ORDENAMIENTOS

Definición del problema de ordenamientos

- **Entrada:**
  Una secuencia de N elementos $(a_1, a_2, \ldots, a_N)$

- **Salida:**
  Una permutación $(a_{1'}, a_{2'}, \ldots, a_{N'})$ de la secuencia de entrada de tal manera que $a_{1'} <= a_{2'} <= \ldots <= a_{N'}$ (ordenamiento ascendente)

Aspectos a considerar en el ordenamiento

- Tamaño
- Estabilidad
- Ordenamiento de claves

ALGORITMOS DE ORDENAMIENTO BASADOS EN COMPARACIONES

- Estos algoritmos involucran la comparación entre 2 objetos a y b para determinar una de las tres posibles relaciones entre ellos: menor que, igual o mayor que.

- El objetivo final consiste en entender las diferentes estrategias empleadas (algoritmo) para resolver el problema de ordenamiento y aplicarlas de manera análoga en algún otro problema con una estructura similar.

Ordenamiento de Burbuja (Bubble Sort)

- Velocidad: O(n^2), lento.
- Espacio requerido: El tamaño del arreglo inicial.
- Complejidad de codificación: Simple.
  - Este es el más simple y (desafortunadamente) el peor algoritmo de ordenamiento.

Ordenamiento de Burbuja (Bubble Sort)

- ***Idea básica del algoritmo*** :
  - Itera repetidamente a lo largo del arreglo, intercambiando parejas de elementos adyacentes cuando no se encuentran en orden.

- ***Comportamiento del algoritmo de burbuja (Negrita = región ordenada):***
  - 5 2 3 1 4
2 3 1 4 **5**
2 1 3 **4 5**
1 2 **3 4 5**
1 **2 3 4 5**
1 2 3 4 5 >> done

- ***Pseudocódigo:***
  - BubbleSort(A)

```
for i <- length[A]-1 down to 1
  for j <- 0 to i-1
    if (A[j] > A[j+1]) // change ">" to "<" to do a descending sort
      temp <- A[j]
      A[j] <- A[j+1]
      A[j+1] <- temp
```

- [299 - Train Swapping]
  [612 - DNA Sorting]
  [10327 - Flip Sort]

Ordenamiento por Selección (Selection Sort)

- Velocidad: $O(n^2)$, lento.
- Espacio requerido: El tamaño del arreglo inicial.
- Complejidad de codificación: Simple.

Ordenamiento por Selección (Selection Sort)

- ***Idea básica del algoritmo*** :
  - Itera repetidamente a lo largo del arreglo, en cada iteración selecciona al elemento más pequeño o más grande y lo coloca en su posición final ordenada.

- ***Comportamiento del algoritmo de selección (Negrita = región ordenada):***
  - 5 2 1 3 4
1 2 5 3 4
1 2 5 3 4
1 2 3 5 4
1 2 3 4 5 >> done

Ordenamiento por Selección (Selection Sort)

- ***Pseudocódigo:***
  - SelectionSort(A)

```
 for i <- 0  to  length[A]-2
     menor <- i
     for j <- i+1 to length[A]- 1
         if (A[j] < A[menor]) // change ">" to "<" to do a descending sort
             menor <- j
     temp <- A[i]
     A[i ]<- A[menor]
     A[menor] <- temp
```

- [120 – Stack of FlapJacks]()

Ordenamiento por inserción (Insertion Sort)

- Velocidad: $O(n^2)$ lento, pero ligeramente más rápido que los algoritmos anteriores en un arreglo ordenado parcialmente.
- Espacio requerido: El tamaño del arreglo inicial.
- Complejidad de codificación: Simple.

Ordenamiento por inserción (Insertion Sort)

- ***Idea básica del algoritmo:***
  – Mantener una “región ordenada” al inicio del arreglo. Posteriormente se hace “crecer” la región ordenada insertando repetidamente el primer elemento de la región no ordenada en el arreglo.

  – Este algoritmo se ejecutará con mayor desempeño en un arreglo parcialmente ordenado, debido a que no realiza ningún intercambio si los elementos se encuentran parcialmente ordenados.

Ordenamiento por inserción (Insertion Sort)

- ***Pseudocódigo***
  - InsertionSort(A)

```
for j <- 1 to length[A]-1
    key <- A[j]          // insert A[j] into the sorted sequence A[1..j-1]
    i <- j – 1
    while (i>=0 and A[i]>key)
        A[i+1] <- A[i]
        i <- i – 1
   A[i+1] <- key
```

- [10041 – Vito's Family]()

Ordenamiento por mezcla (Merge Sort)

- **Velocidad:** O(*n log n*) rápido, es más eficiente que los algoritmos anteriores para entradas grandes.
- **Espacio requerido:** El doble del tamaño del arreglo inicial.
- **Complejidad de codificación:** Media.

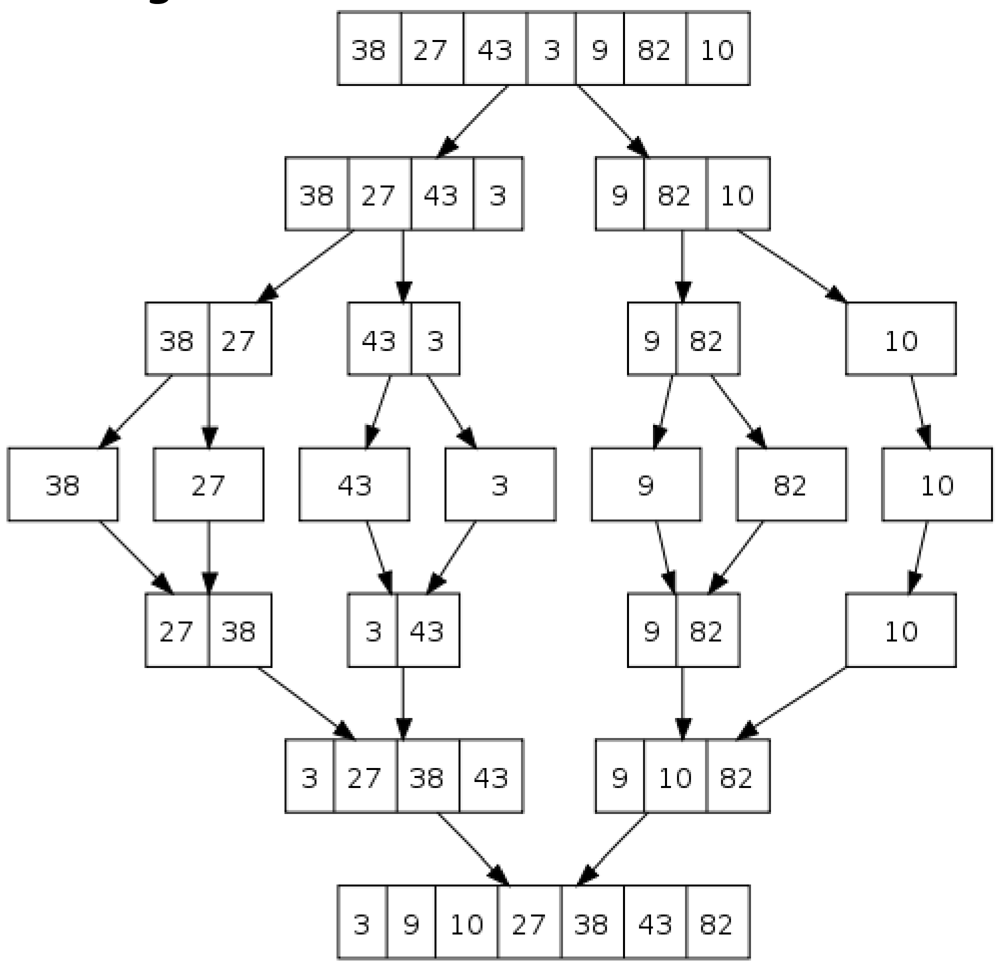
Ordenamiento por mezcla (Merge Sort)

- ***Idea básica del algoritmo:***
  - Utiliza la estrategia divide y vencerás.

    - **Divide:** Divide la secuencia de n elementos en dos subsecuencias de n/2 elementos cada una.
    - **Conquista:** Ordena las 2 subsecuencias recursivamente utilizando el algoritmo de mezcla (Merge Sort).
    - **Combina:** Mezcla las 2 subsecuencias ordenadas para producir la respuesta ordenada.

Ordenamiento por mezcla (Merge Sort)

- ***Idea básica del algoritmo:***



Ordenamiento por mezcla (Merge Sort)

- ***Pseudocódigo***
  - MergeSort(A,p,r)

```
if p < r
    q <- ⌊(p + r)/2⌋
    MergeSort(A, p, q)
    MergeSort(A, q + 1, r)
    Merge (A, p, q, r)
```

Ordenamiento por mezcla (Merge Sort)

- ***Pseudocódigo***
  - Merge(A, p, q, r)

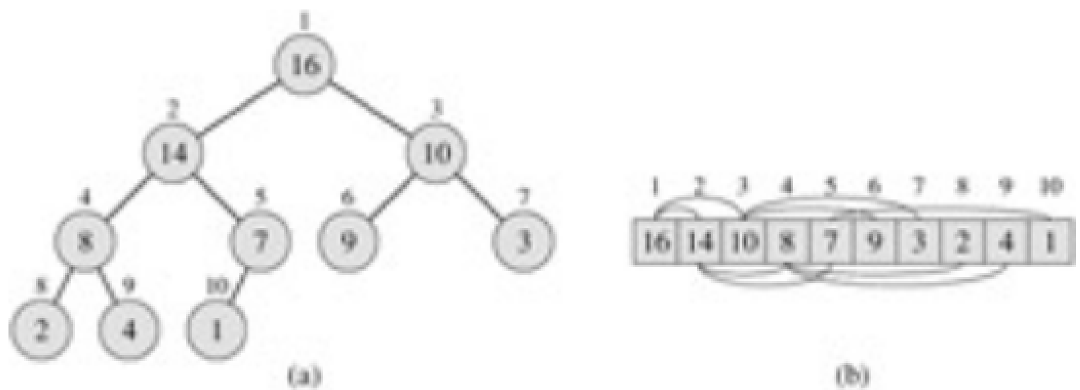
```
n1 ← q - p + 1
n2 ← r - q
create arrays L[n1 + 1] and R[n2 + 1]
for i ← 0 to n1-1
    do L[i] ← A[p + i ]
for j ← 0 to n2-1
    do R[j] ← A[q + j + 1]
L[n1] ← ∞
R[n2] ← ∞
i ← 0
j ← 0
for k ← p to r
    do if L[i] ≤ R[j]
        then A[k] ← L[i]
            i ← i + 1
        else A[k] ← R[j]
            j ← j + 1
```

Ordenamiento por Montículo (Heap Sort)

- **Velocidad:** O(*n lg n*) rápido, al igual que Merge Sort.
- **Espacio requerido:** El tamaño del arreglo inicial.
- **Complejidad de codificación:** Media.

Ordenamiento por Montículo (Heap Sort)

- ***Idea básica del algoritmo:***
  - Utiliza la estructura de datos Heap (Montículo).
  - Un montículo es un arreglo que puede ser visto como árbol binario casi completo.



Ordenamiento por Montículo (Heap Sort)

- ***Idea básica del algoritmo:***
  - Un arreglo A que representa un montículo es un objeto con 2 atributos:
    - length[A], que indica el número de elementos en el arreglo.
    - heap-size[A], es el número de elementos en el montículo almacenados dentro del arreglo.

Ordenamiento por Montículo (Heap Sort)

- ***Idea básica del algoritmo:***
  - La raíz del árbol es A[1], y dado el índice i de un nodo, podemos calcular el índice de su padre PARENT(i), hijo izquierdo LEFT(i), e hijo derecho RIGHT(i) de la siguiente manera:

    - ***PARENT(i)***

      return $\lfloor i/2 \rfloor$
    - ***LEFT(i)***

      return 2i
    - ***RIGHT(i)***

      return 2i + 1

Ordenamiento por Montículo (Heap Sort)

- ***Idea básica del algoritmo:***
  - Existen 2 tipo de montículos binarios: max-heaps y min-heaps.
    - En un max-heap, para cada nodo i diferente de la raíz se debe cumplir la siguiente propiedad:
      - A[PARENT(i)] ≥ A[i]
      - De esta manera el elemento más grande en un max-heap se encuentra en la raíz, y los subárboles de un nodo determinado contienen valores no mayores al valor del nodo.
    - En un min-heap, para cada nodo i diferente de la raíz se debe cumplir la siguiente propiedad:
      - A[PARENT(i)] ≤ A[i]
      - El elemento más pequeño en un min-heap se encuentra en la raíz.

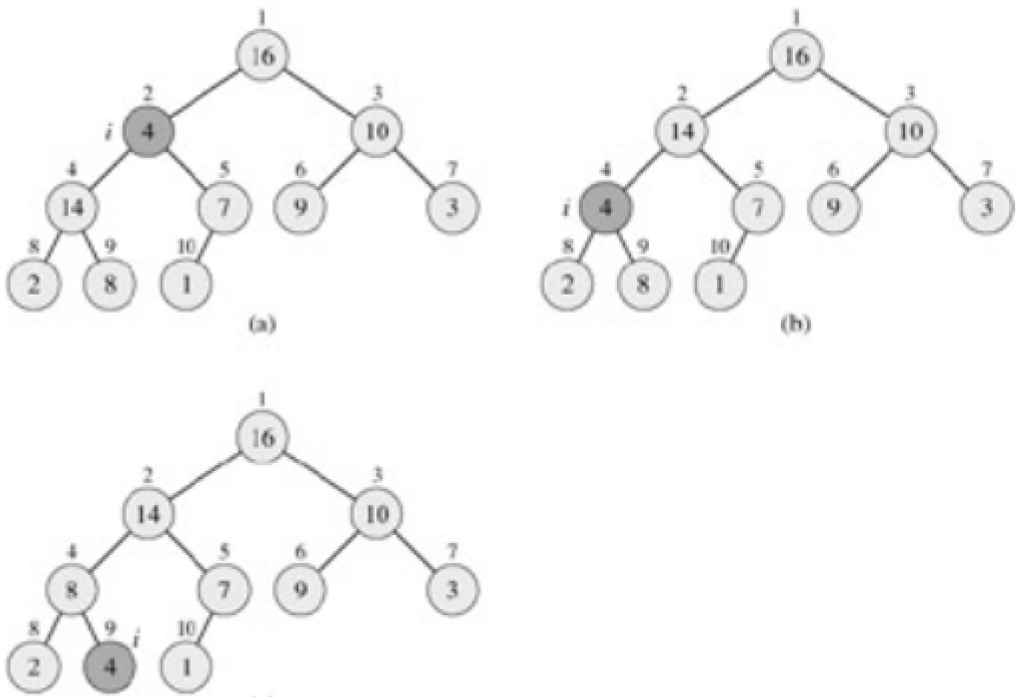
Ordenamiento por Montículo (Heap Sort)

- ***Idea básica del algoritmo:***
  - MAX-HEAPIFY es una subrutina para manipular max-heaps. Sus entradas son un arreglo A y un índice i dentro del arreglo. Cuando se invoca MAX-HEAPIFY, se asume que los árboles binarios con raíz LEFT(i) y RIGHT(i) son max-heaps, pero que A[i] puede ser más pequeño que sus hijos, violando la propiedad de max-heap.
  - La funcion MAX-HEAPIFY permite al valor A[i] "descender" por el max-heap, de tal manera que el subábol con raíz en el elemento con índice i se convierta también en un max-heap.
  - MAX-HEAPIFY(A, i)

```
l ← LEFT(i)
r ← RIGHT(i)
if l ≤ heap-size[A] and A[l] > A[i]
  then largest ← l
  else largest ← i
if r ≤ heap-size[A] and A[r] > A[largest]
  then largest ← r
if largest ≠ i
  then exchange A[i] ↔ A[largest]
     MAX-HEAPIFY(A, largest)
```

Ordenamiento por Montículo (Heap Sort)

- La acción de MAX-HEAPIFY(A,2) donde heap-size[A]=10



(a) (b) (c)

Ordenamiento por Montículo (Heap Sort)

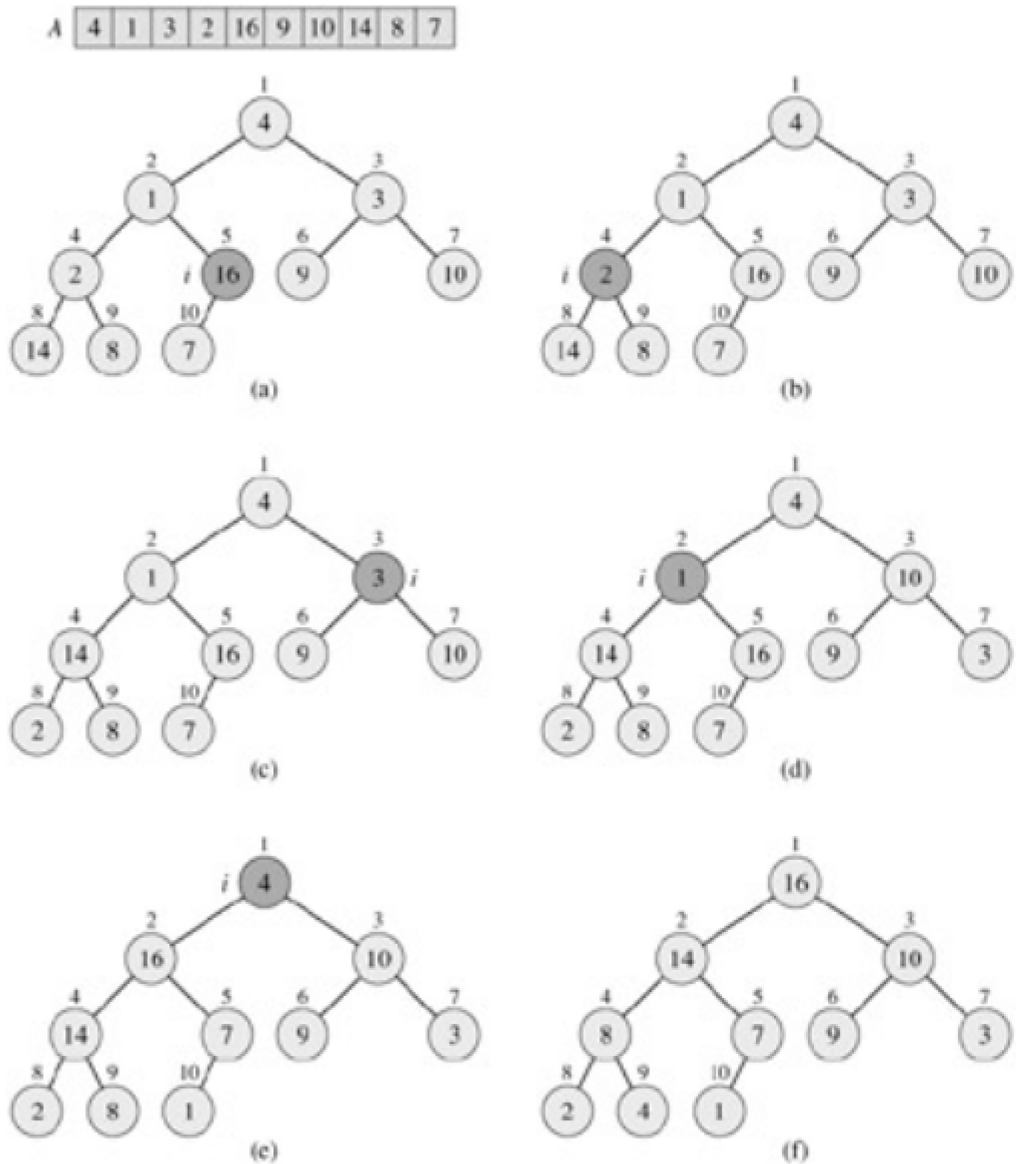
- Se puede utiliza MAX-HEAPIFY en forma ascendente para convertir un arreglo A[1 …n] en un max-heap.

  – BUILD-MAX-HEAP(A)

```
heap-size[A] ← length[A]
for i ← ⌊length[A]/2⌋ downto 1
    do MAX-HEAPIFY(A, i)
```

Ordenamiento por Montículo

- BUILD-HEAP

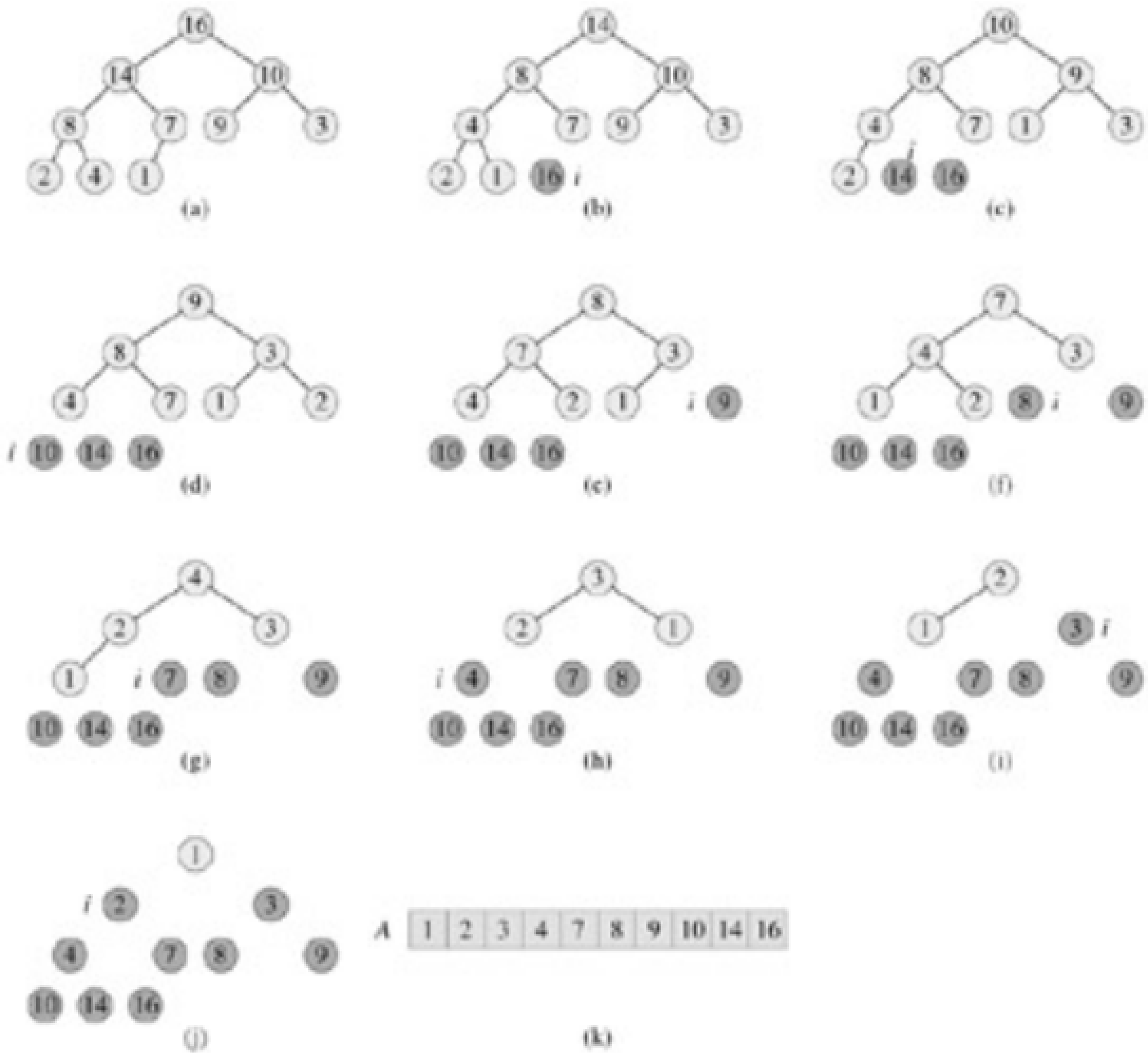


Ordenamiento por Montículo (Heap Sort)

- ***Pseudocódigo:***
  - HEAPSORT(A)

```
BUILD-MAX-HEAP(A)
for i ← length[A] downto 2
    do exchange A[1] ↔ A[i]
        heap-size[A] ← heap-size[A] - 1
        MAX-HEAPIFY(A, 1)
```

Ordenamiento por Montículo (Heap Sort)



Ordenamiento rápido (Quick Sort)

- **Velocidad:** O(*n lg n*) es rápido en el caso promedio, sin embargo en el peor caso su complejidad es O(n^2). Este algoritmo es el más utilizado en la práctica.

- **Espacio requerido:** El tamaño del arreglo inicial.

- **Complejidad de codificación:** Media.

Ordenamiento rápido (Quick Sort)

- ***Idea básica del algoritmo:***
  - Utiliza la estrategia divide y vencerás.
    - **Divide:** Particiona (organiza) el arreglo A[p…r] en 2 subarreglos (posiblemente vacios) A[p…q-1] y A[q+1…r], de tal manera que cada elemento de A[p…q-1] es menor o igual que A[q], el cual a su vez, es menor o igual a cada elemento de A[q+1…r]. El cálculo del índice q se realiza en este procedimiento de partición.
    - **Conquista:** Ordena los 2 subarreglos A[p…q - 1] y A[q + 1…r] recursivamente utilizando el algoritmo QuickSort.
    - **Combina:** Dado que los 2 subarreglos se ordenan directamente, no se requiere combinarlos: el arreglo completo A[p…r] ya esta ordenado.

Ordenamiento rápido (Quick Sort)

- ***Pseudocódigo:***
  - QUICKSORT(A, p, r)

```
if p < r
  then q ← PARTITION(A, p, r)
       QUICKSORT(A, p, q - 1)
       QUICKSORT(A, q + 1, r)
```

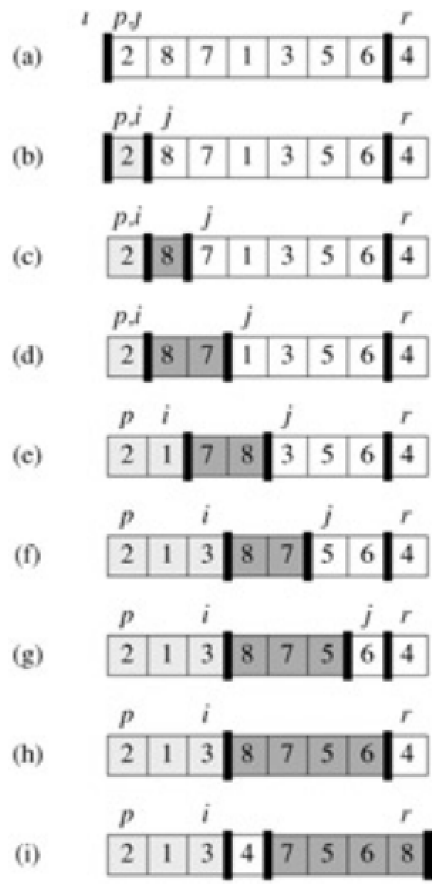
Ordenamiento rápido (Quick Sort)

- ***Pseudocódigo:***
  - PARTITION(A, p, r)

```
x ← A[r]
i ← p - 1
for j ← p to r - 1
    do if A[j] ≤ x
        then i ← i + 1
            exchange A[i] ↔ A[j]
exchange A[i + 1] ↔ A[r]
return i + 1
```

Ordenamiento rápido (Quick Sort)

El comportamiento de PARTITION:



Ordenamiento rápido (Quick Sort)

- En el algoritmo anterior el peor caso ocurre cuando el procedimiento de partición produce un subproblema con n-1 elementos y el otro con 0 elementos. Asumiendo que esta partición desbalanceada se presenta en cada llamada recursiva, entonces el tiempo de ejecución es O(n^2).

- En el mejor caso posible, el procedimiento de partición PARTITION produce 2 subproblemas, cada uno de tamaño no mayor a n/2, debido a que uno es de tamaño $\lfloor n/2 \rfloor$ y el otro de tamaño $\lceil n/2 \rceil$- 1. En este caso, quicksort tiene un tiempo de ejecución de O(*n lg n*) .

Ordenamiento rápido (Quick Sort)

- ***Pseudocódigo:***
  - RANDOMIZED-PARTITION(A, p, r)

```
i ← RANDOM(p, r)
exchange A[r] ↔ A[i]
return PARTITION(A, p, r)
```

Ordenamiento rápido (Quick Sort)

- ***Pseudocódigo:***
  - RANDOMIZED-QUICKSORT(A, p, r)

```
if p < r
   then q ← RANDOMIZED-PARTITION(A, p, r)
        RANDOMIZED-QUICKSORT(A, p, q - 1)
        RANDOMIZED-QUICKSORT(A, q + 1, r)
```

Ordenamiento por conteo (Counting Sort)

- **Velocidad:** O(*n), es más rápido que los algoritmos anteriores basados en comparaciones, su tiempo de ejecución es lineal.*

- **Espacio requerido:** Se requiere el arreglo inicial A[1…n] (contiene n elementos enteros en el rango 0 a k). Así como otros 2 arreglos: el arreglo B[1..n] para almacenar la salida ordenada, y otro arreglo C[0…k] que proporciona almacenamiento de trabajo temporal

- **Complejidad de codificación:** Media.

Ordenamiento por conteo (Counting Sort)

- ***Idea básica del algoritmo:***
  - La idea básica del counting sort es determinar, para cada elemento de entrada x el número de elementos menores que x. Esta información se utiliza para colocar el elemento x directamente en su posición final en el arreglo ordenado. Por ejemplo, si hay 17 elementos menores que x, entonces x se debe colocar en la posición 18 dentro del arreglo ordenado.
  - Este procedimiento debe manejar la situación en la cual muchos elementos tienen el mismo valor, debido a que estos no se van a colocar en la misma posición.

Ordenamiento por conteo (Counting Sort)

- **Pseudocódigo:**

```
COUNTING-SORT(A, B, k)
      for i ← 0 to k
         do C[i] ← 0
      for j ← 1 to length[A]
         do C[A[j]] ← C[A[j]] + 1
      ▹ C[i] now contains the number of elements equal to i.
      for i ← 1 to k
         do C[i] ← C[i] + C[i - 1]
      ▹ C[i] now contains the number of elements less than or equal to i.
      for j ← length[A] downto 1
        do B[C[A[j]]] ← A[j]
           C[A[j]] ← C[A[j]] - 1
```

Ordenamiento por conteo (Counting Sort)

- Comportamiento del algoritmo: