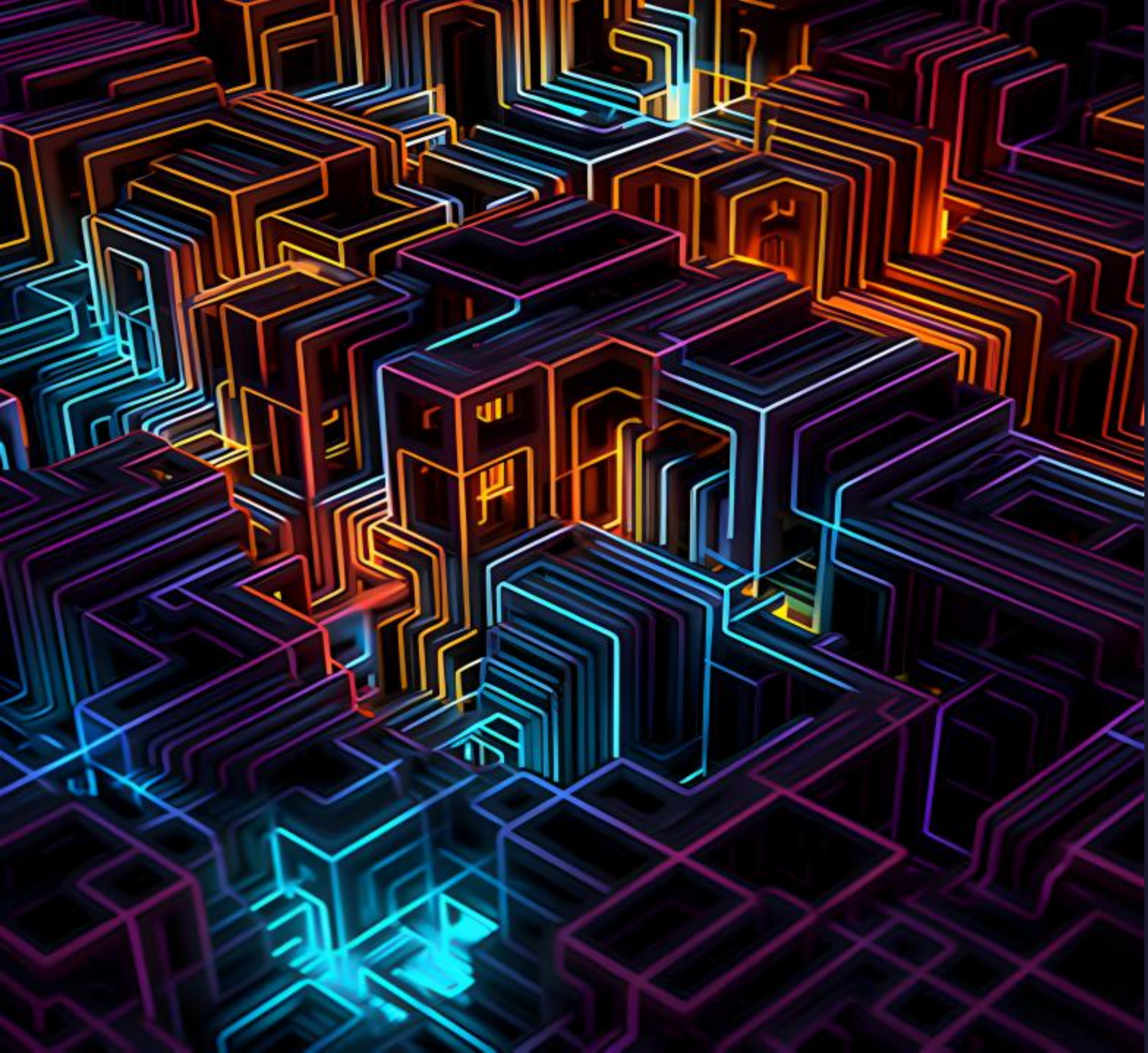


Técnicas de diseño de algoritmos: divide y vencerás

Análisis y diseño de algoritmos avanzados

Dra. Valentina Narváez Terán

Tecnológico de Monterrey

Actividad de sesión 04: decrementa-y-vencerás

- Diseña y codifica un algoritmo para cálculo de a^n bajo decrementa-y-vencerás

  Pista: $a^4 * a^4 = ??$

  Responde
  ¿Cuál es su complejidad?
  ¿Como sabes que tu implementación es en efecto de tipo decrementa y vencerás?

Técnicas de diseño: divide y vencerás

1. Un problema se divide en subproblemas de tamaño menor
2. Los subproblemas se resuelven de manera independiente (usualmente de manera recursiva)
3. Las soluciones a los subproblemas se combinan para formar la solución total

"Great Goddess of Algorithmics, grant that twice two be not four"

Técnicas de diseño: divide y vencerás

1. Un problema se divide en subproblemas de tamaño menor
2. Los subproblemas se resuelven de manera independiente (usualmente de manera recursiva)
3. Las soluciones a los subproblemas se combinan para formar la solución total

Algunos ejemplos:

Merge-sort
Quick-sort

Recorridos en arboles binarios

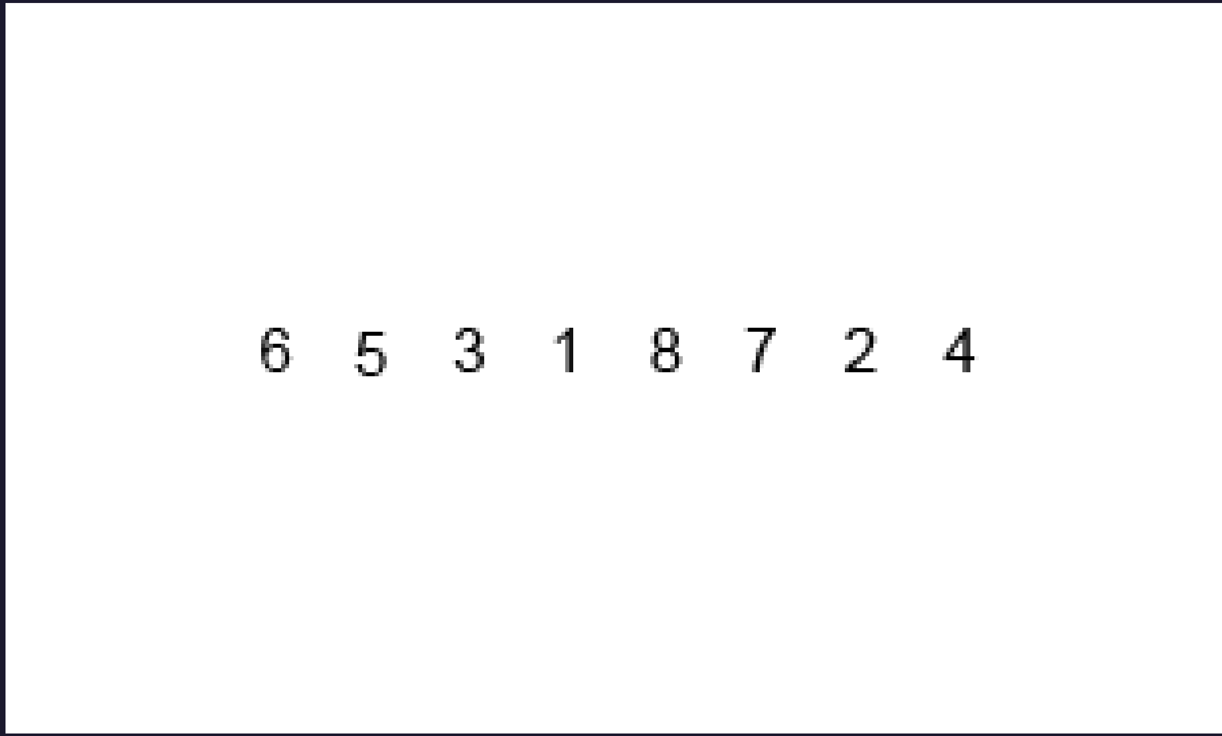
"Great Goddess of Algorithmics, grant that twice two be not four"

Algoritmos de ordenamiento: merge-sort

Fusión (Merge sort)

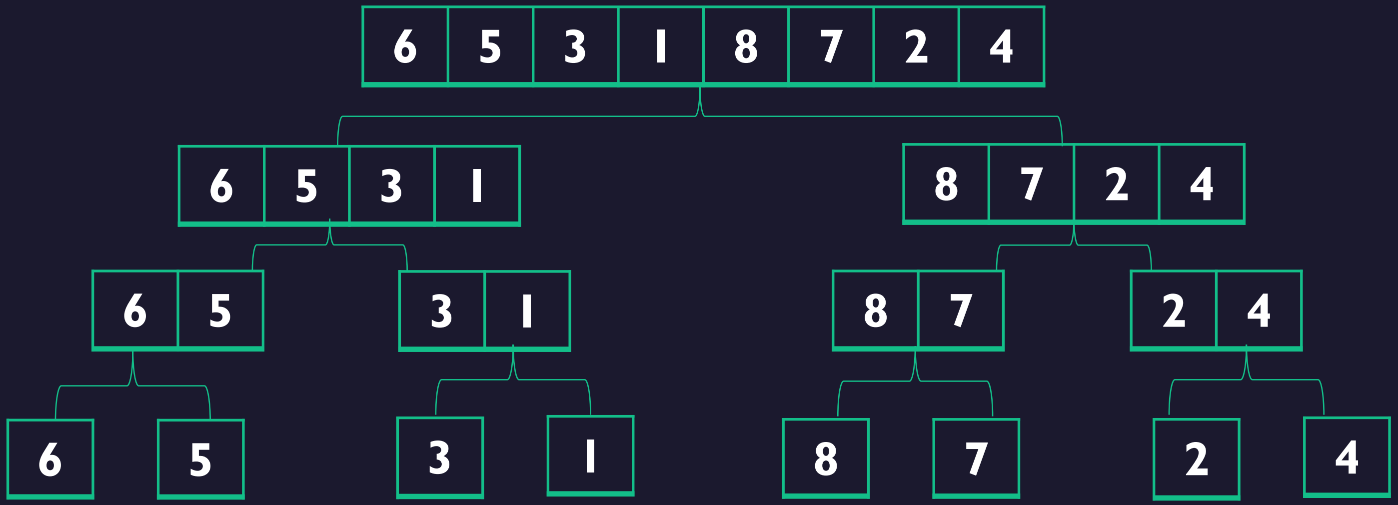
$O(n \log n)$

Crea particiones, dividiendo el arreglo a la mitad de forma recursiva, y fusionando las particiones



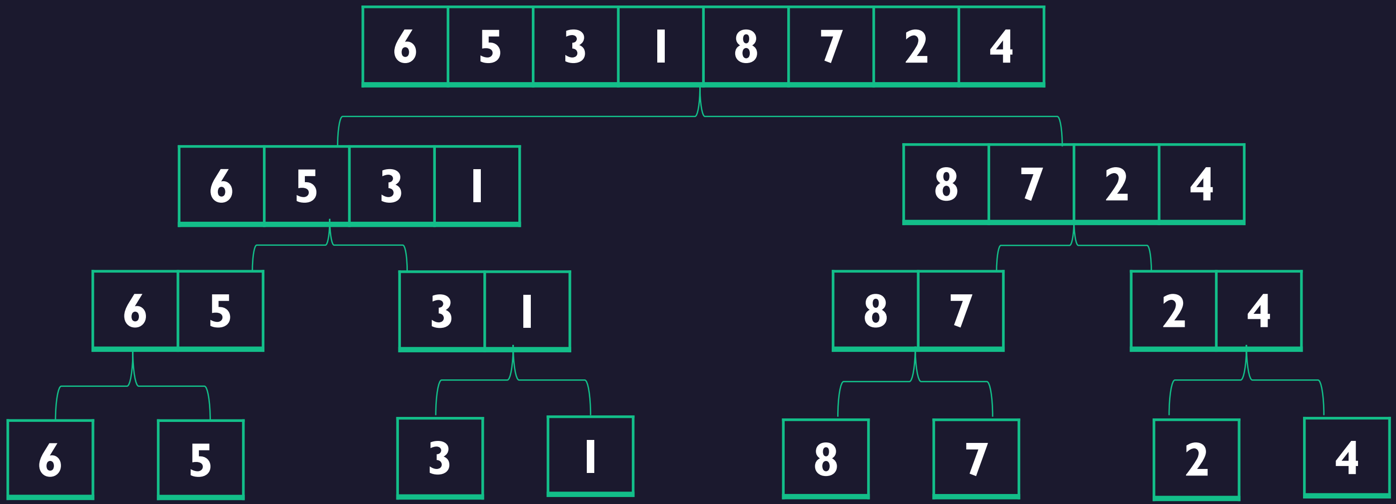
En lenguajes como C/C++ el parámetro A sería una referencia a memoria, y todas las operaciones ocurren in-situ

Los valores de A se sobrescriben dentro de la función merge usando los índices L, M y R




```
ALGORITHM mergeSort(A, L, R)
// Input: Arreglo A, índices de inicio L, y
final R de partición
// Output: A ordenado de L a R

if L < R then
    M = L + (R − L) / 2
    mergeSort(A, L, M)
    mergeSort(A, M + 1, R)
    merge(A, L, M, R)
```

Algoritmos de ordenamiento: merge-sort

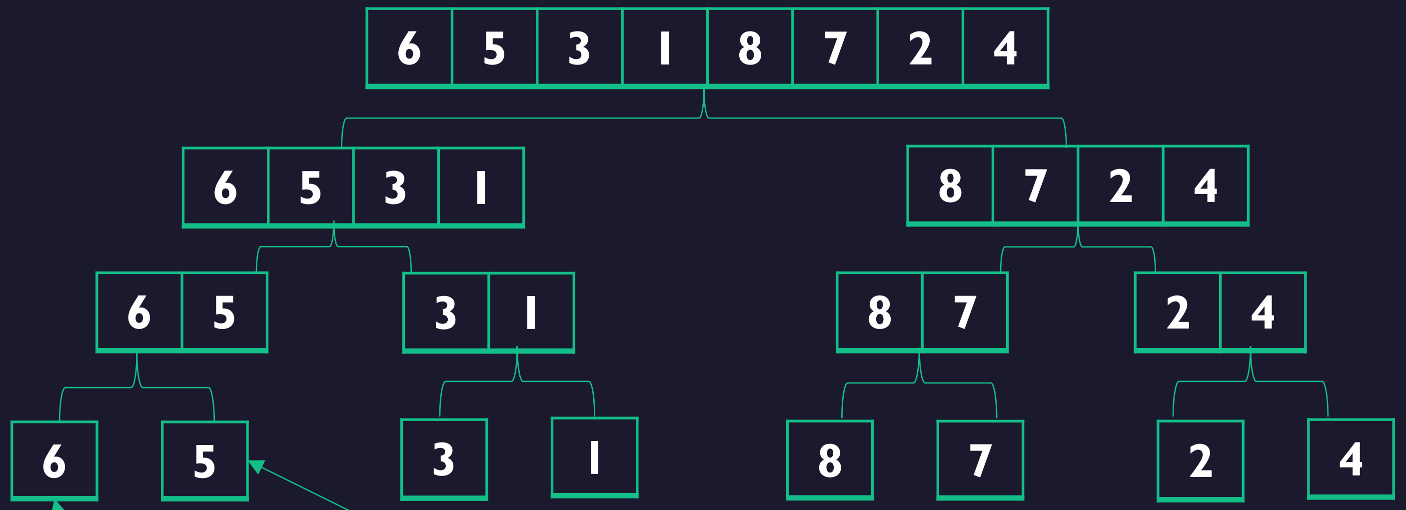


En lenguajes como Python, es posible pasar listas como parámetros usando slicing

```python
def mergeSort(A):
  n = len(A)
  if n > 1:
    B = mergeSort(A[:n//2] )
    C = mergeSort(A[n//2:] )
    return merge(B, C)
  else:
    return A
```

Algoritmos de ordenamiento: merge-sort

```
1:  ALGORITHM merge(A, l, m, r)
2:  // Input: Arreglo A índices de inicio l,
3:  punto medio m y final r de partición
4:  // Output: A ordenado
5:
6:  B = copia de A[L, ... , M]
7:  C = copia de A[M + 1, ..., R]
8:  s1, s2 = tamaños de B y C
9:  i, k, k = L
10: while i < s1 and j < s2
11:         if B[i] < C[j]
12:                 A[k] = B[i]
13:                 i++
14:         else
15:                 A[k] = C[j]
16:                 j++
17: while i < s1
18:         A[k] = B[i]
19:         i++ y k++
20: while j < s2
21:         A[k] = C[j]
22:         j++ y k++
```



Partición izquierda
Desde l a m

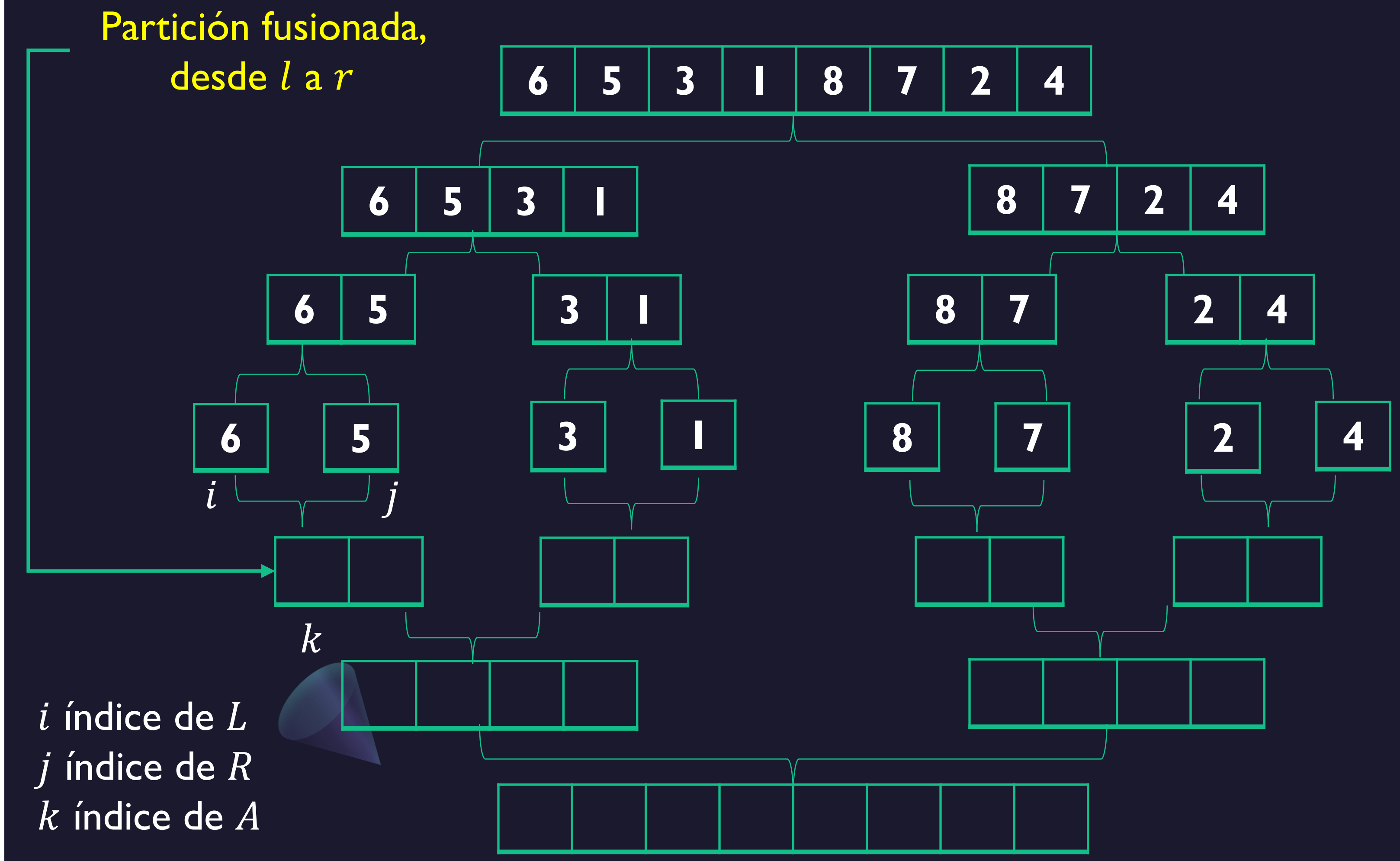
Partición derecha
Desde $m + 1$ hasta r

s_1 y s_2 : número de elementos en las particiones izquierda y derecha (líneas 4 y 5)

Algoritmos de ordenamiento: merge-sort

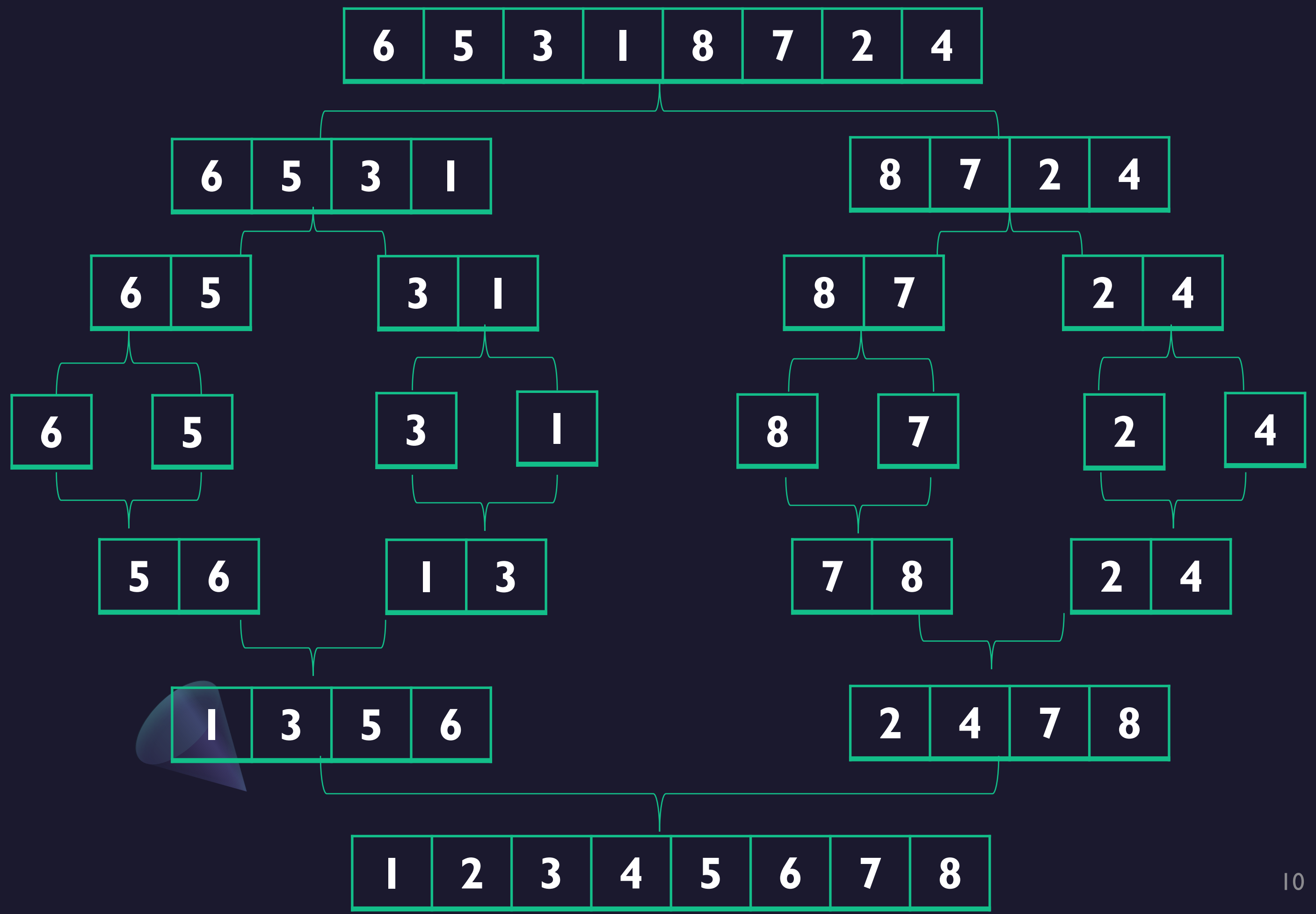
ALGORITHM $merge(A, l, m, r)$
// **Input:** Arreglo A índices de inicio l,
punto medio m y final r de partición
// **Output:** A ordenado

B = copia de $A[L, \dots, M]$
C = copia de $A[M+1, \dots, R]$
s1, s2 = tamaños de B y C
$i, k, k = L$
while $i < s1$ **and** $j < s2$
    if $B[i] < C[j]$
        $A[k] = B[i]$
        i++
    else
        A[k] = C[j]
        j++
while i < s1
    $A[k] = B[i]$
    i++ y k++
while $j < s2$
    $A[k] = C[j]$
    j++ y k++

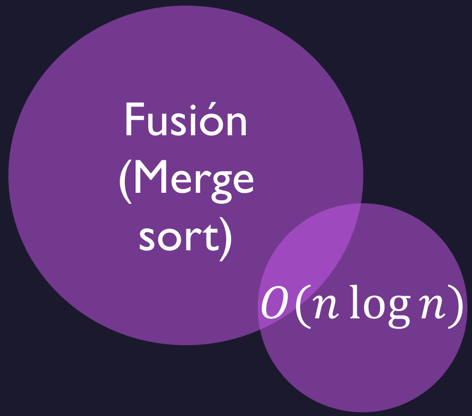


Algoritmos de ordenamiento: merge-sort

```
ALGORITHM merge(A, l, m, r)
// Input: Arreglo A índices de inicio l,
punto medio m y final r de partición
// Output: A ordenado

B = copia de A[L, ... , M]
C = copia de A[M + 1, ... , R]
s1, s2 = tamaños de B y C
i, k, k = L
while i < s1 and j < s2
        if B[i] < C[j]
                A[k] = B[i]
                i++
        else
                A[k] = C[j]
                j++
while i < s1
        A[k] = B[i]
        i++ y k++
while j < s2
        A[k] = C[j]
        j++ y k++
```



Algoritmos de ordenamiento: merge-sort



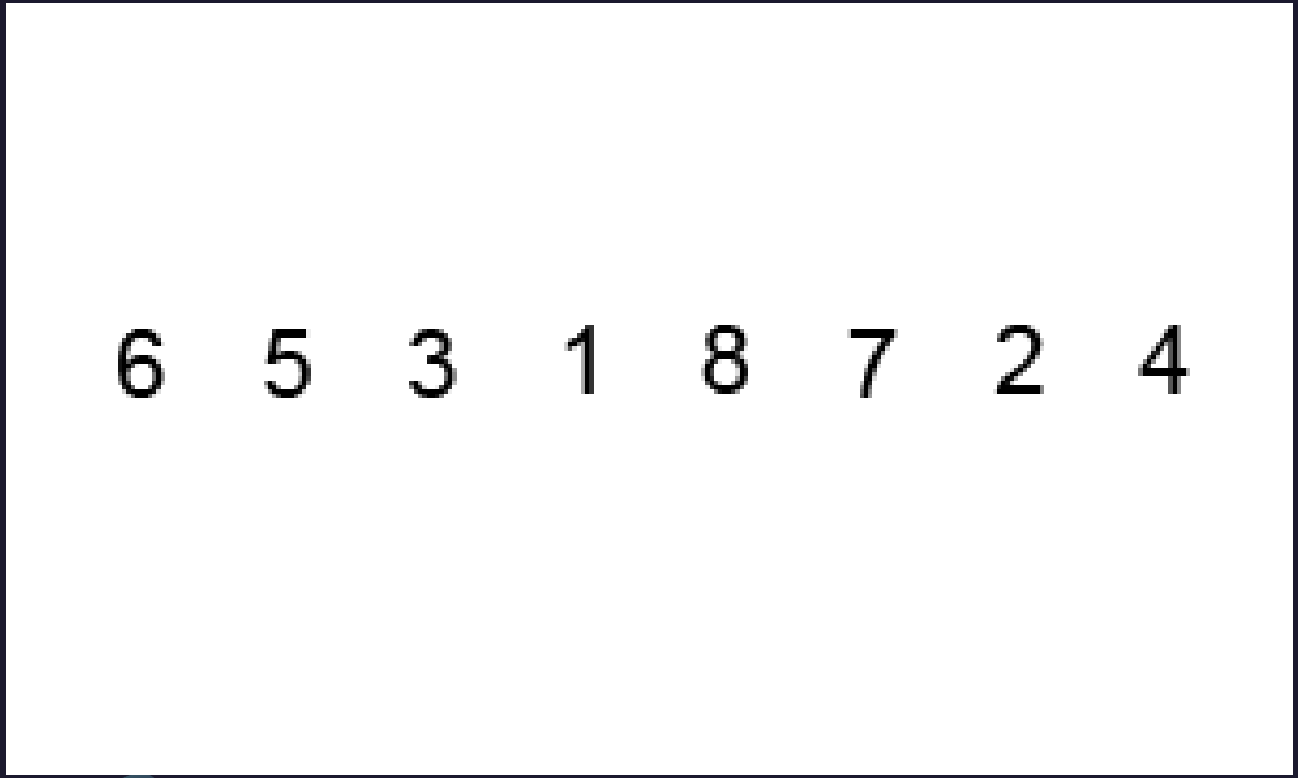

```
ALGORITHM mergeSort(A, L, R)
// Input: Arreglo A, índices de inicio L, y
final R de partición
// Output: A ordenado de L a R

if L < R then
        M = L + (R − L) / 2
        mergeSort(A, L, M)
        mergeSort(A, M + 1, R)
        merge(A, L, M, R)
```

```
ALGORITHM merge(A, l, m, r)
// Input: Arreglo A índices de inicio l,
punto medio m y final r de partición
// Output: A ordenado

B = copia de A[L, … , M]
C = copia de A[M + 1, … , R]
s1, s2 = tamaños de B y C
i, k, k = L
while i < s1 and j < s2
        if B[i] < C[j]
                A[k] = B[i]
                i++
        else
                A[k] = C[j]
                j++
while i < s1
        A[k] = B[i]
        i++ y k++
while j < s2
        A[k] = C[j]
        j++ y k++
```

Algoritmos de ordenamiento: quick sort

Quick sort

$O(n^2)$



Usa el valor de un elemento como *pivote* para crear particiones recursivamente.

Los elementos menores o mayores que el pivote se colocan a su izquierda o derecha.

Algoritmos de ordenamiento: quick sort

- El **pivote** puede ser elegido de diferentes formas:
  - aleatoria
  - punto inicial
  - medio, o
  - final de la partición,
  - valor mediana

A	6	5	3	1	8	7	2	4

$l = 0$

punto de inicio

punto medio

$m = l + (r - l) / 2$

$r = \mathrm{n} - 1$

punto final

Partición inicial: todo el arreglo, comprendido entre $l = 0$ y $r = n - 1$

Algoritmos de ordenamiento: quick sort

- El **pivote** puede ser elegido de diferentes formas:
  - aleatoria
  - punto inicial
  - Punto medio
  - final de la partición (veremos este ejemplo),
  - valor mediana

A	6	5	3	1	8	7	2	4

$l = 0$

punto de inicio

punto medio

$m = l + (r - l) / 2$

$r = \mathrm{n} - 1$

punto final

Partición inicial: todo el arreglo, comprendido entre $l = 0$ y $r = n - 1$

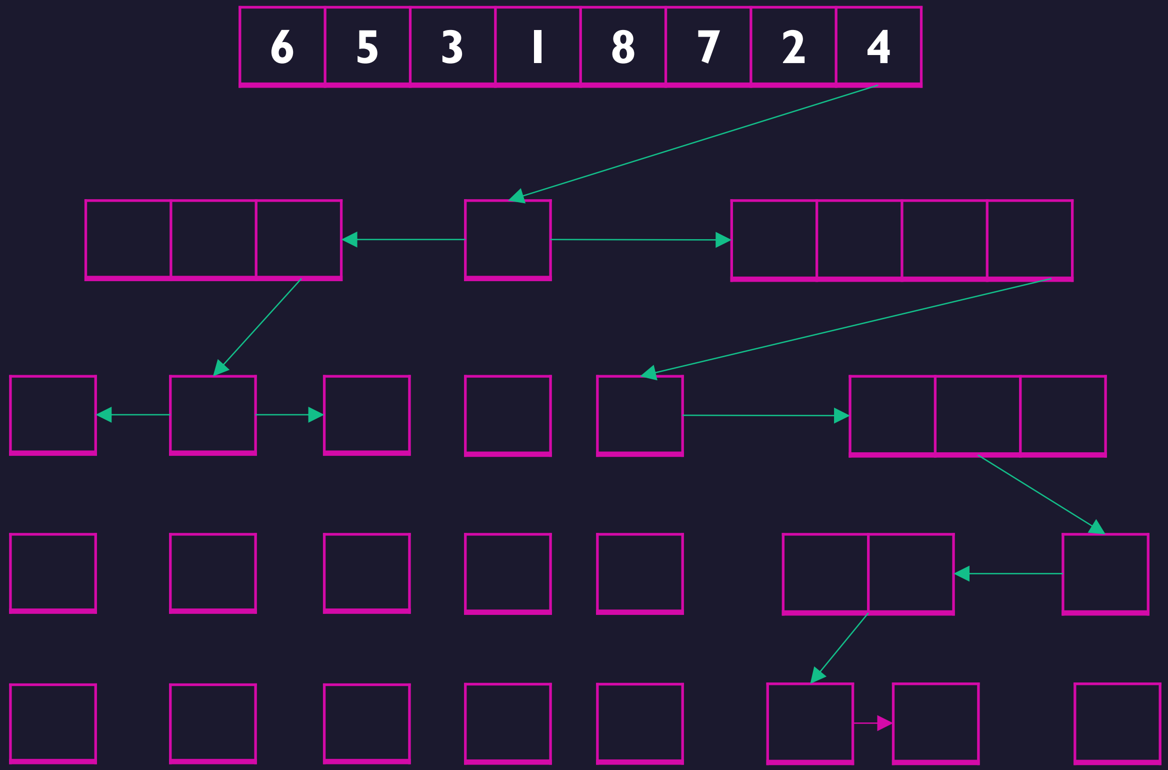
```
ALGORITHM quickSort(A, L, R)
// Input: Arreglo A, índices de inicio L, y final R de partición
// Output: A ordenado de L a R

if L < R then
        p_index = partition(A, L, R)
        quickSort(A, L, p_index − 1)
        quickSort(A, p_index + 1, R)
```

Algoritmos de ordenamiento: quick sort

Pivote: r el último elemento en la partición

ALGORITHM *partition* (A, L, R)
// **Input:** Arreglo A, índices de inicio L, y final R de partición
// **Output:** Partición con elementos con elementos menores
que pivote a la derecha y mayores a la izquierda

$pivot = A[R]$
$i = L - 1$
for $j = 0$ **to** $R - 1$
    if $A[j] < pivote$ **then**
        Aumenta $i = i + 1$
        Intercambia $A[i]$ con $A[j]$
Intercambia $A[i + 1]$ con $A[r]$
return $i + 1$ // Índice donde quedo pivot

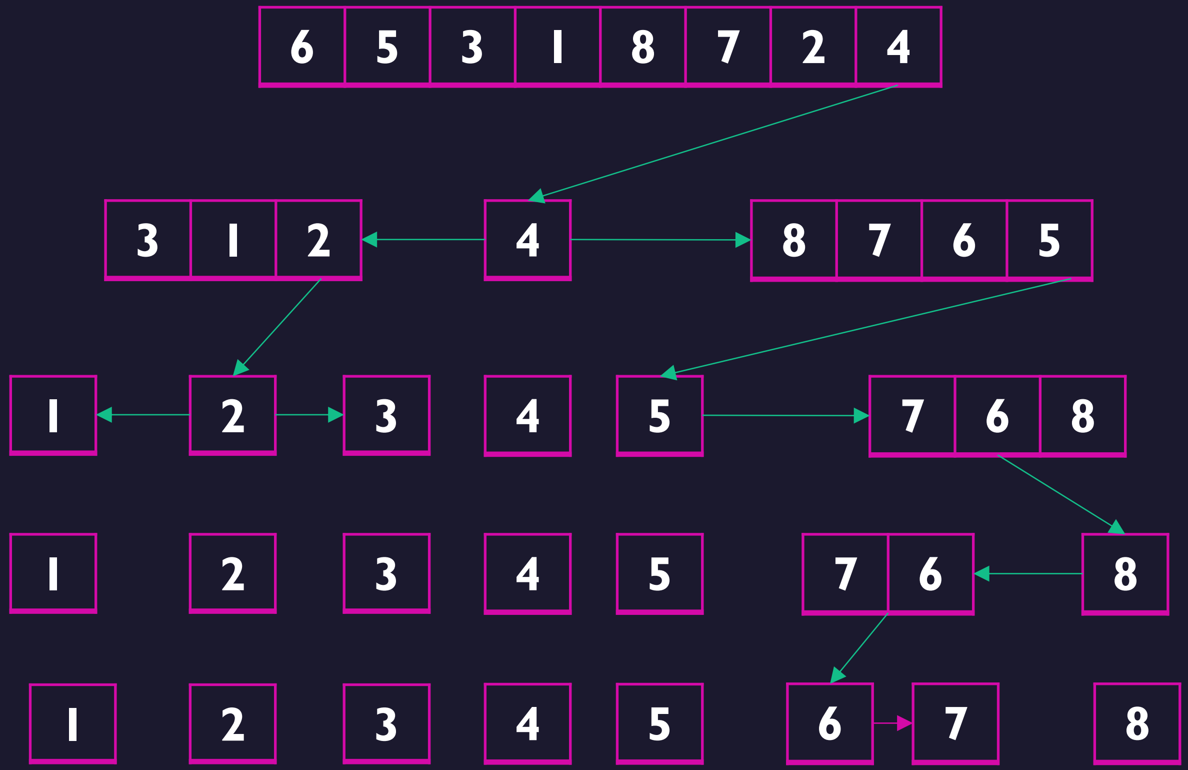


Algoritmos de ordenamiento: quick sort

Pivote: r el último elemento en la partición

ALGORITHM *partition* (A, L, R)
// **Input:** Arreglo A, índices de inicio L, y final R de partición
// **Output:** Partición con elementos con elementos menores
que pivote a la derecha y mayores a la izquierda

$pivot = A[R]$
$i = L - 1$
for $j = 0$ **to** $R - 1$
    if $A[j] < pivote$ **then**
        Aumenta $i = i + 1$
        Intercambia $A[i]$ con $A[j]$
Intercambia $A[i + 1]$ con $A[r]$
return $i + 1$ // Índice donde quedo pivot



Algoritmos de ordenamiento: quick sort

Quick-sort

$O(n^2)$

ALGORITHM $quickSort(A, L, R)$
// **Input:** Arreglo A, índices de inicio L, y final R de
partición
// **Output:** A ordenado de L a R

if $L < R$ **then**
    $p\_index = partition(A, L, R)$
    $quickSort(A, L, p\_index - 1)$
    $quickSort(A, p\_index + 1, R)$

ALGORITHM $partition\ (A, L, R)$
// **Input:** Arreglo A, índices de inicio L, y final R de partición
// **Output:** Partición con elementos con elementos menores
que pivote a la derecha y mayores a la izquierda

$pivot = A[R]$
$i = L - 1$
for $j = 0$ **to** $R - 1$
    if $A[j] < pivote$ **then**
        Aumenta $i = i + 1$
        Intercambia $A[i]$ con $A[j]$
Intercambia $A[i + 1]$ con $A[r]$
return $i + 1$ // Índice donde quedo pivot

Algoritmos de ordenamiento: quick sort

Quick-sort

$O(n^2)$

Una implementación en Python, con un enfoque de mas alto nivel

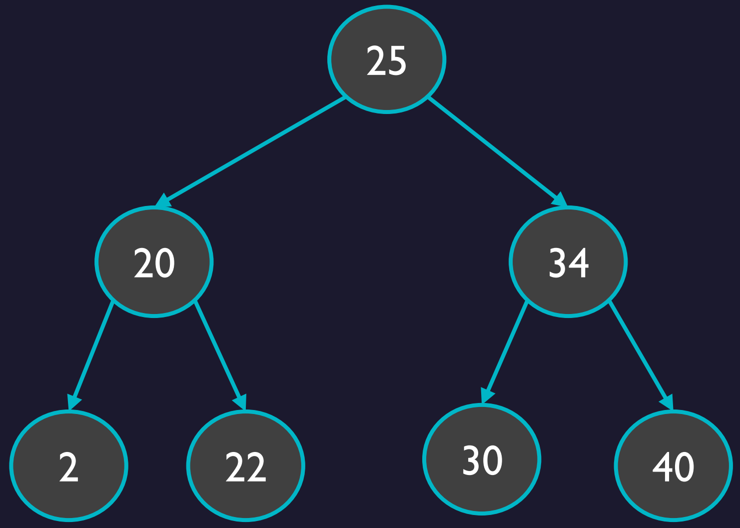
```python
def quicksort(arr):
  if len(arr) <= 1:
      return arr
  pivot = arr[len(arr) // 2]  # Choose the middle element as the pivot
  left = [x for x in arr if x < pivot]  # Elements less than pivot
  middle = [x for x in arr if x == pivot]  # Elements equal to pivot
  right = [x for x in arr if x > pivot]  # Elements greater than pivot
  return quicksort(left) + middle + quicksort(right)

# Example usage:
arr = [3, 6, 8, 10, 1, 2, 1]
sorted_arr = quicksort(arr)
print(sorted_arr)
```

Recorridos en arboles BST: preorder

```
void BST::preorder(NodeTree *auxRoot)
{   if (auxRoot == NULL)
    {   return;      }

    cout << auxRoot->data << "\t" ;
    preorder(auxRoot->left);
    preorder(auxRoot->right);
}
```

nodo-izquierda-derecha

$n - l - r$



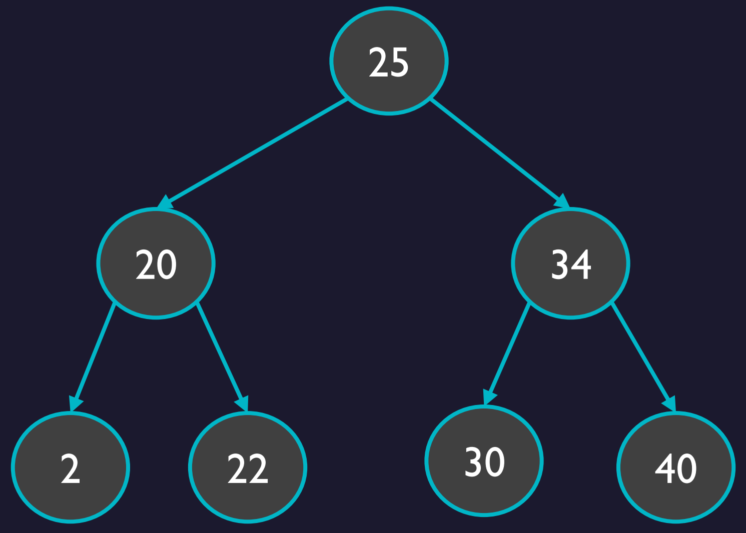
¿Qué complejidad tiene el recorrido?

Recorridos en arboles BST: inorder

```
void BST::inorder(NodeTree *auxRoot)
{   if (auxRoot == NULL)
    {   return;     }

    inorder(auxRoot->left);
    cout << auxRoot->data << "\t" ;
    inorder(auxRoot->right);
}
```

izquierda-nodo-derecha

$l - n - r$

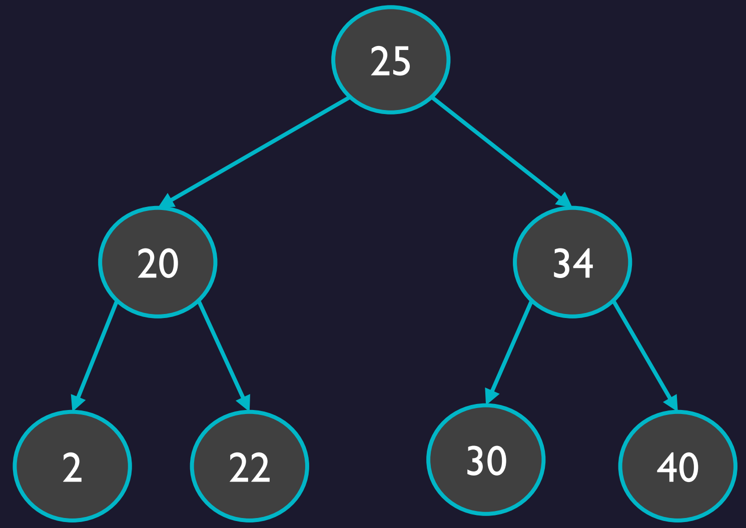


Recorridos en arboles BST: postorder

```
void BST::postorder(NodeTree *auxRoot)
{   if (auxRoot == NULL)
    {   return;     }

    postorder(auxRoot->left);
    postorder(auxRoot->right);
    cout << auxRoot->data << "\t" ;
}
```

izquierda-derecha-nodo

$l - r - n$



Recorridos en arboles BST

Recorrido	orden	Utilidad
Preorder	nodo-izquierda-derecha 25 20 2 22 34 30 40	Generar una replica
Inorder	izquierda-nodo-derecha 2 20 22 25 30 34 40	Elementos del árbol en orden
postorder	izquierda-derecha-nodo 2 22 20 30 40 34 25	Liberar los nodos