Ordenación

AED I

2020

UBP UNIVERSIDAD BLAS PASCAL

Algoritmos de ordenación

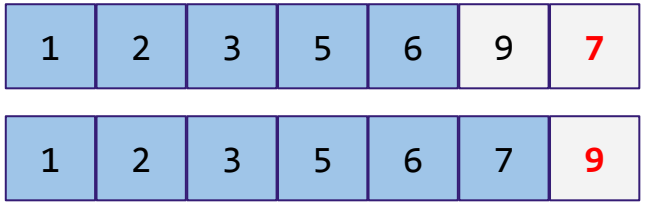
ordenamiento por selección

- Es el algoritmo de ordenación más sencillo (pero no el más rápido),
- Selecciona el menor de todos, lo coloca en el primer lugar apartándolo del resto,
- Selecciona el menor de todos los restantes, lo coloca en el segundo lugar apartándolo del resto,
- Selecciona el menor de todos los restantes, lo coloca en el tercer lugar apartándolo del resto,
- . . . (en cada uno de estos pasos ordena un elemento) . . .hasta terminar.

ordenamiento por selección: pseudocódigo

```
Repetir hasta que no queden elementos no-ordenados:
    Buscar el menor elemento de la parte no-ordenada
    Intercambiar el menor elemento con el primer elemento de la parte
    no-ordenada
```

ordenamiento por selección



ordenamiento por selección: análisis

Peor caso: tenemos que iterar sobre todos los elementos del array para encontrar el menor y tenemos que repetir ese proceso n veces para asegurarnos de que todos los elementos han sido procesados

Mejor caso: el mismo

$$O(n^2)$$
$$\Omega(n^2)$$

Ejercicio 4.01.

Implementar ordenamiento por selección

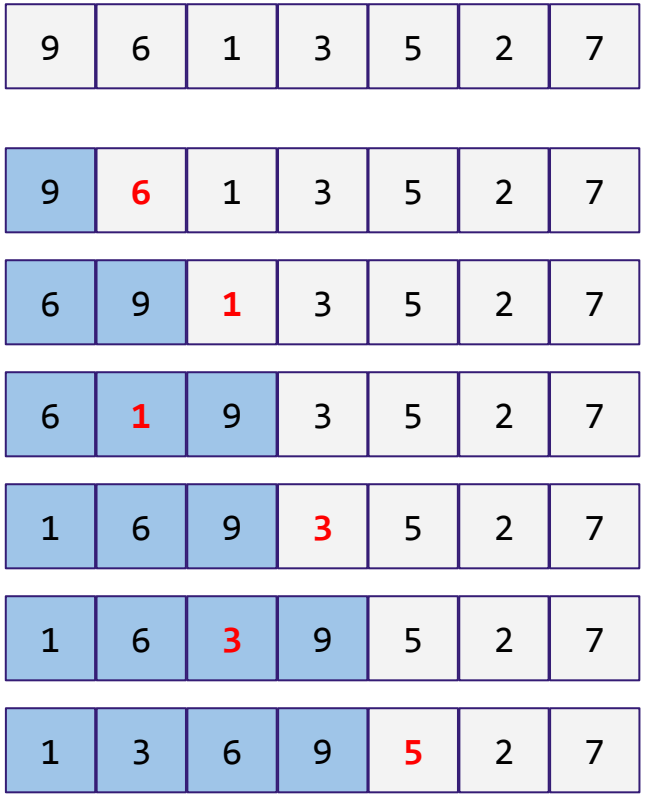
ordenamiento por inserción

- Se procede como cuando tenemos un manojo de cartas en la mano y necesitamos ordenarlas
- Vamos formando un subconjunto de cartas ordenadas, al que cuando agregamos una carta del subconjunto desordenado, la insertamos en el lugar que corresponde

ordenamiento por inserción: pseudocódigo

```
Consideramos el primer elemento como ordenado
Repetir hasta que todos los elementos estén ordenados:
    Seleccionar el primer elemento no-ordenado
    Insertarlo en la parte ordenada del array desplazando los elementos que
    haga falta desplazar para hacerle lugar
```

ordenamiento por inserción



ordenamiento por inserción: análisis

Peor caso: el array está en orden invertido. Entonces tenemos que hacer n intercambios para que cada elemento del array ocupe su lugar definitivo.

Mejor caso: el array está ordenado. No hacemos ningún intercambio

$$O(n^2)$$
$$\Omega(n)$$

Ejercicio 4.02.

Implementar ordenamiento por inserción

Una vez implementado haga los cambios necesarios para que ordene en orden decreciente

ordenamiento por burbuja (bubble)

Se trata de mover los elementos mayores tan a la derecha hasta donde lleguen o los menores tan a la izquierda como les corresponda (de ahí el nombre de burbuja)

ordenamiento por burbuja: pseudocódigo

```
Inicializar el contador de intercambios en -1
Repetir hasta que el contador de intercambios es 0:
    Inicializar el contador de intercambios en 0
    Seleccionar cada par adyacente, si los elementos no están en orden:
        Intercambiar los elementos
        Incrementar el contador de intercambios
```

ordenamiento por burbuja

	9	6	1	3	5	2	7
-1	9	6	1	3	5	2	7
6	6	1	3	5	2	7	9
4	1	3	5	2	6	7	9
1	1	3	2	5	6	7	9
1	1	2	3	5	6	7	9
0	1	2	3	5	6	7	9

ordenamiento por burbuja: análisis

Peor caso: el array está en orden invertido. Entonces tenemos que hacer que cada elemento flote hasta su lugar definitivo en n intercambios.

Mejor caso: el array está ordenado. En la primera pasada no se detecta ningún intercambio y el procesamiento se detiene

$$O(n^2)$$
$$\Omega(n)$$

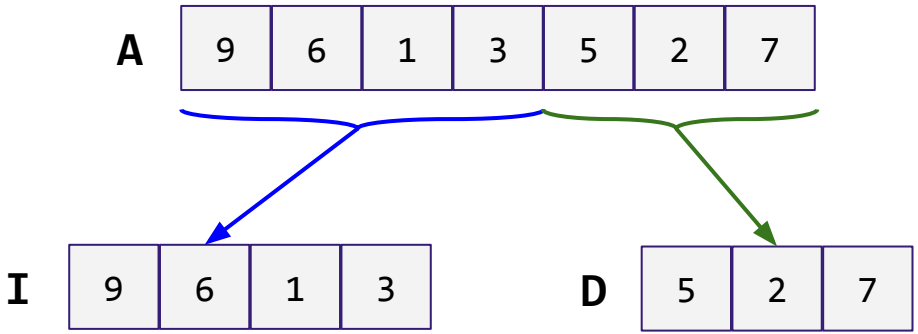
Ejercicio 4.03.

Implementar ordenamiento por burbuja

ordenamiento por combinación (mergesort)

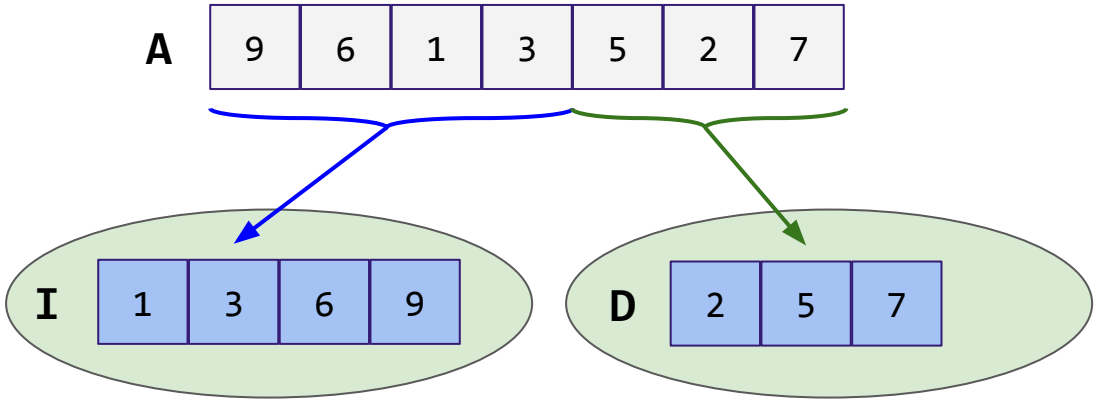
- Creado en 1945 por John Von Neumann
- Está basado en el paradigma **divide y vencerás**
- En términos generales consiste en dividir el arreglo desordenado repetidamente hasta llegar a tener subarreglos de 1 elemento
- Luego se **combinan** estos subarreglos de forma de obtener otro subarreglo ordenado hasta llegar a producir el arreglo original ordenado

mergesort



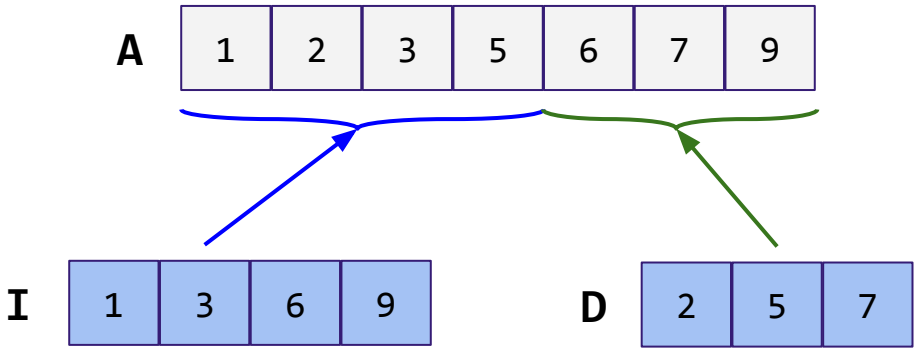
Dividimos el arreglo en dos subarreglos de tamaño aproximadamente igual

mergesort



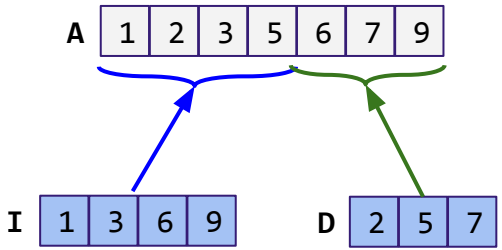
Ordenamos los subarreglos de alguna manera

mergesort



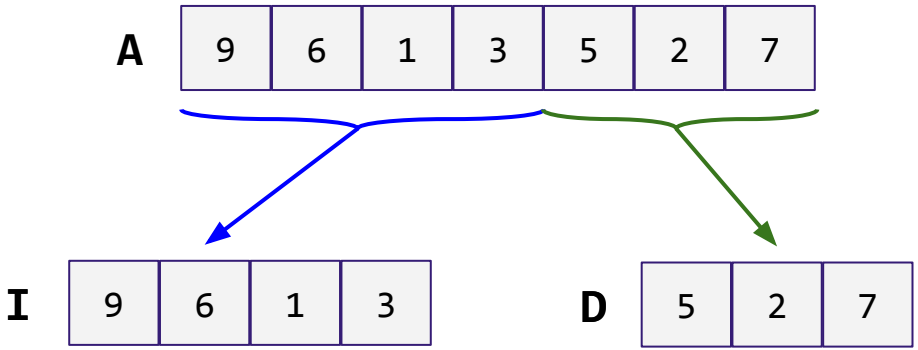
Combinamos los subarreglos ordenados para formar el arreglo original ordenado

mergesort



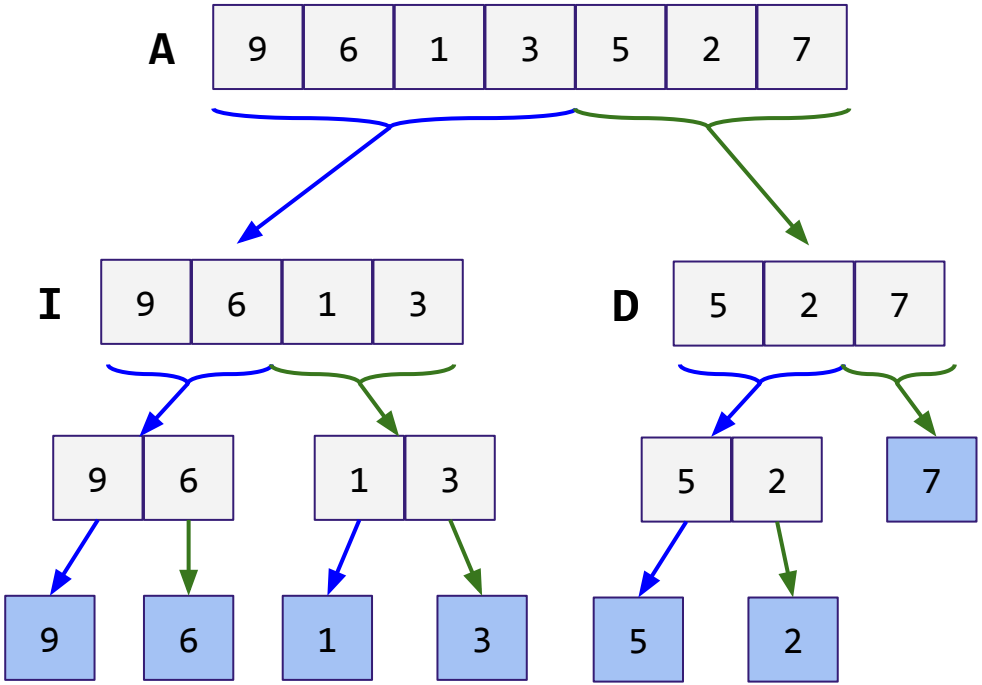

```
merge(I, D, A, nI, nD, n){

    i=0; j=0; k=0;

    mientras(i < nI && j < nD){
        si (I[i] < D[j]){
            A[k] = I[i]; k++; i++;
        }
        sino {
            A[k] = D[j]; k++; j++;
        }
    }
    mientras(i < nI){A[k] = I[i]; k++; i++;}
    mientras(j < nD){A[k] = D[j]; k++; j++;}
}
```

mergesort

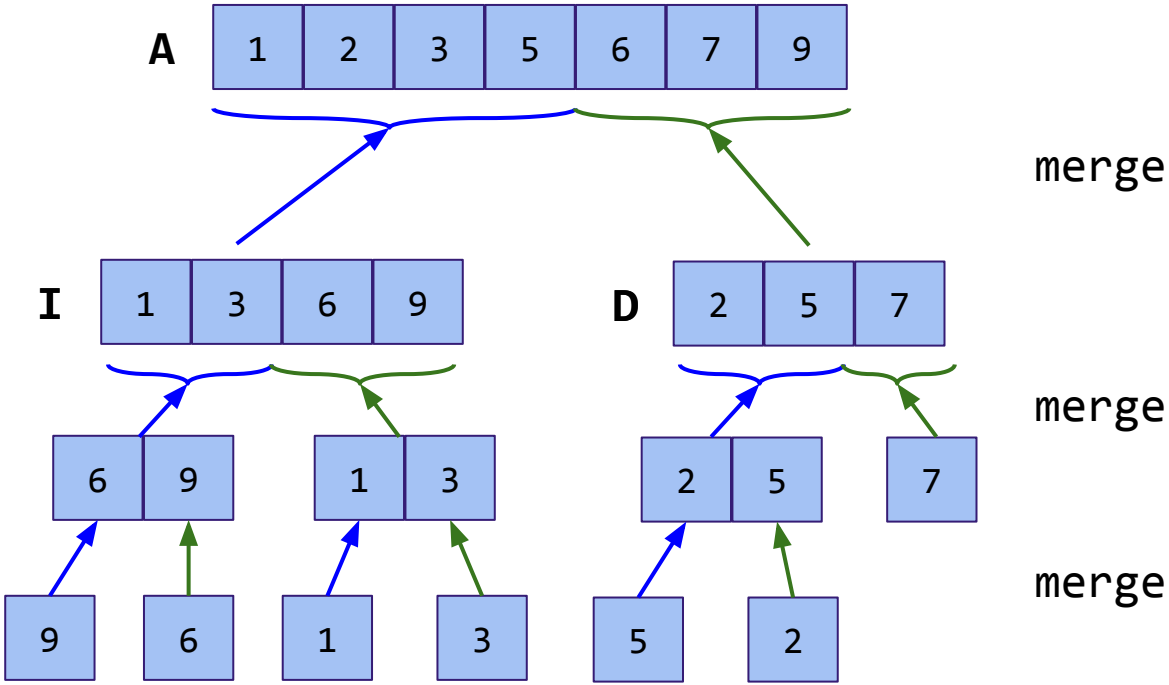


No dijimos **cómo ordenamos** los subarreglos

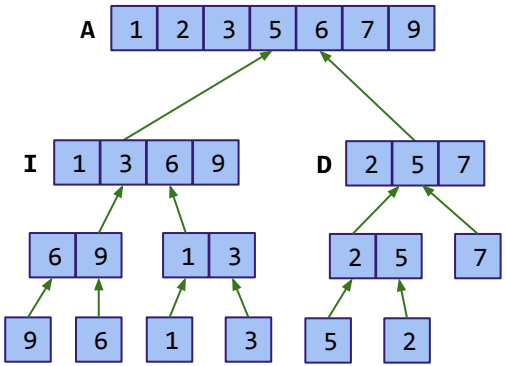
mergesort



mergesort



mergesort




```
mergesort(A, n){
    si (n < 2) retornar;
    mitad = n/2;
    izquierdo = nuevo array[mitad];
    derecho = nuevo array[n - mitad];

    para (i=0 hasta i=mitad)
        {izquierdo[i] = A[i]}
    para (i=mitad hasta n)
        {derecho[i - mitad] = A[i]}

    mergesort(izquierdo, mitad);
    mergesort(derecho, n-mitad);
    merge(izquierdo, derecho, A, mitad, n-mitad, n);
}
```

mergesort

```
merge(I, D, A, nI, nD, n){

      i=0; j=0; k=0;

      mientras(i < nI && j < nD){
            si (I[i] < D[j]){
                  A[k] = I[i]; k++; i++;
            }
            sino {
                  A[k] = D[j]; k++; j++;
            }
      }
      mientras(i < nI){A[k] = I[i]; k++; i++;}
      mientras(j < nD){A[k] = D[j]; k++; j++;}
}
```

```
mergesort(A, n){
      si (n < 2) retornar;
      mitad = n/2;
      izquierdo = nuevo array[mitad];
      derecho = nuevo array[n - mitad];

      para (i=0 hasta i=mitad)
            {izquierdo[i] = A[i]}
      para (i=mitad hasta n)
            {derecho[i - mitad] = A[i]}

      mergesort(izquierdo, mitad);
      mergesort(derecho, n-mitad);
      merge(izquierdo, derecho, A, mitad, n-mitad, n);
}
```

mergesort: complejidad

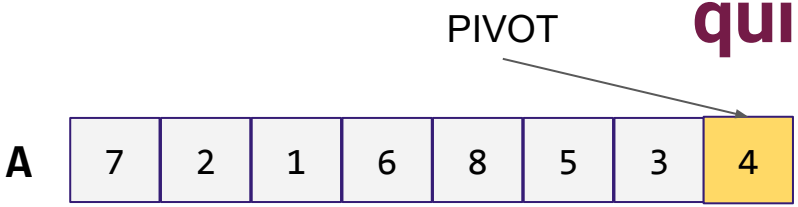
O(n logn)
Ω(n logn)

Almacenamiento: O(n)
no es un método in-place

Ejercicio 4.04.

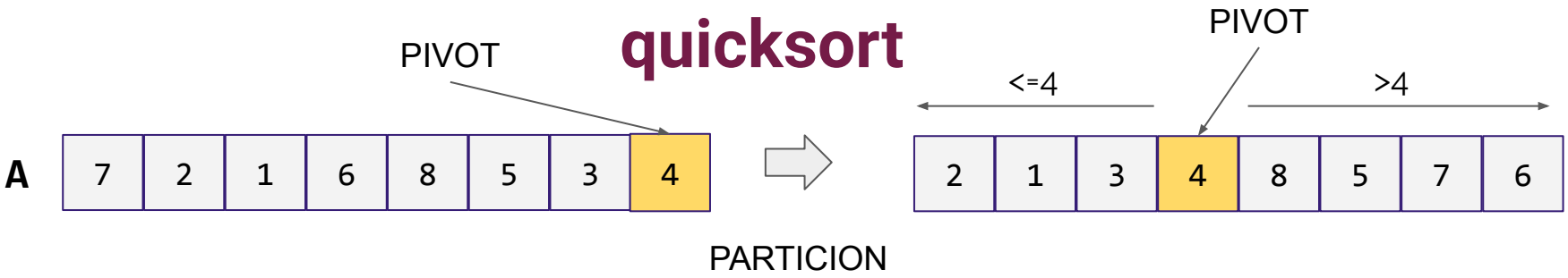
Implementar mergesort

quicksort



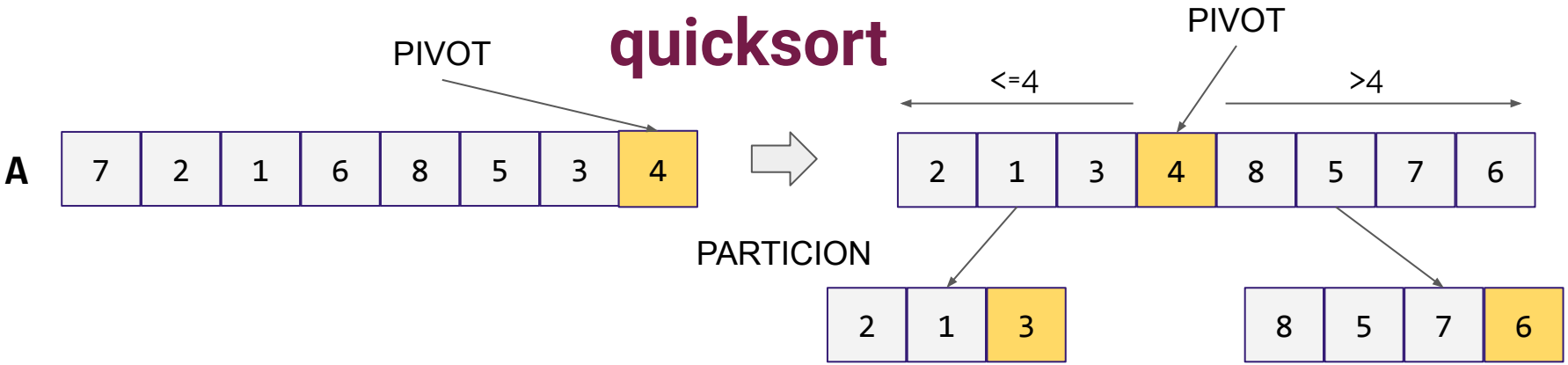
Seleccionamos un elemento arbitrario que llamamos **Pivot**

quicksort



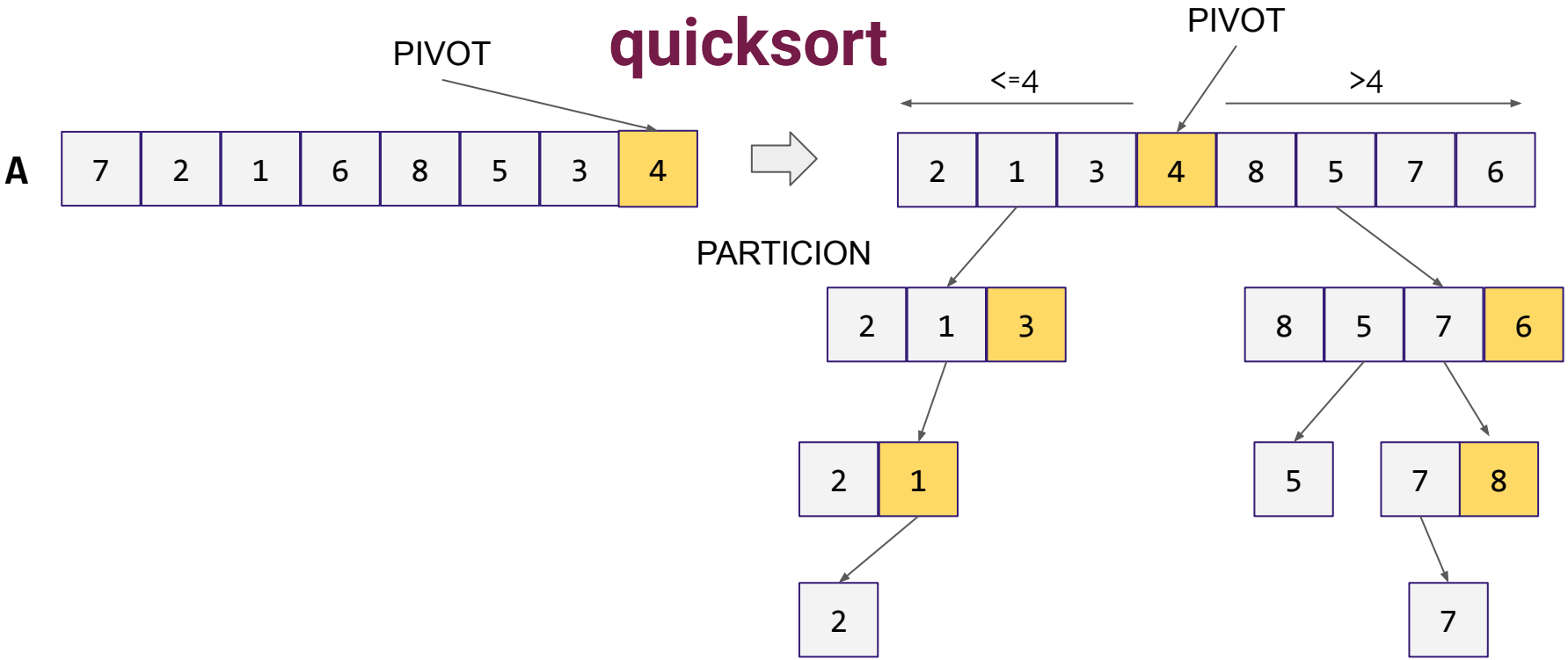
Particionamos el arreglo ubicando los elementos menores que el **pivot** a su izquierda y aquellos que son mayores a la derecha

quicksort



En este momento hemos dividido el problema en dos subproblemas más pequeños. Esto se hace dentro del mismo arreglo

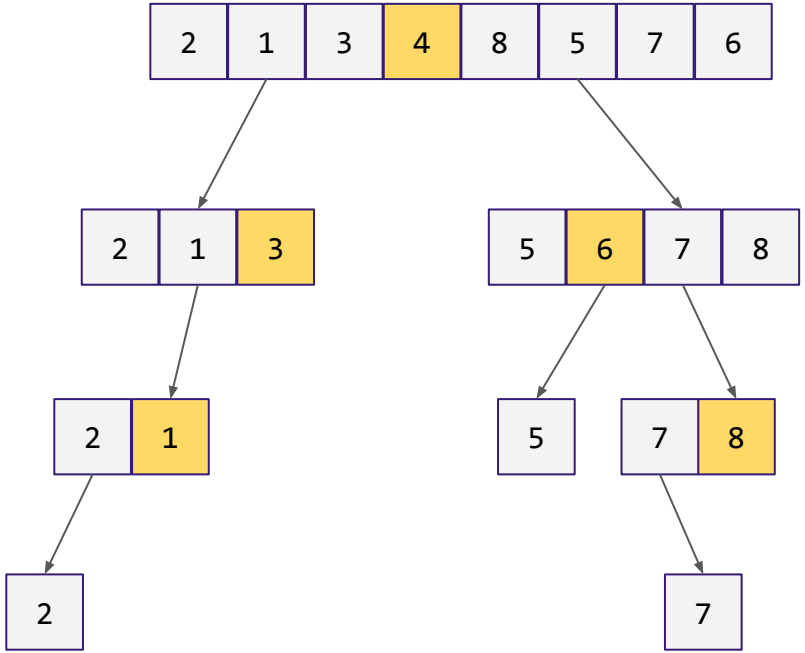
quicksort



Realizaremos este proceso de forma recursiva

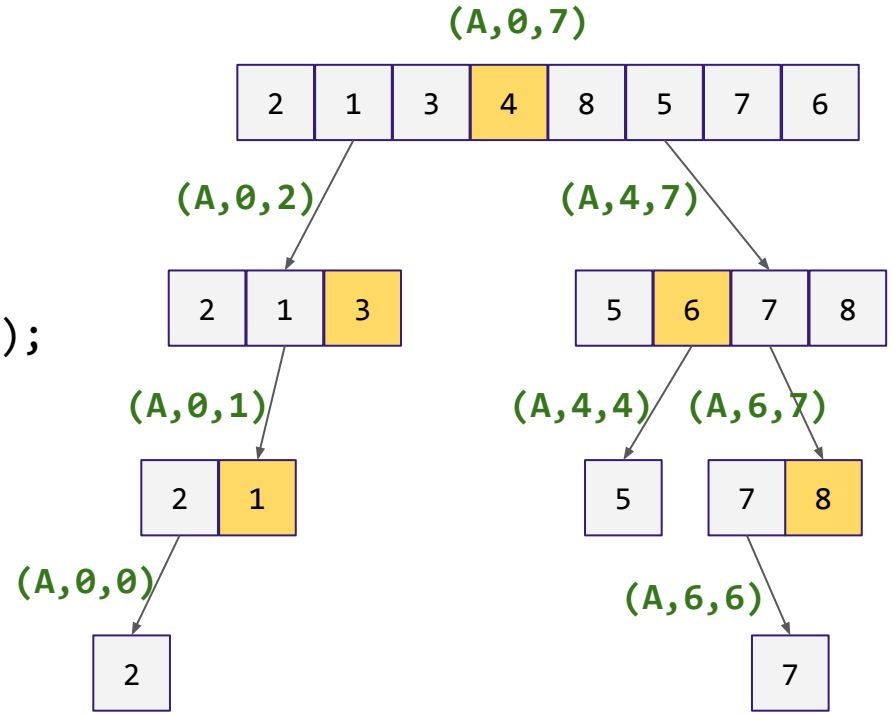
quicksort

```
quicksort(A, principio, fin){
    si (principio < fin){
        p = particion(A, principio, fin);
        quicksort(A, principio, p-1);
        quicksort(A, p+1, fin);
    }
}
```

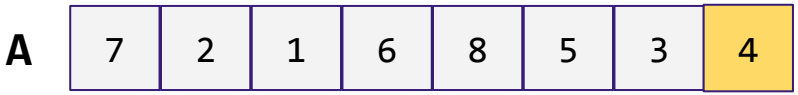


quicksort

```
quicksort(A, principio, fin){
    si (principio < fin){
        p = particion(A, principio, fin);
        quicksort(A, principio, p-1);
        quicksort(A, p+1, fin);
    }
}
```

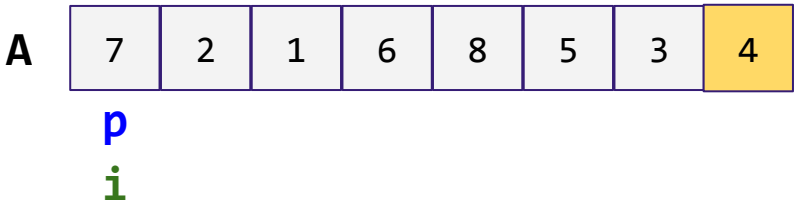


quicksort



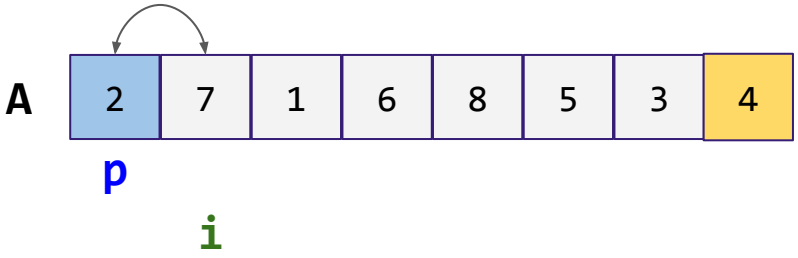

```
particion(A, principio, fin){
    pivot = A[fin];
    p = principio;
    para (i = principio hasta fin - 1){
        si (A[i] <= pivot){
            intercambiar(A[i], A[p]);
            p++;
        }
    }
    intercambiar(A[p], A[fin]);
    retornar p;
}
```

quicksort



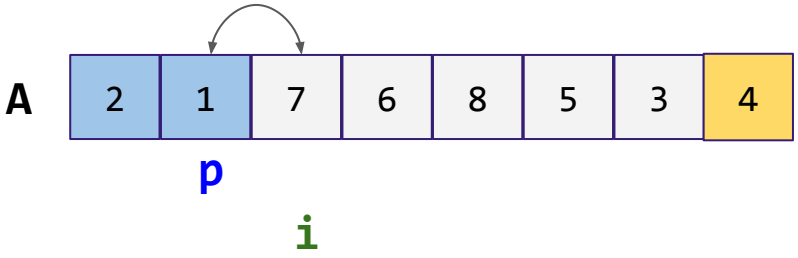

```
particion(A, principio, fin){
    pivot = A[fin];
    p = principio;
    para (i = principio hasta fin - 1){
        si (A[i] <= pivot){
            intercambiar(A[i], A[p]);
            p++;
        }
    }
    intercambiar(A[p], A[fin]);
    retornar p;
}
```

quicksort



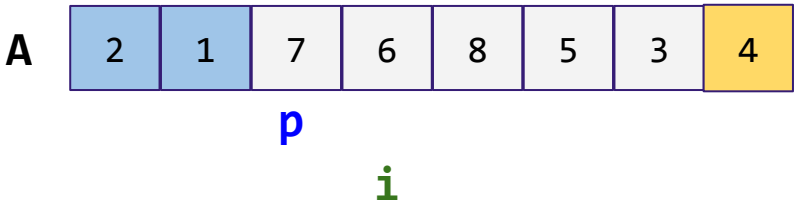

```
particion(A, principio, fin){
    pivot = A[fin];
    p = principio;
    para (i = principio hasta fin - 1){
        si (A[i] <= pivot){
            intercambiar(A[i], A[p]);
            p++;
        }
    }
    intercambiar(A[p], A[fin]);
    retornar p;
}
```

quicksort



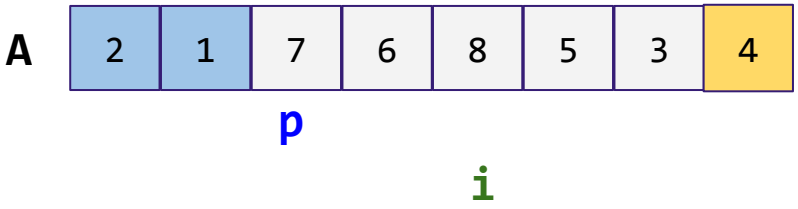

```
particion(A, principio, fin){
    pivot = A[fin];
    p = inicio;
    para (i = principio hasta fin - 1){
        si (A[i] <= pivot){
            intercambiar(A[i], A[p]);
            p++;
        }
    }
    intercambiar(A[p], A[fin]);
    retornar p;
}
```

quicksort



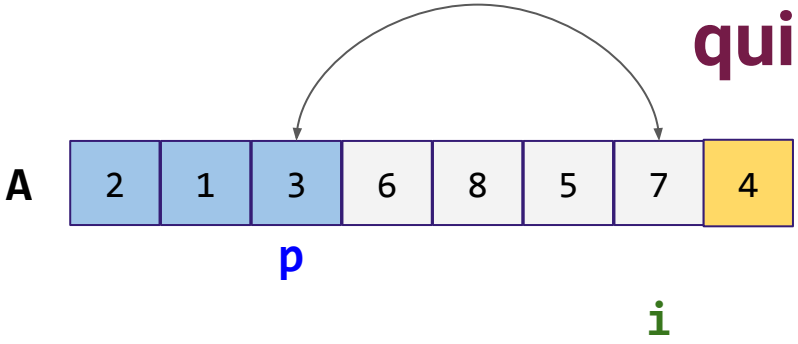

```
particion(A, principio, fin){
    pivot = A[fin];
    p = principio;
    para (i = principio hasta fin - 1){
        si (A[i] <= pivot){
            intercambiar(A[i], A[p]);
            p++;
        }
    }
    intercambiar(A[p], A[fin]);
    retornar p;
}
```

quicksort



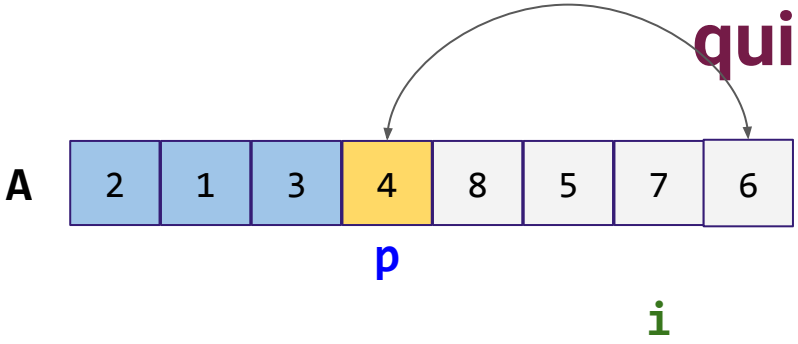

```
particion(A, principio, fin){
    pivot = A[fin];
    p = principio;
    para (i = principio hasta fin - 1){
        si (A[i] <= pivot){
            intercambiar(A[i], A[p]);
            p++;
        }
    }
    intercambiar(A[p], A[fin]);
    retornar p;
}
```

quicksort




```
particion(A, principio, fin){
    pivot = A[fin];
    p = principio;
    para (i = principio hasta fin - 1){
        si (A[i] <= pivot){
            intercambiar(A[i], A[p]);
            p++;
        }
    }
    intercambiar(A[p], A[fin]);
    retornar p;
}
```

quicksort




```
particion(A, principio, fin){
    pivot = A[fin];
    p = principio;
    para (i = principio hasta fin - 1){
        si (A[i] <= pivot){
            intercambiar(A[i], A[p]);
            p++;
        }
    }
    intercambiar(A[p], A[fin]);
    retornar p;
}
```

quicksort

```
quicksort(A, principio, fin){
     si (principio < fin){
          p = particion(A, principio, fin);
          quicksort(A, principio, p-1);
          quicksort(A, p+1, fin);
     }
}
```

```
particion(A, principio, fin){
     pivot = A[fin];
     p = principio;
     para (i = principio hasta fin-1){
          si (A[i] <= pivot){
               intercambiar(A[i], A[p]);
               p++;
          }
     }
     intercambiar(A[p], A[fin]);
     retornar p;
}
```

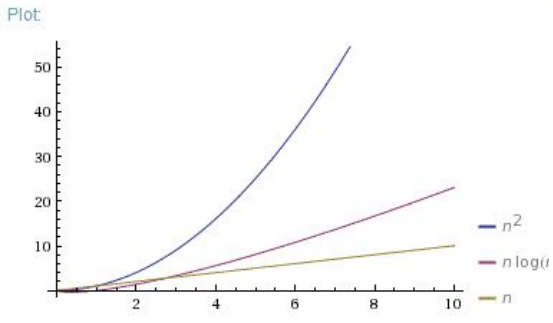
quicksort: complejidad

O(n logn) caso promedio
O(n^2) peor caso

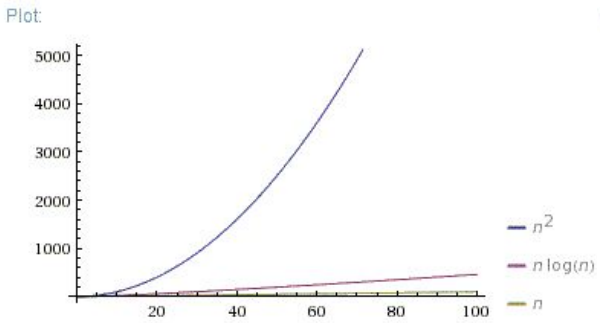
Es un método in-place

$O(n^2)$ vs $O(n \log n)$

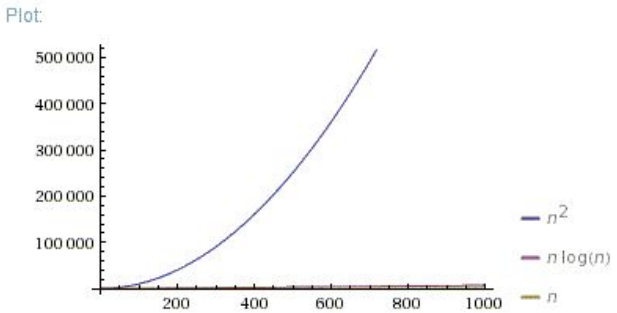
n = [0..10]



n = [0..100]



n = [0..1000]



Ejercicio 4.05.

Implementar quicksort

Ejercicio 4.06.

Implementar ordenación por inserción en una lista enlazada simple

Ejercicio 4.07.

Encontrar el número que falta en un array

Ejemplo

input

a = {5,1,4,3,7,6}

output

2

Ejercicio 4.08.

Ordenar valores por su frecuencia

Ejemplo

Input 3, 4, 3, 3, 5, 2, 4, 6

Output 3, 4, 2, 5, 6

Pista: debe usar más de un array

Ejercicio 4.09.

Ordenar un array usando pilas

Ayuda: pseudocodigo

```
declarar pila1, pila2, temp
meter el array en pila1
mientras pila1 no esté vacía{
     temp = pila1.pop()
     mientras (pila2 no esté vacía y temp < pila2.arriba){
          pila1.push(pila2.pop())
     }
     pila2.push(temp)]
}
El array ordenado está en pila2
```

Ejercicio 4.09.

```
void ordenarConPilas(int a[], int n){
    Pila p1, p2;
    int temp;
    for (int i = 0; i < n; i++)
    {
       p1.poner(a[i]);
    }
    while(!p1.estaVacia()){
       temp = p1.sacar();
       while (!p2.estaVacia() && temp < p2.arriba()){
            p1.poner(p2.sacar());
       }
       p2.poner(temp);
    }
    for (int i = n-1 ; i >= 0 ; i--)
    {
       a[i] = p2.sacar();
    }
}
```

Ejercicio 4.10.

Dado un conjunto de valores enteros en un array, escribí un programa que obtenga el par (o los pares) de números más cercanos entre sí.

Ejercicio 4.11.

Escribir una función que ordene una lista enlazada simple usando el método de la burbuja

lectura adicional

Animación mostrando el comportamiento de los distintos algoritmos de ordenación:
https://www.toptal.com/developers/sorting-algorithms

Por qué quicksort es mejor que los otros algoritmos de ordenación en la práctica:
https://cs.stackexchange.com/questions/3/why-is-quicksort-better-than-other-sorting-algorithms-in-practice

Radix Sort, el algoritmo que rompe la barrera de O(n logn)
https://en.wikibooks.org/wiki/Algorithm_Implementation/Sorting/Radix_sort
http://codercorner.com/RadixSortRevisited.htm