido (Quick Sort)

- Es otro algoritmo de ordenamiento eficiente que utiliza la estrategia Divide y Vencerás. Creado por Tony Hoare en 1960.

- Es uno de los algoritmos más populares debido a su eficiencia en la práctica. Tiene una complejidad promedio de O(n log n), pero su rendimiento depende de la elección del pivote.

¿Cómo Funciona?

- Elegir un pivote: el elemento pivote puede ser el primer, último, medio, o un pivote aleatorio.
- Particionar: Se reorganiza el arreglo colocando los elementos menores que el pivote a su izquierda y los mayores a su derecha.
- Recursión: El algoritmo se aplica recursivamente a las dos mitades del arreglo (a la izquierda y derecha del pivote).

Observaciones

- La elección del pivote tiene un gran impacto en la eficiencia del algoritmo. Elegir un pivote de manera aleatoria ayuda a evitar peores casos ($O(n^2)$).

- Es un algoritmo in-place (no necesita memoria adicional significativa como Merge Sort)

- En sistemas con poca memoria la recursión profunda puede causar un desbordamiento de pila.

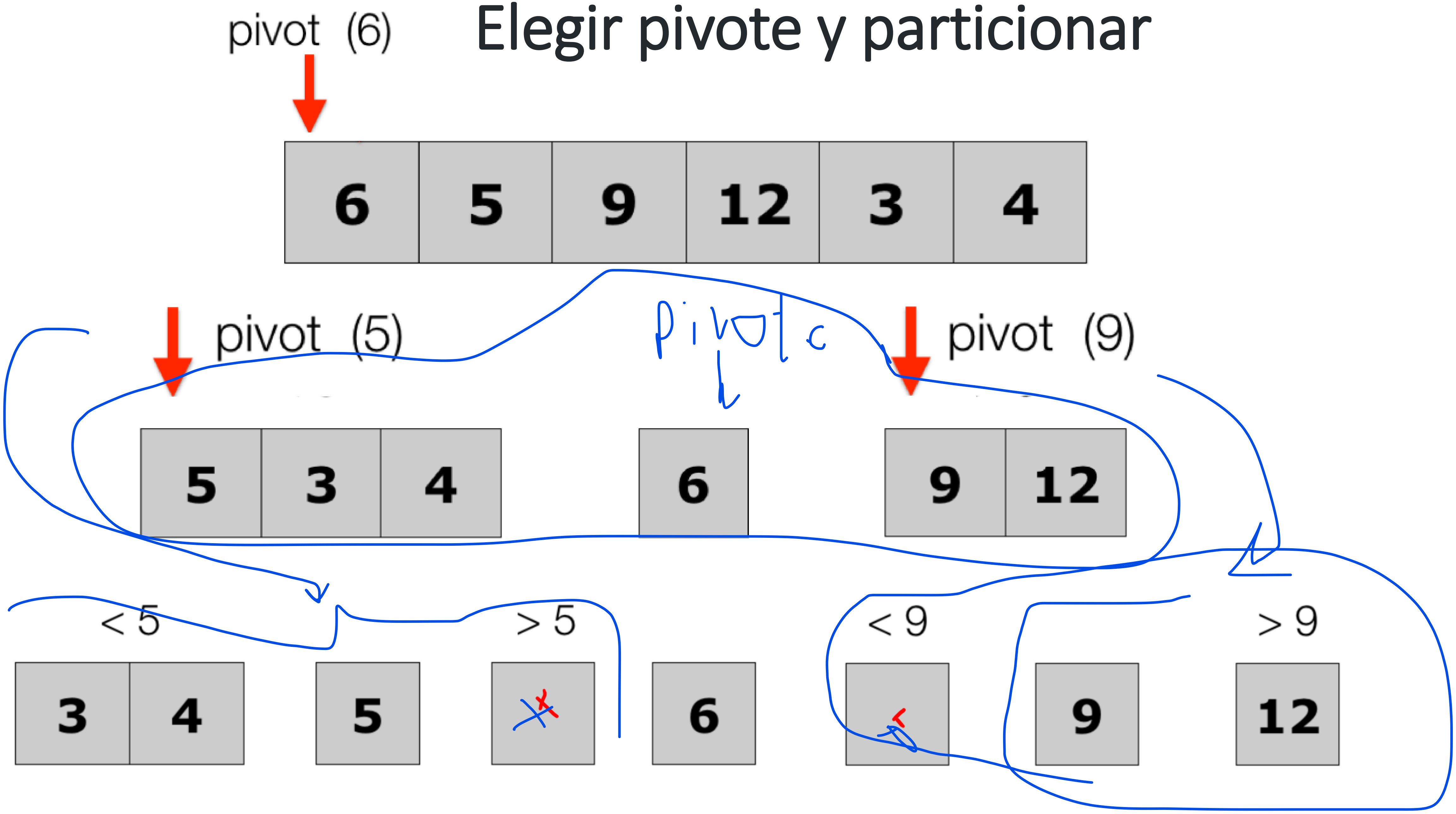
- A diferencia de Merge Sort, Quick Sort no es un algoritmo estable (elementos con valores iguales pueden cambiar su orden relativo),

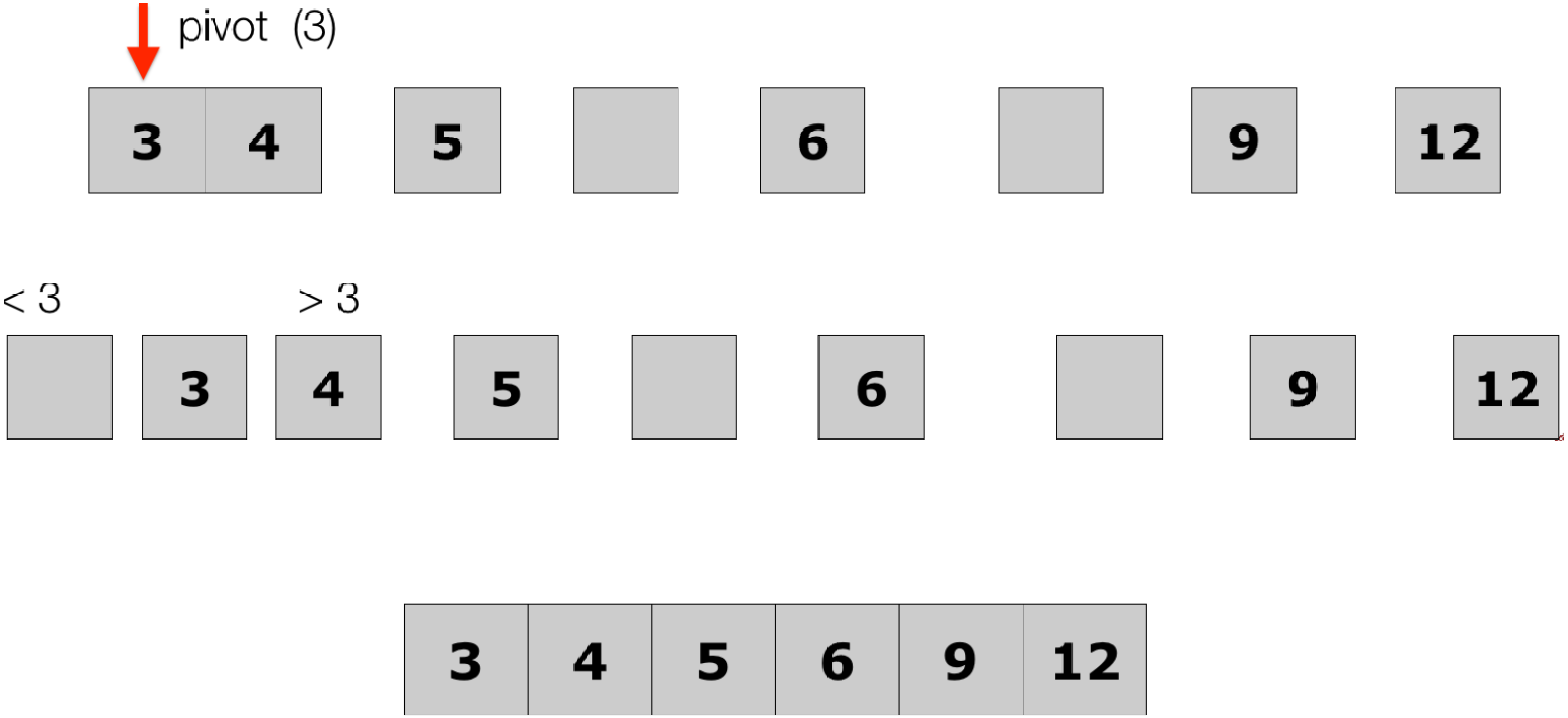
Ejemplo: Quick Sort

- Arreglo a ser ordenado:

6	5	9	12	3	4

Elegir pivote y particionar




pivot (3)
3
4
5
6
9
12
< 3
> 3
3
4
5
6
9
12
3
4
5
6
9
12

Quick Sort recursivo en C++

Ver repositorio de la semana 3

Ejercicio: adivinar un número

- Implemente un programa en C++ que permita al usuario adivinar un número aleatorio generado por la computadora en un rango de 1 a 100.
- Durante la ejecución, el programa brinda retroalimentación al usuario, indicando si el número ingresado es **demasiado alto**, **demasiado bajo** o **correcto**. El ciclo de adivinanzas continúa hasta que el usuario acierta el número.

¿Cuántos intentos son necesarios para adivinar el número? Responda la misma pregunta cuando el rango es entre 1 y 1000

Para optimizar la cantidad de intentos, es fundamental comprender el funcionamiento del algoritmo de **Búsqueda Binaria**, ya que este permite reducir significativamente el número de pruebas necesarias.

5. Búsqueda Binaria

- Es un algoritmo eficiente **para encontrar un elemento dentro de un arreglo ordenado.**

3	4	5	1	2

aplicar busqueda binaria a un arreglo circular.

- El **principio básico** es dividir el arreglo en mitades de forma iterativa y reducir el espacio de búsqueda a la mitad en cada paso.
- La búsqueda binaria tiene una complejidad temporal O(log n), lo que la hace mucho más eficiente que la búsqueda lineal (O(n)).

¿Cómo funciona? Calcular el índice medio = (inicio + fin)/2;

- Comparar el valor buscado con el valor medio. Si coinciden, el algoritmo finaliza.
- Si el valor medio es mayor que el valor buscado, el nuevo rango de búsqueda será la mitad izquierda del arreglo.
- Si el valor medio es menor que el valor buscado, el nuevo rango de búsqueda será la mitad derecha del arreglo
- Repetir el proceso hasta que el valor sea encontrado o el rango de búsqueda se reduzca a cero (inicio > fin). <> (inicio <= fin)

¿Por qué es más eficiente?

- Cada vez, divide por 2 la cantidad de datos que quedan por buscar.

- **Recordemos el ejercicio "adivinar un número":** Si empezamos con 100 números, tendremos que adivinar como máximo 7 veces !.
  Ejemplo:
  Número a adivinar 1 : 50 -> 25 -> 12 -> 6 -> 3 ->1
  Número a adivinar 100: 50 -> 75 -> 88 -> 93 -> 96 ->98 -> 100

- Podemos calcular el número de intentos para adivinar el número?
  Ejemplo con 16 elementos, si adivinamos incorrectamente hasta llegar a un elemento. 16 -> 8 -> 4 -> 2 ->1 $16 * \left(\frac{1}{2}\right)^{4} = 1$

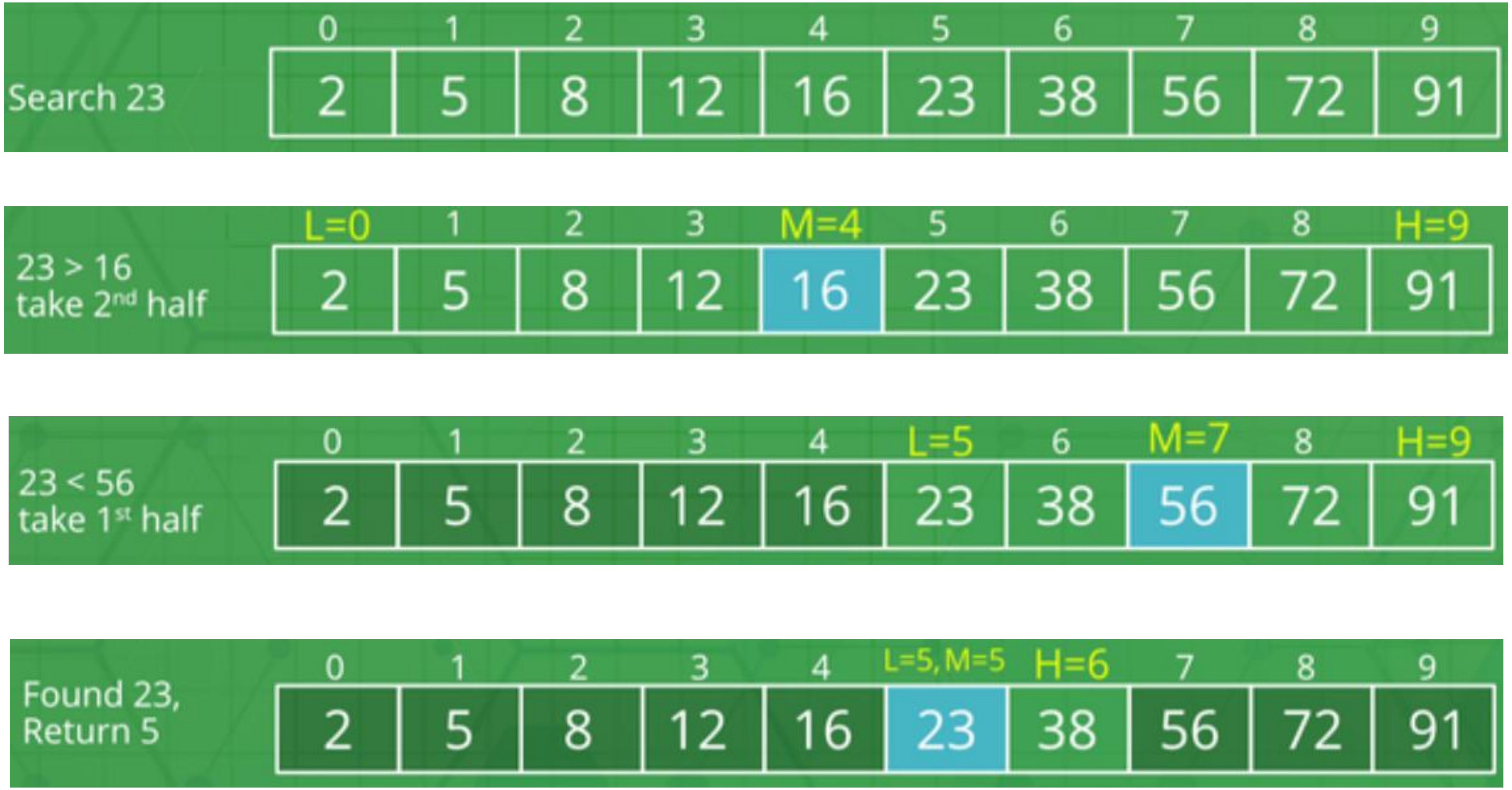
¿Por qué es más eficiente?

- En general: para n elementos $\mathrm{n} * \left(\frac{1}{2}\right)^{k} = 1$, donde k es el número de veces que debemos dividir.

  De donde: $k = \log_2(\mathrm{n})$

- Si n = 1000, $k = \log_2(1000) = 9.96 \approx 10$ (máximo número de pasos.)

Ejemplo: Búsqueda Binaria



Una comparación entre la búsqueda secuencial y binaria puede observarse aquí.

Búsqueda Binaria en C++

Ver repositorio de la semana 3

¿Versión recursiva?

Resumen

- **Complejidad temporal**: tiempo que tarda un algoritmo en función del tamaño de la entrada.

- **Complejidad espacial**: mide la cantidad de memoria usada por el algoritmo.

- **La notación Big-O** se utiliza para expresar la peor complejidad del algoritmo.

- **Merge Sort**: **Divide** el arreglo en dos mitades. **Ordena** ambas mitades de manera recursiva. **Fusiona** las mitades ordenadas.
  - ✓Complejidad: Tiempo: O(n log n) (siempre).
    Espacio: O(n) (espacio para la fusión).
  - ✓Estabilidad: Conserva el orden relativo de los elementos iguales.

- **Quick Sort**: **Selecciona un pivote**. **Reorganiza los elementos** para que los menores que el pivote queden a la izquierda y los mayores a la derecha. **Recursivamente ordena** los subarreglos.
  - ✓Complejidad: Temporal: $O(n \log n)$ (mejor caso, caso promedio)
    $O(n^2)$ (peor caso, pivote desequilibrado).
    Espacio: $O(\log n)$ para la recursión.
  - ✓Inestabilidad: Puede cambiar el orden relativo de elementos iguales.

- **Búsqueda Binaria**: Algoritmo eficiente para arreglos ordenados. Divide el arreglo en dos mitades y compara el elemento buscado con el valor en el punto medio. Repite el proceso con la mitad correspondiente.
  - ✓Complejidad: Tiempo: $O(\log n)$ (ideal para arreglos grandes).
  - ✓Espacio: $O(1)$ (en su versión iterativa).