Universidad Nacional Mayor de San Marcos
Universidad del Perú. Decana de América.

Análisis y diseño de algoritmos

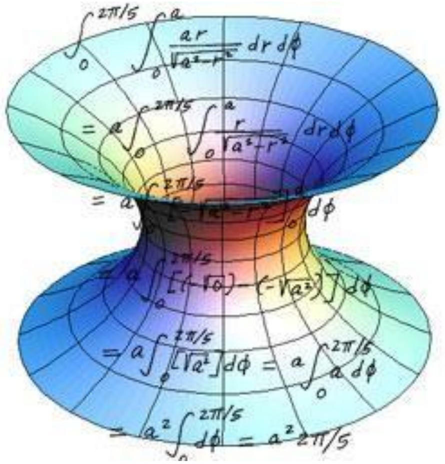
Sesión 04

Rúbrica del proyecto

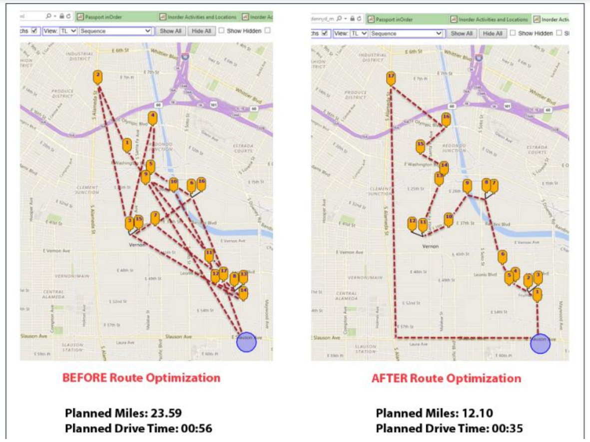
Criterios	Logrado	En proceso	En inicio
Utilización de diferentes tipos de algoritmos	Al menos utiliza 3 tipos de algoritmos desarrollados en el curso. **(5 puntos)**	Sólo utiliza 2 tipos de algoritmos desarrollados en clase. **(3.5 puntos)**	Sólo utiliza un tipo de algoritmo desarrollado en clase. **(2 puntos)**
Utilización de diferentes estructuras de datos	Al menos utiliza 3 tipos de estructura de datos desarrollados en el curso. **(5 puntos)**	Sólo utiliza 2 tipos de estructuras de datos desarrollados en clase. **(3.5 puntos)**	Sólo utiliza un tipo de estructura de datos desarrollado en clase.. **(2 puntos)**
Se presenta un análisis de complejidad del código de software	Describe cuál es el nivel de complejidad temporal y espacial. **(4 puntos)**	Describe sólo un tipo de complejidad. **(3 puntos)**	No describe ningún análisis de complejidad del código de software. **(0.5 puntos)**
Comportamiento de la aplicación ante diferentes valores de entrada	El programa de software no incrementa su complejidad en forma notable cuando la entrada es muy grande. **(3 puntos)**	El programa de software incrementa su complejidad si la entrada crece a valores enormes y que pueden ocurrir en la realidad. **(1.5 puntos)**	Si la entrada crece a valores grandes y reales la aplicación no funciona correctamente. **(0.5 punto)**
Utilidad de la Aplicación	El trabajo presenta la solución a un problema existente en la realidad y los algoritmos y estructura de datos se pueden utilizar en grandes sistemas. Se puede utilizar por el usuario común **(3 puntos)**	El trabajo no presenta la solución a un problema existente en la realidad, los algoritmos y estructura de datos se pueden utilizar en grandes sistemas El trabajo se puede utilizar por un usuario común **(2 puntos)**	El trabajo no presenta la solución a un problema existente en la realidad, los algoritmos y estructura de datos no se pueden utilizar en grandes sistemas. El trabajo se puede utilizar por un usuario común. **(1 punto)**

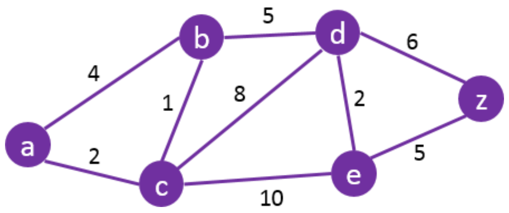
Ideas para el proyecto

Calculadora científica



WorkForce Management





Dijkstra's Algorithm

Sistema antiplagio

Logro de la sesión

Al finalizar la sesión, el estudiante realiza un análisis y desarrollo de algoritmos de ordenamiento utilizando un lenguaje de programación

Agenda

- Ordenamiento y Búsqueda
- Algoritmo de Ordenación Burbuja
- Algoritmo de ordenación por selección
- Algoritmo de ordenación por inserción
- Algoritmo de ordenación Quicksort
- Algoritmo de ordenación Mergesort
- Algoritmo de ordenación Heapsort
- Otros Algoritmos de ordenamiento

Ordenamiento y Búsqueda

- Son problemas fundamentales en las ciencias de la computación y programación.
- El ordenamiento es clave para facilitar la búsqueda
- Múltiples algoritmos para resolver el mismo problema: ¿Cómo sabemos qué algoritmo es "mejor"?
- Los ejemplos usarán arreglos de números enteros para ilustrar los algoritmos.

Ordenamiento

U.S. All-time List - Marathon

As of 4/24/08

Women

1	1	2:19:36	Deena Kastor nee Drossin
2		2:21:16	Drossin (2)
3	2	2:21:21	Joan Benoit Samuelson
4		2:21:25	Kastor (3)
5		2:22:43a	Benoit (2)
6		2:24:52a	Benoit (3)
7		2:26:11	Benoit (4)
8	3	2:26:26a	Julie Brown
9	4	2:26:40a	Kim Jones

Song Name	Time	Track # ▲	Artist	Album
Letters from the Wasteland	4:29	1 of 10	The Wallflowers	Breach
When You're On Top	3:54	1 of 13	The Wallflowers	Red Letter Days
Hand Me Down	3:35	2 of 10	The Wallflowers	Breach
How Good It Can Get	4:11	2 of 13	The Wallflowers	Red Letter Days
Sleepwalker	3:31	3 of 10	The Wallflowers	Breach
Closer To You	3:17	3 of 13	The Wallflowers	Red Letter Days
I've Been Delivered	5:01	4 of 10	The Wallflowers	Breach
Everybody Out Of The Water	3:42	4 of 13	The Wallflowers	Red Letter Days
Witness	3:34	5 of 10	The Wallflowers	Breach
Three Ways	4:19	5 of 13	The Wallflowers	Red Letter Days
Some Flowers Bloom Dead	4:43	6 of 10	The Wallflowers	Breach
Too Late to Quit	3:54	6 of 13	The Wallflowers	Red Letter Days
Mourning Train	4:04	7 of 10	The Wallflowers	Breach
If You Never Got Sick	3:44	7 of 13	The Wallflowers	Red Letter Days
Up from Under	3:38	8 of 10	The Wallflowers	Breach
Health and Happiness	4:03	8 of 13	The Wallflowers	Red Letter Days
Murder 101	2:31	9 of 10	The Wallflowers	Breach
See You When I Get There	3:09	9 of 13	The Wallflowers	Red Letter Days
Birdcage	7:42	10 of 10	The Wallflowers	Breach
Feels Like Summer Again	3:48	10 of 13	The Wallflowers	Red Letter Days
Everything I Need	3:37	11 of 13	The Wallflowers	Red Letter Days
Here in Pleasantville	3:40	12 of 13	The Wallflowers	Red Letter Days
Empire in My Mind (Bonus Track)	3:31	13 of 13	The Wallflowers	Red Letter Days

Métodos de Ordenamiento

La ordenación

También llamada **clasificación de datos**, consiste en la disposición de los mismos de acuerdo con alguna característica, la cual puede ser de dos formas distintas:

- **Ascendente** (menor a mayor)
- **Descendente** (mayor a menor)

Los métodos de ordenación se clasifican en dos categorías:

- **Ordenación interna** (de arreglos)
- **Ordenación externa** (de archivos)

La **ordenación interna**, recibe este nombre ya que los elementos del arreglo se encuentran en la memoria principal de la computadora.

Existen numerosos algoritmos de ordenamiento interno y podemos clasificarlos en directos, indirectos, y otros.

Entre los algoritmos de ordenamiento directos tenemos:
Ordenamiento por intercambio/ Ordenamiento por burbuja
Ordenamiento por selección
Ordenamiento por inserción

Entre los algoritmos de ordenamiento indirectos tenemos:
Ordenamiento por mezcla (Mergesort)
Ordenamiento rápido (Quicksort)
Ordenamiento por Montículos (Heap sort)

También existen otros algoritmos de ordenamiento, como por ejemplo:
Ordenamiento por Incrementos (Shell sort).
Ordenamiento por Cubetas (Bin sort).
Ordenamiento por Residuos (Radix).

Ordenamiento y Búsqueda

- Una de las dificultades con el ordenamiento es trabajar con un contenedor de almacenamiento de tamaño fijo (arreglo), si se cambia el tamaño, es costoso (lento).
- Los algoritmos de ordenamiento simple se ejecutan en un tiempo cuadrático $O(n^2)$ -Big O
  - Ordenamiento por burbuja
  - Ordenamiento por selección
  - Ordenamiento por inserción

Ordenamiento estable

- La estabilidad es una propiedad del ordenamiento.
- Si un algoritmo de ordenamiento garantiza que el orden relativo de los elementos iguales permanece igual, entonces es un algoritmo estable
    - $[7_1, 6, 7_2, 5, 1, 2, 7_3, -5]$
        - subíndices agregados para mayor claridad
    - $[-5, 1, 2, 5, 6, 7_1, 7_2, 7_3]$
        - Resultado de un algoritmo de ordenamiento estable

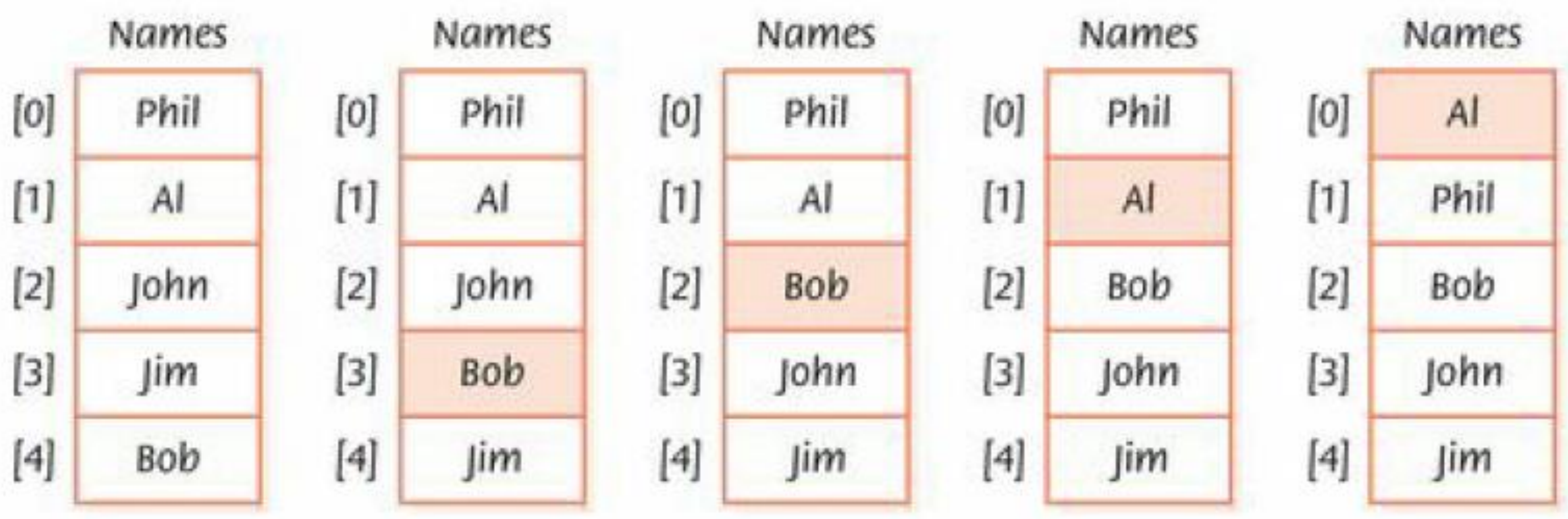
- Ejemplo del mundo real:
    - ordenar una tabla en Wikipedia por un criterio, luego otro
    - ordenar por país, luego por principales victorias

Ordenamiento divertido

¿Porqué no utilizar el ordenamiento por burbuja?

Algoritmo de ordenación por burbuja

Un algoritmo de ordenación bien conocido es el de ordenación por burbuja, el cual está basada en el proceso de comparar dos nombres adyacentes repetidamente e intercambiarlos si no se encuentran en el orden correcto. Supongamos que la lista en cuestión tiene n entradas. La ordenación por burbuja comenzaría comparando (y es posible que intercambiando) las entradas en las posiciones n y n - 1. A continuación, consideraría las entradas en las posiciones n - 1 y n – 2, y continuará avanzando por la lista hasta que la primera y segunda entrada de la lista hubieran sido comparadas (y es posible que intercambiadas). Observe que cada pasada a través de la lista hará que la entrada más pequeña quede en la posición inicial. De forma similar, otra de esas pasadas hará que la segunda entrada más pequeña quede en la segunda posición de la lista. Por lo tanto, haciendo un total de n - 1 pasadas a través de la lista, conseguiremos una lista completamente ordenada.



(a) First iteration (sorted elements are shaded)

Algoritmo de ordenación por burbuja

OTRA EXPLICACIÓN

Se basa en el principio de comparar pares de elementos adyacentes e intercambiarlos entre sí hasta que estén todos ordenados

Ejemplo de ordenar un arreglo de 1 dimensión de menos a más:

A[0]	A[1]	A[2]	A[3]	A[4]
6	5	3	1	0

Los pasos a dar son:

1. Comparar A[0] y A[1], si están en orden se mantienen como están, en caso contrario se intercambia entre sí.
2. A continuación se comparan A[1] y A[2]; de nuevo se intercambian si es necesario.
3. El proceso continua hasta que cada elemento del arreglo ha sido comprada con sus elementos adyacentes y se han realizado los intercambios necesarios.

Algoritmo de ordenación por burbuja

		A[0]	A[1]	A[2]	A[3]	A[4]
Primera Vuelta	1° Comparación	6	5	3	1	0
	2° Comparación	5	6	3	1	0
	3° Comparación	5	3	6	1	0
	4° Comparación	5	3	1	6	0
	Resultado	5	3	1	0	6
Segunda Vuelta	5° Comparación	5	3	1	0	6
	6° Comparación	3	5	1	0	6
	7° Comparación	3	1	5	0	6
	8° Comparación	3	1	0	5	6
	Resultado	3	1	0	5	6
Tercera Vuelta	9° Comparación	3	1	0	5	6
	10° Comparación	1	3	0	5	6
	11° Comparación	1	0	3	5	6
	12° Comparación	1	0	3	5	6
	Resultado	1	0	3	5	6
Cuarta Vuelta	13° Comparación	1	0	3	5	6
	14° Comparación	0	1	3	5	6
	15° Comparación	0	1	3	5	6
	16° Comparación	0	1	3	5	6
	Resultado	0	1	3	5	6

- Dado una lista de n elementos, si se hace n-1 veces la operación de comparación se tiene el numero mayor en la última posición. Para tener la lista completamente ordenada como máximo se tiene que repetir n-1 veces la operación. La ordenación requerirá a lo más (n-1)*(n-1) intercambios.
- En el ejemplo para una lista de 5 elementos se hace (5-1)*(5-1) comparaciones como máximo

Algoritmo de ordenación por burbuja

		A[0]	A[1]	A[2]	A[3]	A[4]
Primera Vuelta	1° Comparación	6	5	3	1	0
	2° Comparación	5	6	3	1	0
	3° Comparación	5	3	6	1	0
	4° Comparación	5	3	1	6	0
	Resultado	5	3	1	0	6
Segunda Vuelta	5° Comparación	5	3	1	0	6
	6° Comparación	3	5	1	0	6
	7° Comparación	3	1	5	0	6
	8° Comparación	3	1	0	5	6
	Resultado	3	1	0	5	6
Tercera Vuelta	9° Comparación	3	1	0	5	6
	10° Comparación	1	3	0	5	6
	11° Comparación	1	0	3	5	6
	12° Comparación	1	0	3	5	6
	Resultado	1	0	3	5	6
Cuarta Vuelta	13° Comparación	1	0	3	5	6
	14° Comparación	0	1	3	5	6
	15° Comparación	0	1	3	5	6
	16° Comparación	0	1	3	5	6
	Resultado	0	1	3	5	6

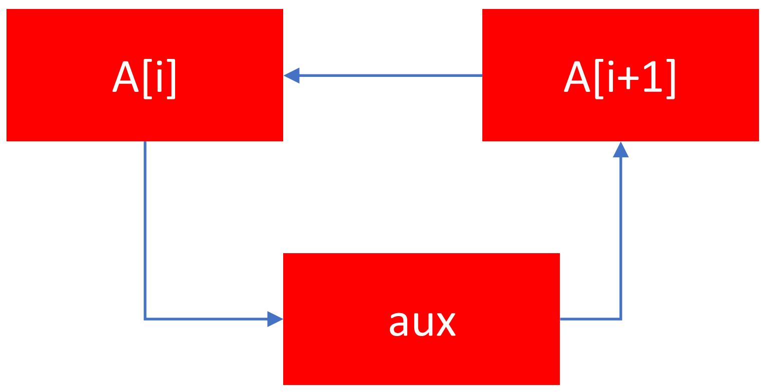
La acción intercambiar entre sí los valores de dos elementos A[i] y A[i+1], es una acción compuesta que contiene las siguientes acciones:

Si A[i]>A[i+1] entonces

aux=A[i];

A[i]=A[i+1];

A[i+1]=aux;



Algoritmo de ordenación por burbuja – Código Java

```
import java.util.Scanner;
public class burbuja {
    public static void main(String[] args) {
        int n=0,aux=0;
        Scanner entrada= new Scanner(System.in);
        System.out.println("Ingrese el tamaño de la lista de números");
        n=entrada.nextInt();
        int[] A=new int[n];
        for(int i=0;i<n;i++){
            System.out.println("Ingrese el número " + (i+1)+ " de la lista");
            A[i]=entrada.nextInt();
        }
        System.out.println("Lista desordenada:");
        for(int i=0;i<n;i++){
            System.out.print(A[i]+" ");
        }
        for(int i=0;i<n-1;i++){
            for(int j=0;j<n-1;j++){
                if(A[j]>A[j+1]){
                    aux=A[j];
                    A[j]=A[j+1];
                    A[j+1]=aux;
                }
            }
        }
        System.out.println();
        System.out.println("Lista ordenada:");
        for(int i=0;i<n;i++){
            System.out.print(A[i]+" ");
        }
    }
}
```

Para ordenar la lista completa se deben realizar las sustituciones correspondientes (n-1)*(n-1). Así en el caso de una lista de 100 elementos se deben realizar casi 10,000 iteraciones.

Algoritmo de ordenación por burbuja – Código Java mejorado

		A[0]	A[1]	A[2]	A[3]	A[4]
Primera Vuelta	1° Comparación	6	5	3	1	0
	2° Comparación	5	6	3	1	0
	3° Comparación	5	3	6	1	0
	4° Comparación	5	3	1	6	0
	Resultado	5	3	1	0	6
Segunda Vuelta	5° Comparación	5	3	1	0	6
	6° Comparación	3	5	1	0	6
	7° Comparación	3	1	5	0	6
	8° Comparación	3	1	0	5	6
	Resultado	3	1	0	5	6
Tercera Vuelta	9° Comparación	3	1	0	5	6
	10° Comparación	1	3	0	5	6
	11° Comparación	1	0	3	5	6
	12° Comparación	1	0	3	5	6
	Resultado	1	0	3	5	6
Cuarta Vuelta	13° Comparación	1	0	3	5	6
	14° Comparación	0	1	3	5	6
	15° Comparación	0	1	3	5	6
	16° Comparación	0	1	3	5	6
	Resultado	0	1	3	5	6

Se puede realizar una mejora en la velocidad de ejecución del algoritmo. Obsérvese en el primer recorrido de la lista cuando i=0, el mayor valor se mueve al último elemento A[n-1] o A[4]. Por consiguiente en el siguiente paso o vuelta ya no es necesario comparar A[n-2] y A[n-1] o A[3] y A[4]. En otras palabras el límite superior del bucle for puede ser n-2.

Algoritmo de ordenación por burbuja – Código Java mejorado

```java
import java.util.Scanner;
public class burbuja {
    public static void main(String[] args) {
        int n=0,aux=0;
        Scanner entrada= new Scanner(System.in);
        System.out.println("Ingrese el tamaño de la lista de números");
        n=entrada.nextInt();
        int[] A=new int[n];
        for(int i=0;i<n;i++){
            System.out.println("Ingrese el número " + (i+1)+ " de la lista");
            A[i]=entrada.nextInt();
        }
        System.out.println("Lista desordenada:");
        for(int i=0;i<n;i++){
            System.out.print(A[i]+" ");
        }
        for(int i=0;i<n-1;i++){
            for(int j=0;j<n-1-i;j++){
                if(A[j]>A[j+1]){
                    aux=A[j];
                    A[j]=A[j+1];
                    A[j+1]=aux;
                }
            }
        }
        System.out.println();
        System.out.println("Lista ordenada:");
        for(int i=0;i<n;i++){
            System.out.print(A[i]+" ");
        }
    }
}
```

Algoritmo de ordenación por burbuja – Análisis de Complejidad

El algoritmo consta de dos bucles anidados, está dominado por los dos bucles, de ahí que el análisis del algoritmo en relación a la complejidad sea inmediato; siendo n el número de elementos, el primer bucle hace n-1 pasadas y el segundo n-i-1 comparaciones en cada pasada (i es el índice del bucle externo, i = 0 .. n-2). El número total de comparaciones se obtiene desarrollando la sucesión matemática formada para los distintos valores de i:

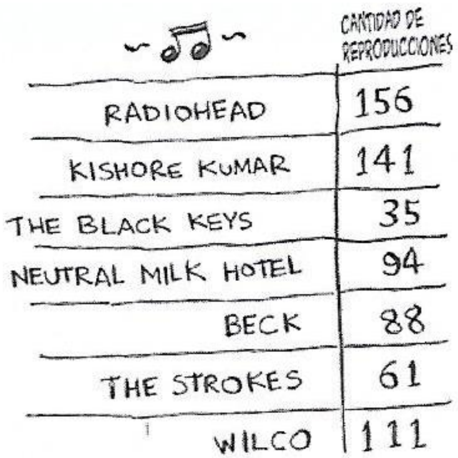
n-1, n-2, n-3, ... , 1

Entonces sería la suma (n-1) + (n-2) + (n-3) + ... + 3 + 2 + 1 = (n-1)*(n)/2

La ordenación completa requiere tal número de comparaciones y un número similar de intercambios en el peor de los casos. Entonces, el número de comparaciones es (n-1) * (n)/2 = $(n^2 - n)/2$ y en el peor de los casos, los mismos intercambios. El término dominante es n^2, por tanto, la complejidad es $O(n^2)$.

Rendimiento en el peor caso	$O(n^2)$
Rendimiento en el caso promedio	$O(n^2)$
Ordenamiento estable?	Si
Complejidad de espacio	O(1), se necesita sólo una variable temporal

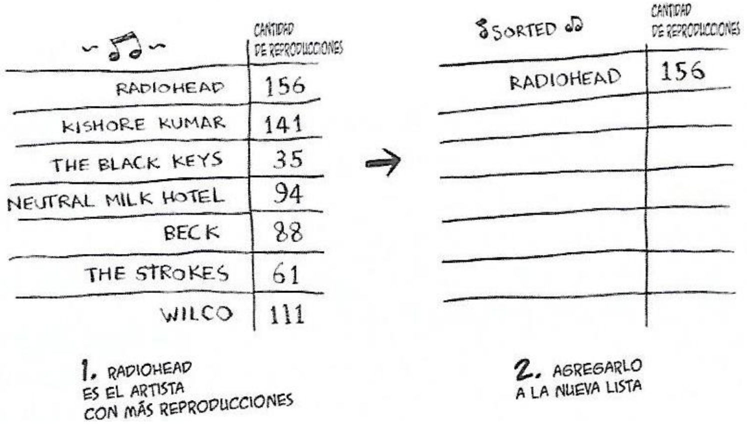
Ordenación por selección



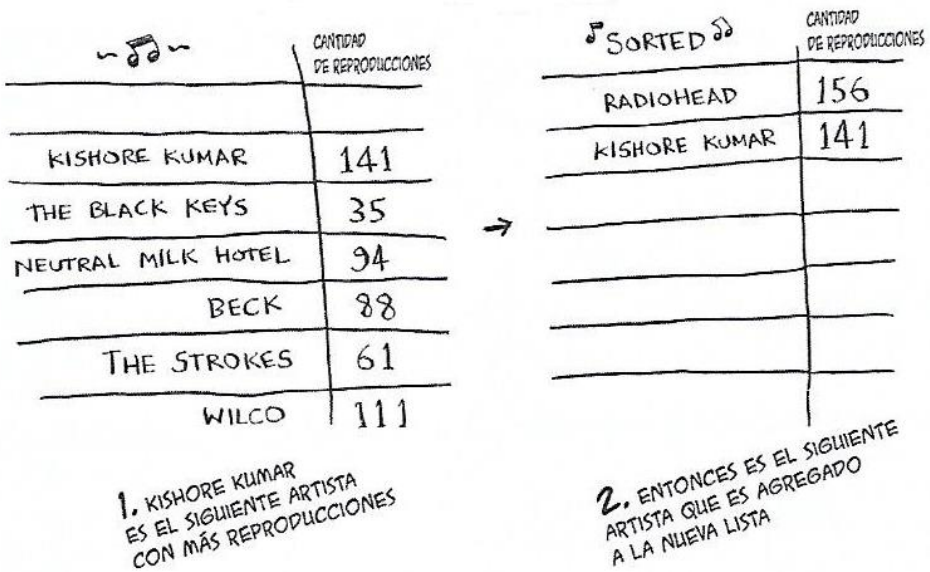
Asume que tienes una biblioteca de música en tu PC, para cada artista conoces la cantidad de veces que lo has escuchado

Quieres ordenar esta lista de más escuchado a menos escuchado ¿Cómo lo harías?

Una forma posible es recorrer la lista y encontrar el artista con mayor cantidad de reproducciones. Luego añadir ese artista a una lista nueva



Hazlo nuevamente para encontrar al siguiente artista más escuchado

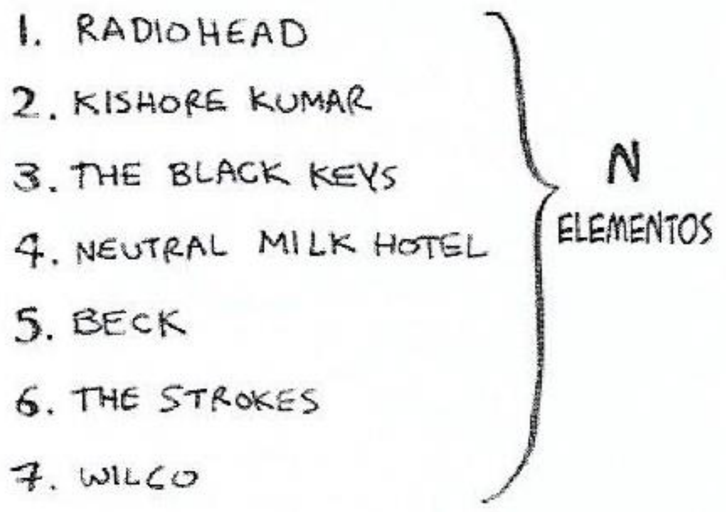


Continúa haciendo esto y terminarás con una lista ordenada

Ordenación por selección

¿Cuánto demoraría en ejecutarse este algoritmo?

Primero tienes que comprobar cada elemento de la lista para encontrar al artista con más reproducciones. Esto toma un tiempo de O(n), tocas el elemento al menos una vez.

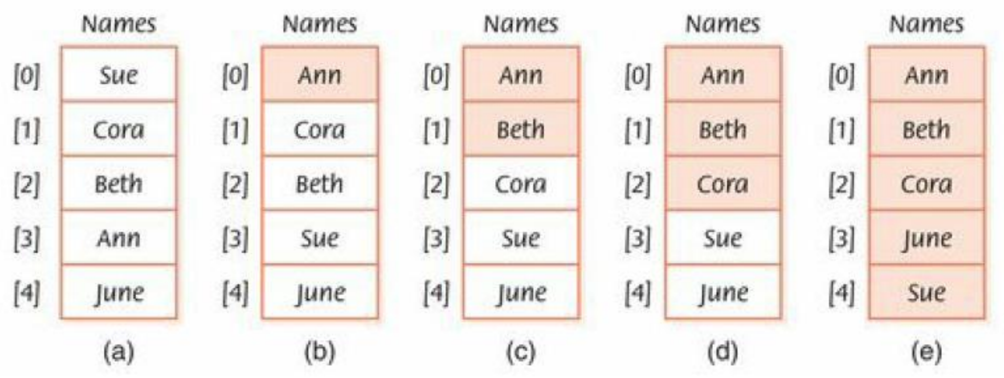


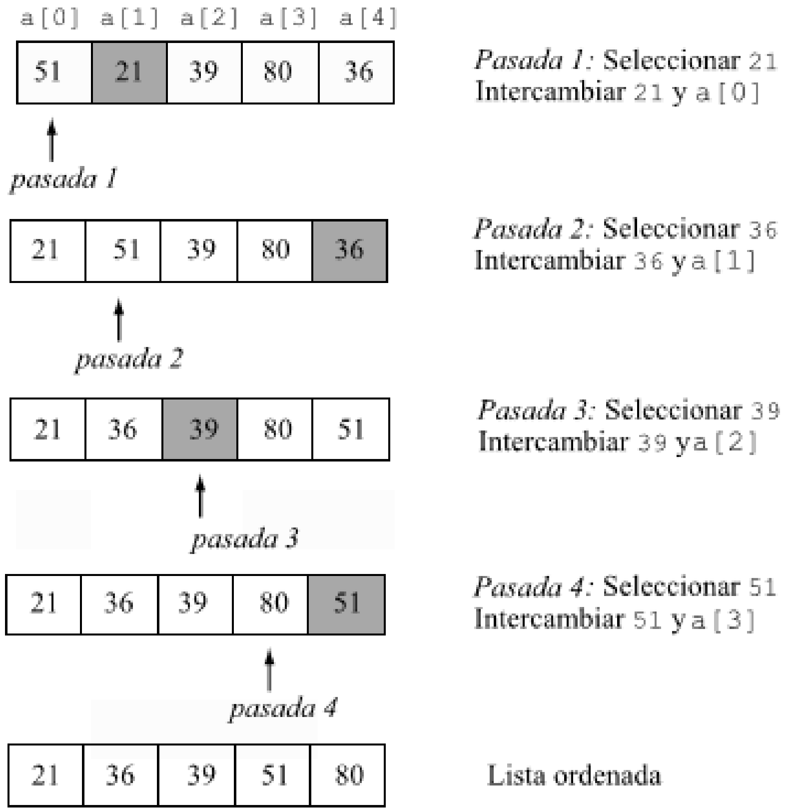
Esta operación la tienes que repetir n veces para terminar de elaborar la lista ordenada entonces todo toma un tiempo de O(nxn) o O(n^2).

Ordenación por selección

Escribir el algoritmo de ordenación por selección en Java

Comienza seleccionando la entrada más pequeña de la lista y moviéndola al principio de la misma. A continuación, selecciona la entrada más pequeña de las entradas restantes de la lista y la mueve a la segunda posición de la lista. Seleccionando repetidamente la entrada más pequeña de la parte restante de la lista y moviendo hacia arriba esa entrada, la versión ordenada de la lista va aumentando a partir de la posición inicial de la misma, mientras que la parte posterior de la lista, compuesta por las entradas restantes no ordenadas, va reduciéndose.





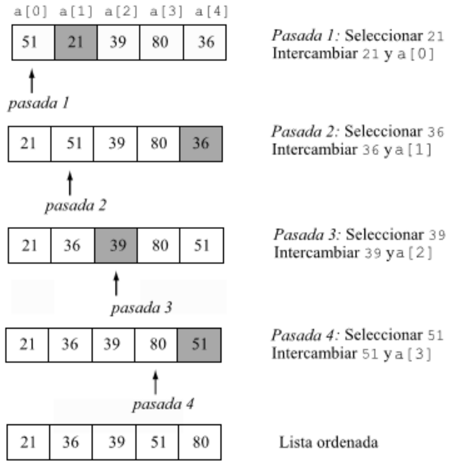
Los pasos sucesivos a dar son:

1. *Seleccionar el elemento menor del vector de n elementos.*
2. *Intercambiar dicho elemento con el primero.*
3. *Repetir estas operaciones con los n – 1 elementos restantes, seleccionando el segundo elemento; continuar con los n – 2 elementos restantes y así sucesivamente hasta que sólo quede el mayor.*

Ordenación por selección

Escribir el algoritmo de ordenación por selección en Java

El método ordSeleccion() ordena un array de números reales de n elementos, n coincide con el atributo length del array. En la pasada i el proceso de selección explora la sublista a[i]–a[n-1] y fija el índice del elemento más pequeño. Después de terminar la exploración de los elementos, a[i] y a [indiceMenor] se intercambian; operación que se realiza llamando al método intercambiar()




```java
public static void ordSeleccion(int a[]) {
    int indiceMenor, i, j, n;
    n = a.length;
    // ordenar a[0]..a[n-2] y a[n-1] en cada pasada
    for (i = 0; i < n - 1; i++) {
        // comienzo de la exploración en índice i
        indiceMenor = i;
        // j explora la sublista a[i+1]..a[n-1]
        for (j = i + 1; j < n; j++) {
            if (a[j] < a[indiceMenor]) {
                indiceMenor = j;
            }
        }
        // sitúa el elemento mas pequeño en a[i]
        if (i != indiceMenor) {
            intercambiar(a, i, indiceMenor);
        }
    }
}

public static void intercambiar(int[] a, int i, int j) {
    int aux = a[i];
    a[i] = a[j];
    a[j] = aux;
}
```

Ordenación por selección

Escribir el algoritmo de ordenación por selección en Java

```java
package Semana1;
public class AED_Programa2 {
    public static void main(String[] args) {
        double[] lista={32,3,5,76,9,101,161,1,21,204,27,8,4,131,6,7,39,14,43,45};
        ordSeleccion(lista);
        for (int i=0;i<lista.length;i++){
        System.out.print("["+lista[i]+"]  ");
    }
    }
    public static void ordSeleccion (double a[]){
        int indiceMenor,i,j,n;
        n=a.length;
        for (i=0;i<n-1;i++){
            //comienzo de la exploracion en indice i
            indiceMenor=i;
            //j explora la sublista a[i+1]..a[n-1]
            for (j=i+1;j<n;j++){
                if(a[j]<a[indiceMenor]){
                    indiceMenor=j;
                }
            }
            //situa el elemento más pequeño en a[i]
            if (i!=indiceMenor){
                intercambiar(a,i,indiceMenor);
            }
        }
    }
    public static void intercambiar(double[] a, int i, int j){
        double aux=a[i];
        a[i]=a[j];
        a[j]=aux;
    }
}
```

Algoritmo de ordenación por selección – Análisis de Complejidad

El algoritmo, requiere un número fijo de comparaciones que sólo dependen del tamaño del array y no de la distribución inicial de los datos. El término dominante del algoritmo es el bucle externo que anida a un bucle interno. Por ello, el número de comparaciones que realiza el algoritmo es el número decreciente de iteraciones del bucle interno: n-1, n-2, ... 2, 1 (n es el número de elementos).

n-1, n-2, n-3, ... , 1

Entonces sería la suma (n-1) + (n-2) + (n-3) + ... + 3 + 2 + 1 = (n)*(n-1)/2

Rendimiento en el peor caso	$O(n^2)$
Rendimiento en el caso promedio	$O(n^2)$
Ordenamiento estable?	No
Complejidad de espacio	O(1), se necesita sólo una variable temporal

Ejercicio

Ejercicio: Mostrar que el método de ordenamiento de arreglos de *selección* tiene orden cuadrático en la cantidad de elementos del arreglo.

```
public static void selection_sort( int []a, int n ) {
    for( int i=0; i<n-1; i++ ) { // Repetir n-1 veces
        int k = i; // Hallar k tq a_k es el mínimo de
        for( int j=i+1; j<n; j++ ) {// a_i, a_{i+1},..., a_{n-1}
            if( a[j] < a[k] )
                k = j;}
        swap( a, i, k ); // Intercambiar a_i con a_k
    }
}
```

Ejercicio

Tamaño de la entrada: n = cantidad de componentes de a.

```
public static void selection_sort( int []a, int n ) {
    for( int i=0; i<n-1; i++ ) {   // n-1 iteraciones (desiguales)
        int k = i; // c1
        for( int j=i+1; j<n; j++ ) { // (n-1)-(i+1)+1) iteraciones
                if( a[j] < a[k] ) // Tiempo del if: c2
                        k = j;}
        swap( a, i, k ); // Tiempo de swap: c3
    }
}
```

$$T(n) = \sum_{i=0}^{n-2} (c_1 + ((n-1) - (i+1) + 1)c_2 + c_3)$$

Ejercicio

$$T(n) = \sum_{i=0}^{n-2} (c_1 + ((n-1) - (i+1) + 1)c_2 + c_3) =$$

$$= \sum_{i=0}^{n-2} c_1 + c_2 \sum_{i=0}^{n-2} (n - i - 1) + \sum_{i=0}^{n-2} c_3$$

$$= c_1(n - 2 - 0 + 1) + c_2 \sum_{i=0}^{n-2} (n - i - 1) + c_3(n - 2 - 0 + 1)$$

$$= c_1(n-1) + c_2 \sum_{i=0}^{n-2} (n - i - 1) + c_3(n-1)$$

Falta ver cuánto vale el término del medio.

Ejercicio

$$\sum_{i=0}^{n-2}(n-i-1) =$$

$$=(n-1)+(n-2)+(n-3)+\cdots+(n-(n-2)-1)=$$

$$=(n-1)+(n-2)+(n-3)+\cdots+1=$$

$$\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$$

Entonces, sabíamos que:

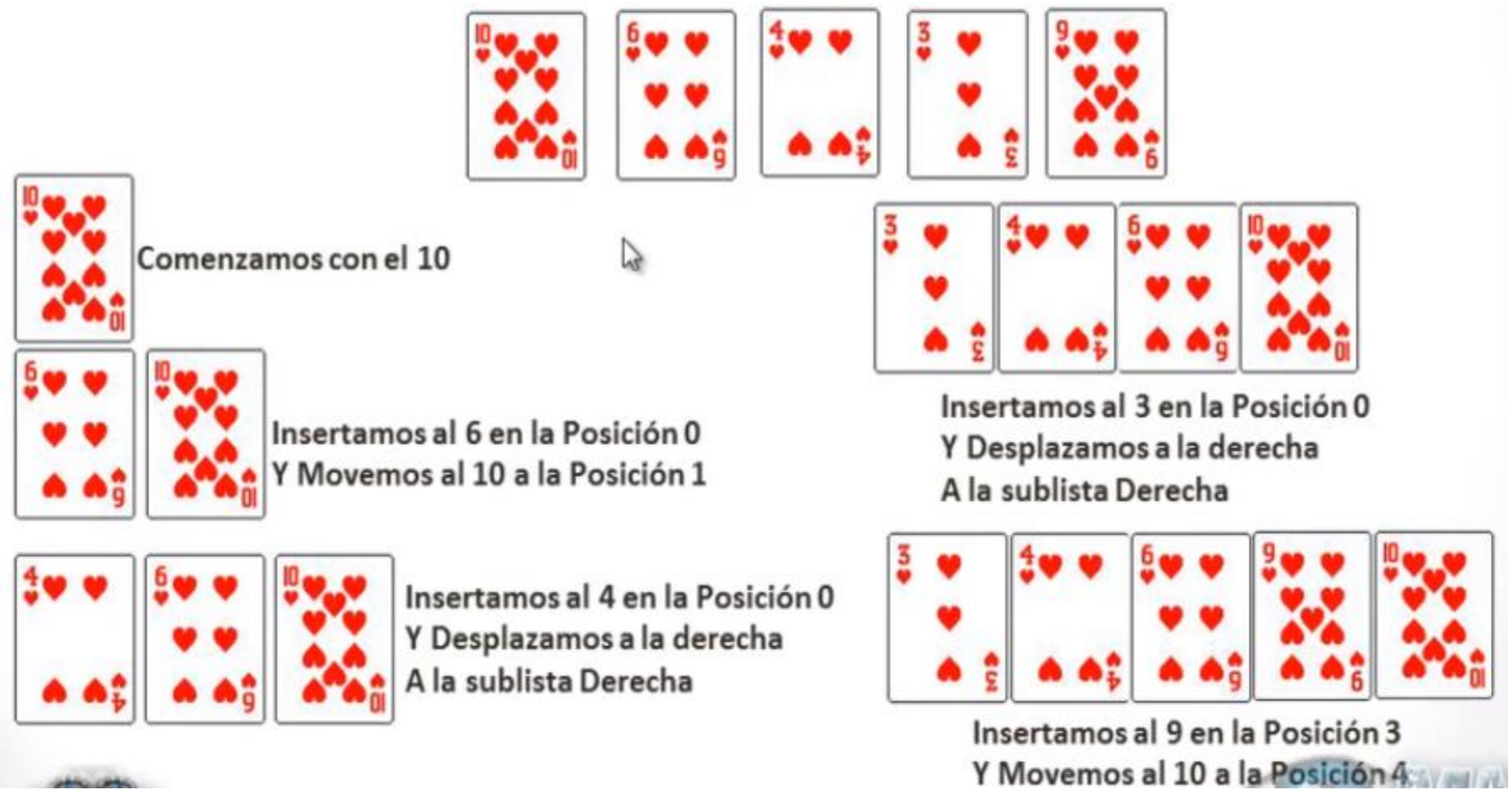
$$T(n) = c_1(n-1) + c_2\sum_{i=0}^{n-2}(n-i-1) + c_3(n-1) =$$

$$= c_1\,(n-1) + c_2\frac{n(n-1)}{2} + c_3(n-1) = O(n^2)$$

Ordenamiento por inserción

El método de ordenación por inserción es similar al proceso típico de ordenar tarjetas de nombres (cartas de una baraja) por orden alfabético consistente en insertar un nombre en su posición correcta dentro de una lista que ya está ordenada.

Algoritmo de ordenación por inserción

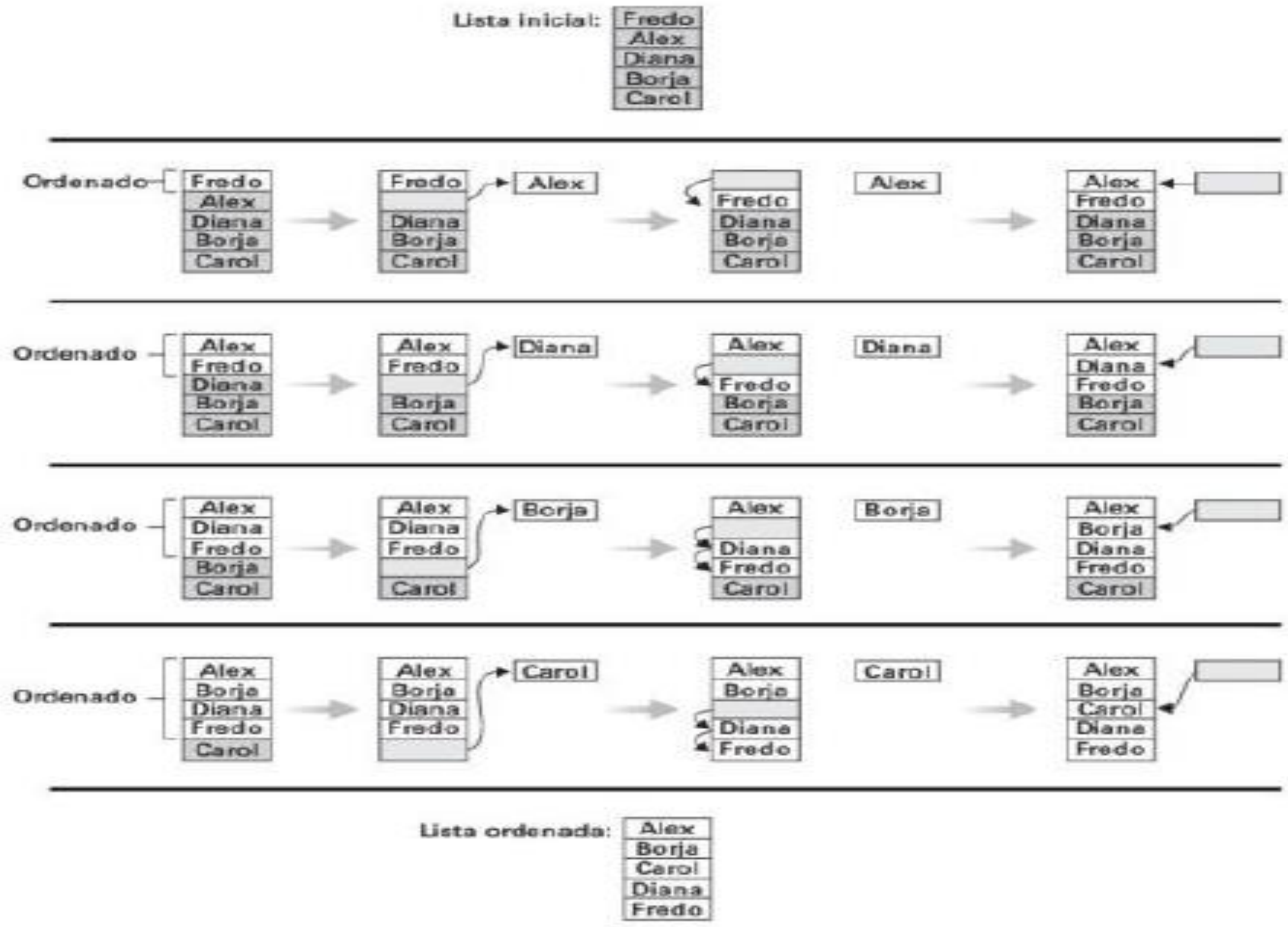


Ordenando alfabéticamente la lista Fredo, Alex, Diana, Borja y Carol

Supuesto: Vamos a ordenar la lista “dentro de si misma”. Se empieza diciendo que Fredo está ordenado, pero Fredo y Alex no lo está.

Algoritmo de ordenación por inserción

Recoger el primer nombre en la parte no ordenada de la lista, deslizar hacia abajo los nombres que sean posteriores alfabéticamente al nombre extraído y luego insertar otra vez en la lista el nombre extraído, en el lugar donde ha quedado un hueco.



Algoritmo de ordenación por inserción

Después de analizar el proceso de ordenación de una lista concreta, ahora se generaliza este proceso para obtener un algoritmo para ordenación de listas genéricas. De lo visto en la diapositiva anterior, cada paso representa el mismo proceso general: Recoger el primer nombre en la parte no ordenada de la lista, deslizar hacia abajo los nombres que sean posteriores alfabéticamente al nombre extraído y luego insertar otra vez en la lista el nombre extraído, en el lugar donde ha quedado un hueco. Si llamamos al nombre extraído con la denominación “entrada pivote”, este proceso puede ser expresado en nuestro pseudocódigo como:

```
Mover la entrada pivote a una ubicación temporal, dejando un hueco en la lista
while (exista un nombre por encima del hueco y dicho nombre sea mayor que el pivote):
    Mover al hueco el nombre situado por encima del mismo dejando un hueco encima de ese nombre
Mover la entrada pivote al hueco de la Lista
```

A continuación, observamos que este proceso debe ejecutarse de manera repetida. Para comenzar el proceso de ordenación, el pivote debe ser la segunda entrada en la lista y luego, antes de cada ejecución adicional, el pivote debe ser la entrada siguiente de la lista, hasta haber colocado adecuadamente la última entrada. Es decir, a medida que se repite la rutina que acabamos de descubrir, la posición inicial de la entrada pivote debe avanzar desde la segunda entrada a la tercera, luego a la cuarta, etc., hasta que la rutina haya terminado de colocar la última entrada de la lista.

Algoritmo de ordenación por inserción - Codificación

El algoritmo correspondiente a la ordenación por inserción contempla los siguientes pasos:

1. El primer elemento a[0] se considera ordenado; es decir, la lista inicial consta de un elemento.
2. Se inserta a[1] en la posición correcta; delante o detrás de a[0], dependiendo de si es menor o mayor.
3. Por cada bucle o iteración i *(desde* i=1 *hasta* n-1) se explora la sublista a[i-1] ... a[0] buscando la posición correcta de inserción de a[i]; a la vez, se mueven hacia abajo (a la derecha en la sublista) una posición todos los elementos mayores que el elemento a insertar a[i], para dejar vacía esa posición.
4. Insertar el elemento a[i] a la posición correcta.

```java
package ordenainterna;
public class MetodosBasicosOrdenacion {
    public static void ordInsercion (int [] a)
  {
     int i, j;
     int aux;
     for (i = 1; i < a.length; i++)
     {
      /*  indice j es para explorar la sublista a[i-1]..a[0]
          buscando la posicion correcta del elemento destino*/
       j = i;
       aux = a[i];
       // se localiza el punto de inserción explorando hacia abajo
       while (j > 0 && aux < a[j-1])
       {
         // desplazar elementos hacia arriba para hacer espacio
         a[j] = a[j-1];
         j--;
       }
       a[j] = aux;
     }
  }
}
```

Algoritmo de ordenación por inserción

```java
package ordenainterna;
public class MetodosBasicosOrdenacion {
   public static void ordInsercion (int [] a)
  {
    int i, j;
    int aux;
    for (i = 1; i < a.length; i++)
    {
     /*  indice j es para explorar la sublista a[i-1]..a[0]
         buscando la posicion correcta del elemento destino*/
      j = i;
      aux = a[i];
      // se localiza el punto de inserción explorando hacia abajo
      while (j > 0 && aux < a[j-1])
      {
        // desplazar elementos hacia arriba para hacer espacio
        a[j] = a[j-1];
        j--;
      }
      a[j] = aux;
    }
  }
}
```

5	6	2	4	7	3	1
5	6	2	4	7	3	1
2	5	6	4	7	3	1
2	4	5	6	7	3	1
2	4	5	6	7	3	1
2	3	4	5	6	7	1
1	2	3	4	5	6	7

Algoritmo de ordenación por inserción – Análisis de Complejidad

A la hora de analizar este algoritmo, se observa que el número de instrucciones que realiza depende del bucle automático for (bucle externo) que anida al bucle condicional while. Siendo n el número de elementos (n == a.length), el bucle externo realiza n-1 pasadas; por cada una de ellas y en el peor de los casos (aux siempre menor que a[j-1]), el bucle interno while itera un número creciente de veces que da lugar a la sucesión 1, 2, 3, ... n-1 (para i == n-1). La suma de los términos de la sucesión se ha obtenido antes y es n(n-1)/2.

Vector ordenado en origen

Comparaciones mínimas: $(n - 1)$

Vector inicialmente en orden inverso

Comparaciones máximas: $n(n - 1)/2$

Comparaciones medias: $((n - 1) + (n - 1)n/2)/2 = (n^2 + n - 2)/4$

Rendimiento en el mejor caso	O(n)
Rendimiento en el peor caso	$O(n^2)$
Rendimiento en el caso promedio	$O(n^2)$
Ordenamiento estable?	No
Complejidad de espacio	O(1), se necesita sólo una variable temporal

Algoritmos de Ordenamiento

https://www.youtube.com/watch?v=nmhjrI-aW5o

https://www.youtube.com/watch?v=xWBP4lzkoyM

https://www.youtube.com/watch?v=OGzPmgsI-pQ

BUBBLE SORT

SELECTION SORT

INSERTION SORT

GeeksforGeeks

A computer science portal for geeks

Ejercicio

Con el fin de medir el tiempo en milisegundos que emplea en ordenar un computador utilizando los métodos: burbuja, selección, inserción, se desea escribir un programa que genere un array de N elementos obtenidos al azar (N puede ser 1000, 10000, 100000, 1000000), se ordene aplicando cada uno de los algoritmos y mida el tiempo empleado por el computador.

Ordenamiento más rápido

"El ordenamiento burbuja parece no tener nada que lo recomiende, excepto un nombre pegadizo y el hecho de que conduce a algunos problemas teóricos interesantes"

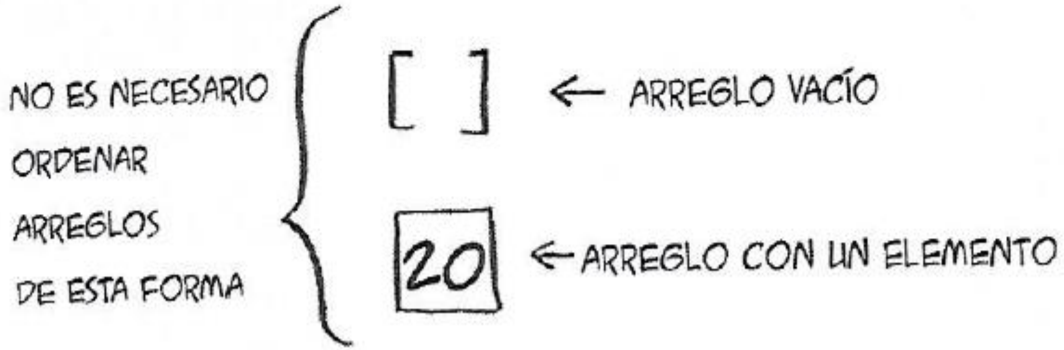
- Don Knuth

Ordenamientos previos

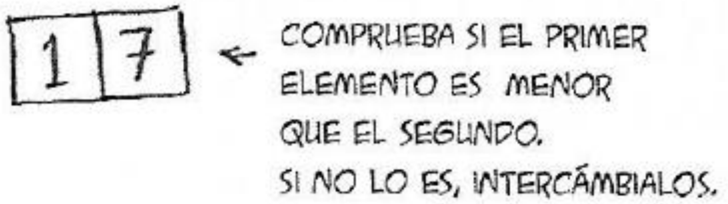
- El ordenamiento por inserción y por selección son casos promedio $O(N^2)$
- Ahora veremos dos algoritmos de clasificación más rápido.
  - quicksort
  - mergesort

Quicksort

Es un método de ordenación que utiliza Divide & Vencerás. ¿Cuál es el arreglo más sencillo que un algoritmo de ordenación puede manejar?

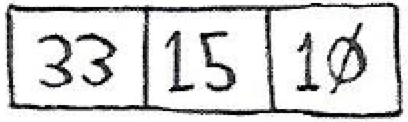


Los arreglos vacíos o con un solo elemento definen el caso base. Puedes devolver esos arreglos como si no hubiese nada que ordenar.



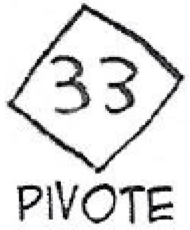
Un arreglo de dos elementos también es fácil de ordenar

¿Y qué pasaría con arreglo de 3 elementos?

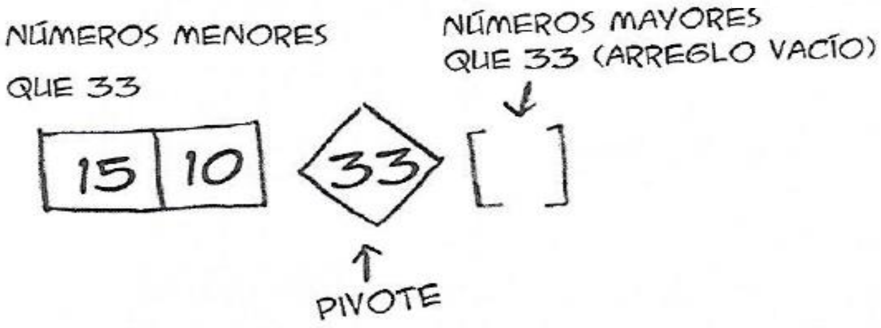


Quicksort funciona como sigue

1. Primero escoges un elemento del arreglo llamado pivote. Por ahora digamos que el pivote es el primer elemento del arreglo.



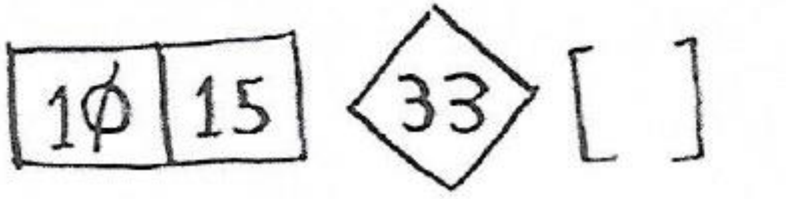
2. Ahora encuentra los elementos más pequeños que el pivote y los elementos más grandes que el pivote



Ahora tienes: Un subarreglo con todos los números menores que el pivote, el pivote y un subarreglo con todos los elementos mayores que el pivote

Quicksort

Los 2 arreglos no están ordenados simplemente particionados, pero si estuvieran ordenados, entonces la ordenación del arreglo completo sería muy sencilla



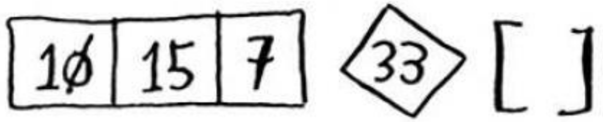
Si los subarreglos están ordenados la solución para obtener un arreglo ordenado es: arreglo izquierdo + pivote + arreglo derecho

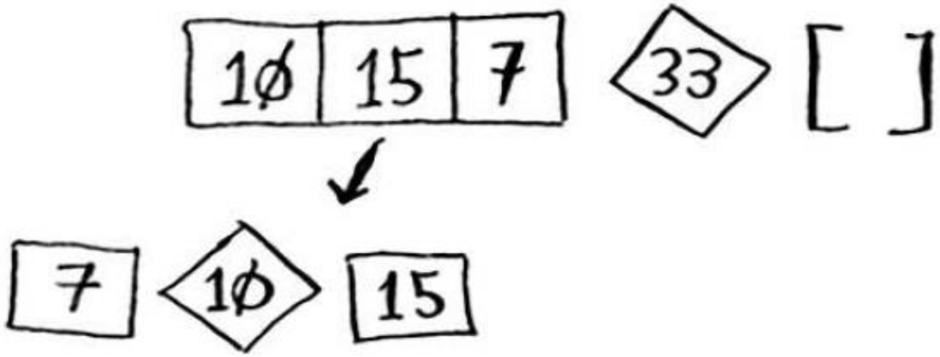
¿Cómo ordenas los subarreglos? El caso base de quicksort ya puede ordenar arreglo de dos elementos

```
quicksort([15, 10]) + [33] + quicksort([])
> [10, 15, 33]
```
A sorted array

Entonces los pasos para ordenar un arreglo con Quicksort sería: 1. Escoger un pivote 2. Particionar el arreglo en dos subarreglos: los elementos menores al pivote y los elementos mayores al pivote 3. Ejecutar Quicksort recursivamente en ambos subarreglos

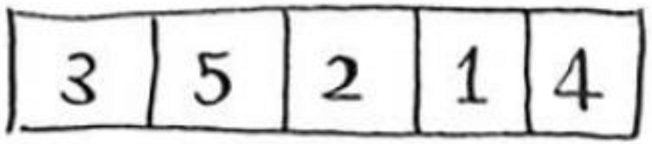
¿Qué tal un arreglo de 4 elementos?





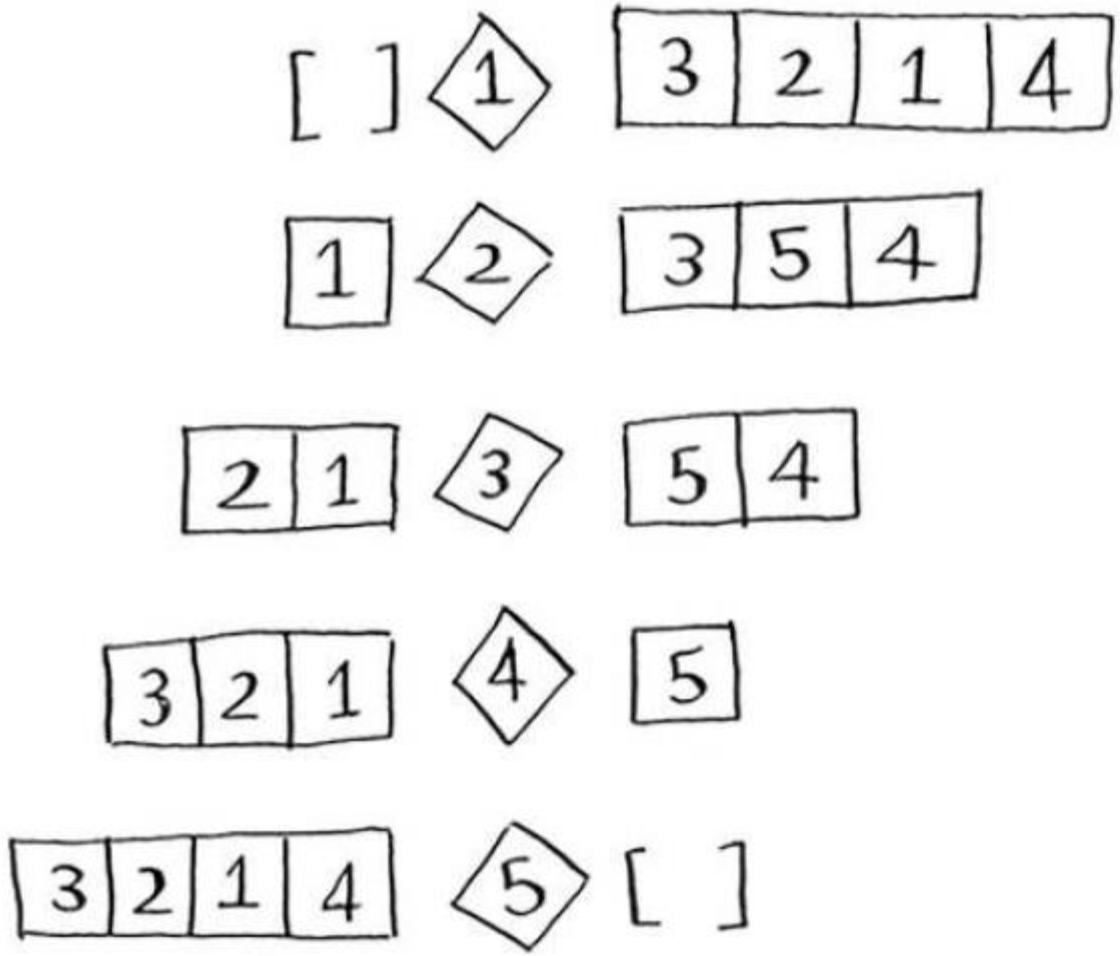
Quicksort

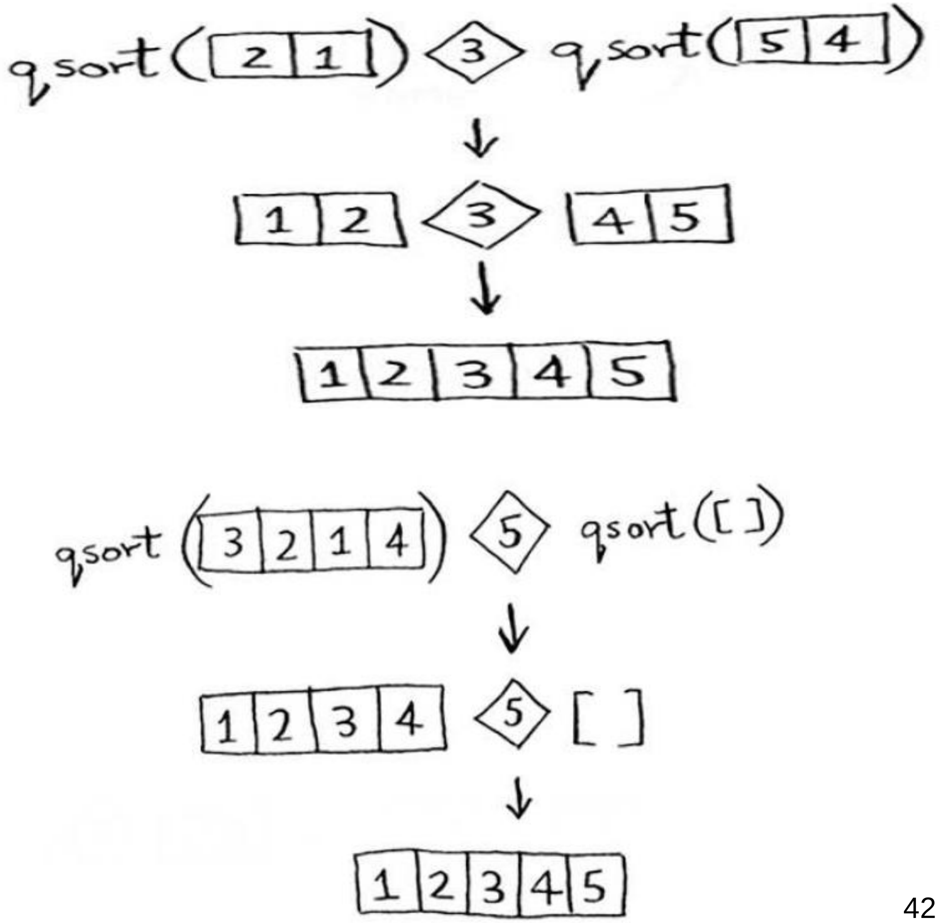
Supón que tienes un arreglo de 5 elementos



Estas son las maneras en que puedes particionar el arreglo dependiendo en qué pivote escojas:

Podemos llamar a Quicksort recursivamente en ambos subarreglos sin importar cuál es el pivote





Algoritmo Quicksort

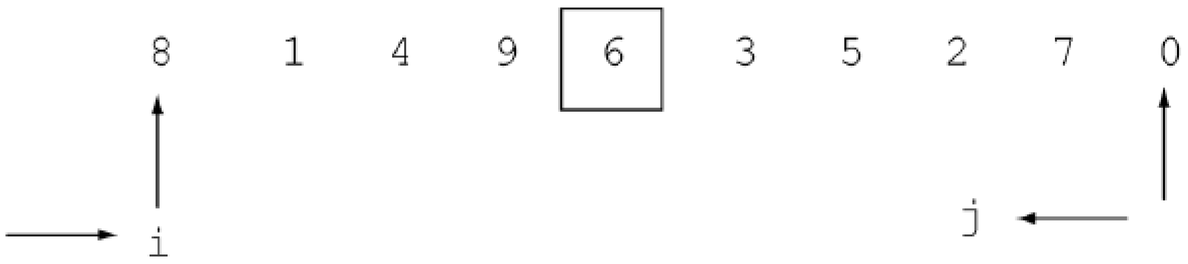
Lista original: 8 1 4 9 6 3 5 2 7 0

Pivote: 6

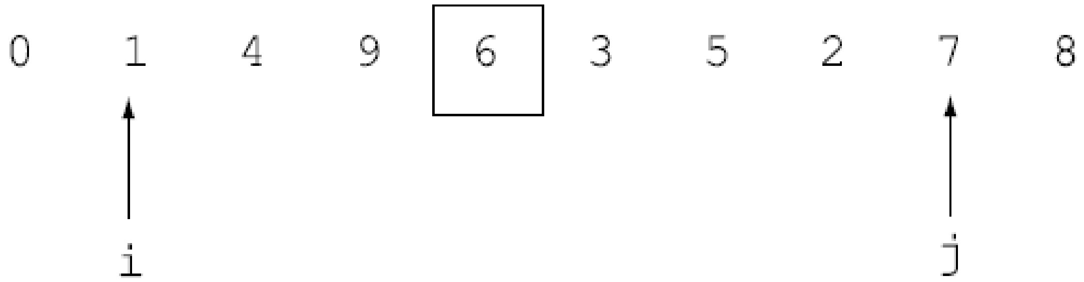
como pivote se elige el elemento central de la lista

Se recorre la lista de izquierda a derecha utilizando un índice i que se inicializa a la posición más baja (inferior) buscando un elemento mayor al pivote.

También se recorre la lista de derecha a izquierda buscando un elemento menor. Para hacer esto se utilizará un índice j inicializado a la posición más alta (superior).



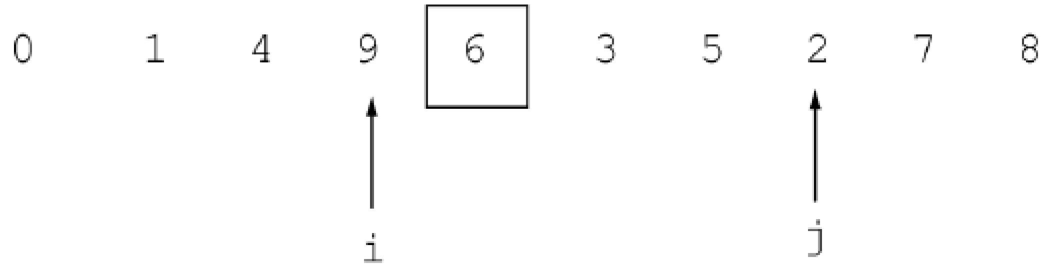
El índice i se detiene en el elemento 8 (mayor que el pivote) y el índice j se detiene en el elemento 0 (menor que el pivote). Ahora se intercambian 8 y 0 para que estos dos elementos se sitúen correctamente en cada sublista; y se incrementa el índice i, y se decrementa j para seguir los intercambios.



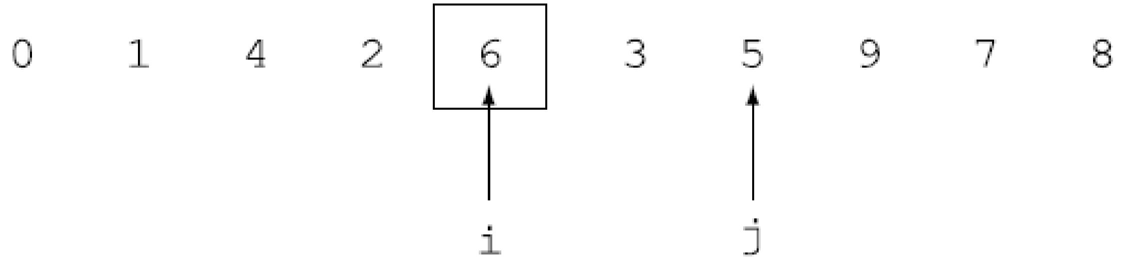
Algoritmo Quicksort

A medida que el algoritmo continúa, i se detiene en el elemento mayor, 9, y j se detiene en el elemento menor, 2.

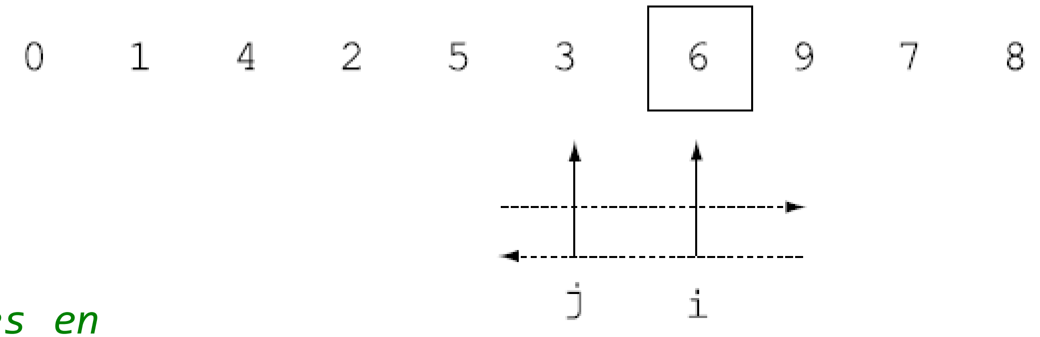
Se intercambian los elementos mientras que i y j no se crucen. En caso contrario, se detiene este bucle. En el caso anterior, se intercambian 9 y 2



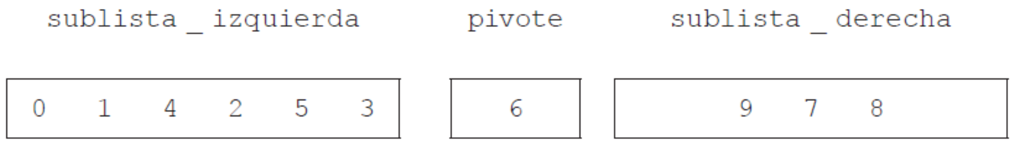
Continúa la exploración y ahora el contador i se detiene en el elemento 6 (que es el pivote) y el índice j se detiene en el elemento menor 5., se intercambian y se modifican los valores de i e j.



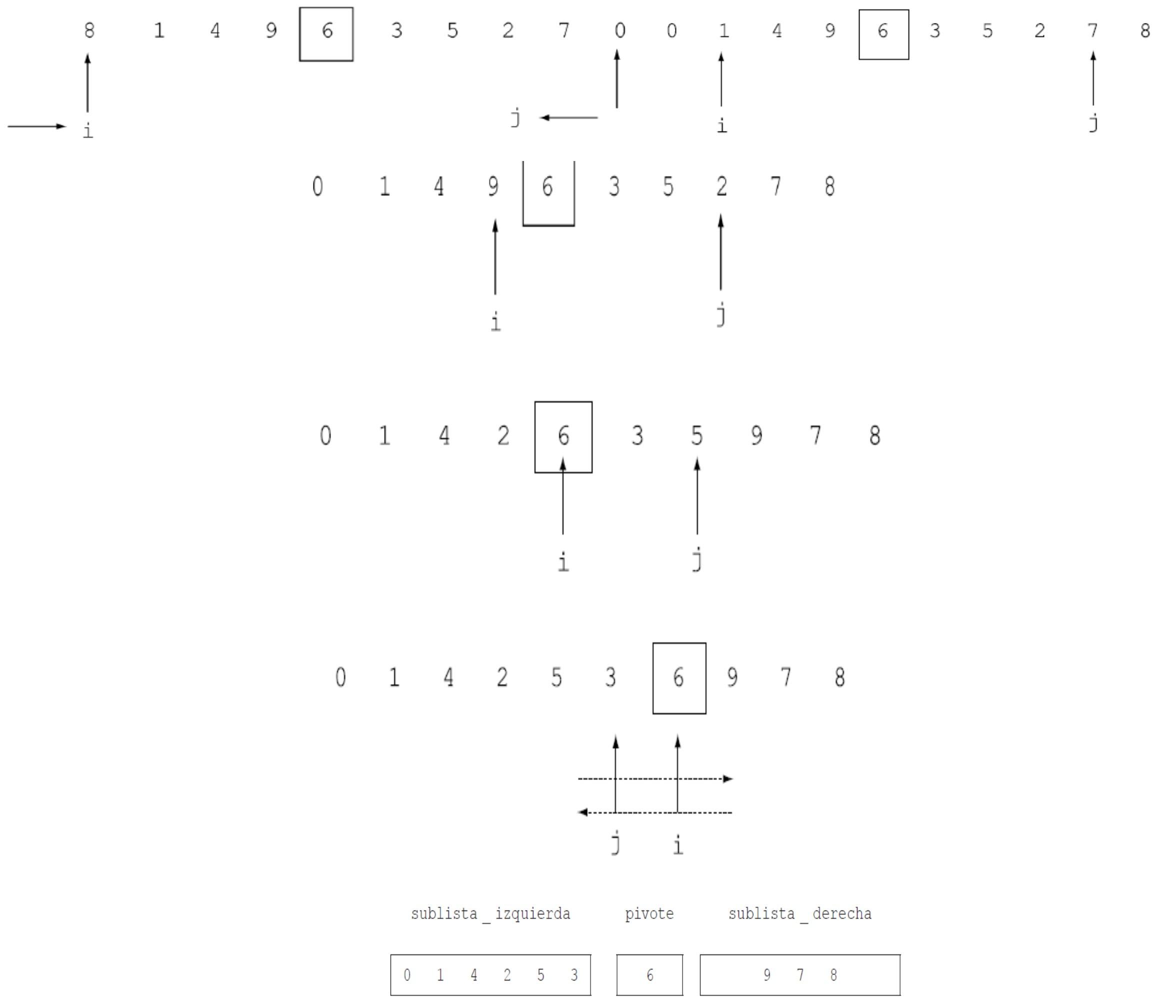
Los índices tienen actualmente los valores i = 5, j = 5. Continúa la exploración hasta que i > j, acaba con i = 6, j = 5.



En esta posición, los índices i y j han cruzado posiciones en el array. Se detiene la búsqueda y no se realiza ningún intercambio, ya que el elemento al que accede j está ya correctamente situado. Las dos sublistas ya han sido creadas, la lista original se ha dividido en dos particiones:



Algoritmo Quicksort - Codificación




```java
public static void intercambiar(int[] a, int i, int j){
    int aux=a[i];
    a[i]=a[j];
    a[j]=aux;
}
```

```java
public static void quicksort(int a[]) {
    quicksort(a, 0, a.length - 1);
}

private static void quicksort(int a[], int primero, int ultimo) {
    if (ultimo <= primero) {
        return;
    }
    int i, j, central;
    double pivote;
    central = (primero + ultimo) / 2;
    pivote = a[central];
    i = primero;
    j = ultimo;
    do {
        while (a[i] < pivote) {
            i++;
        }
        while (a[j] > pivote) {
            j--;
        }
        if (i <= j) {
            intercambiar(a, i, j);
            i++;
            j--;
        }
    } while (i <= j);
    if (primero < j) {
        quicksort(a, primero, j); // mismo proceso con sublista izqda
    }
    if (i < ultimo) {
        quicksort(a, i, ultimo); // mismo proceso con sublista drcha
    }
}
```

Algoritmo Quicksort – Análisis de Complejidad

Supongamos que n (número de elementos de la lista) es una potencia de 2, $n = 2^k$ ($k = \log_2 n$). Además, supongamos que el pivote es el elemento central de cada lista, de modo que quicksort divide la sublista en dos sublistas aproximadamente iguales. En la primera exploración o recorrido hay n-1 comparaciones. El resultado de la etapa crea dos sublistas aproximadamente de tamaño n/2. En la siguiente fase, el proceso de cada sublista requiere de aproximadamente n/2 comparaciones. Las comparaciones totales de esta fase son 2(n/2) = n. la siguiente fase procesa cuatro sublistas que requieren un total de 4*(n/4) comparaciones, etc*

Finalmente, el proceso de división termina después de k pasadas cuando la sublista resultante tenga tamaño 1. El número total de comparaciones es aproximadamente:

$$n + 2*(n/2) + 4*(n/4) + \ldots + n*(n/n) = n + n + \ldots + n = n * k = n * \log_2 n$$

Rendimiento en el peor caso	$O(n^2)$
Rendimiento en el mejor caso	O(nlogn)
Rendimiento en el caso promedio	O(nlogn)
Ordenamiento estable?	No

Algoritmo Quicksort – Análisis T(n)

Código	Mejor caso	Peor caso
private static void quicksort(int a[], int primero, int ultimo) {		
if (ultimo <= primero) {	1	1
return;	1	1
}		
int i, j, central;	3	3
double pivote;	1	1
central = (primero + ultimo) / 2;	3	3
pivote = a[central];	2	2
i = primero;	1	1
j = ultimo;	1	1
do {		
while (a[i] < pivote) {	2*n/2 +2	2
i++;	2*n/2	
}		
while (a[j] > pivote) {	2*n/2 +2	2*(2*n/2 + 2)
j--;	2*n/2	2*2*n/2
}		
if (i <= j) {	1	2*1
int aux = a[i];	3	2*3
a[i] = a[j];	3	2*3
a[j] = aux;	2	2*2
i++;	2	
j--;	2	2*2
}		2*1
} while (i <= j);	1	
if (primero < j) {	1	1
quicksort(a, primero, j); // mismo proceso con sublista izqda		
}	T(n/2)	
if (i < ultimo) {	1	1
quicksort(a, i, ultimo); // mismo proceso con sublista drcha		T(n-1)
}	T(n/2)	
}		

$$T(n) = \begin{cases} c_1 \\ c_a + c_b n + T(n-1) \\ c_2 + c_3 n + 2T(\frac{n}{2}) \end{cases}$$

Caso base

Peor caso

Mejor caso

V=[1,2,3,4,5,6,7]

V=[8,9,10,2,5,6,7]

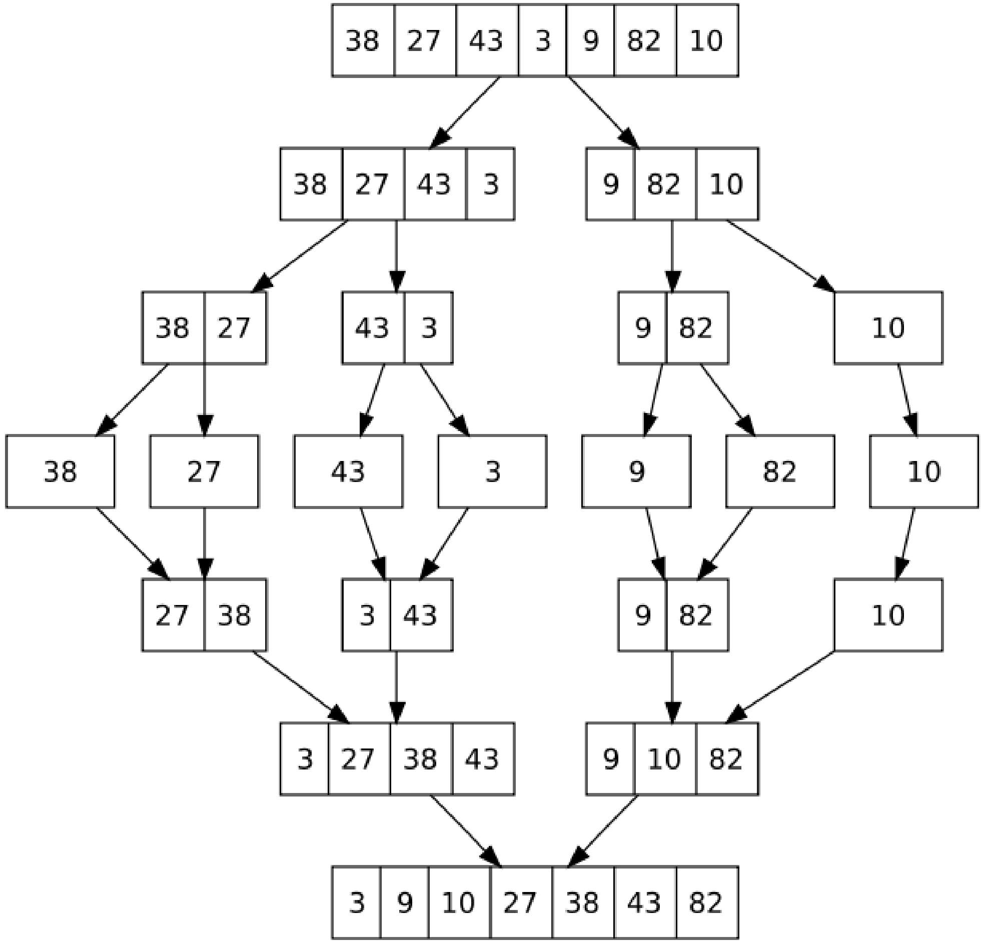
V=[9,12,11,7,4,2,3]

Algoritmo Merge Sort

Don Knuth cita a John von Neumann como el creador de este algoritmo

1. Si una lista tiene 1 elemento o 0 elementos, está ordenada
2. Si una lista tiene más de 1 elemento dividirla en 2 listas separadas
3. Realice este algoritmo en cada una de esas listas más pequeñas
4. Tome las 2 listas ordenadas y combínelas

Merge Sort



Cuando se implementa, se utiliza una matriz temporal, en lugar de varias matrices temporales.
¿Por qué?

Código Merge Sort

```java
    public static void mergesort(int a[]) {
        mergesort(a, 0, a.length - 1);
    }
    public static void mergesort(int[] a, int primero, int ultimo) {
        int central;
        if (primero < ultimo) {
            central = (primero + ultimo) / 2;
            mergesort(a, primero, central);
            mergesort(a, central + 1, ultimo);
            mezcla(a, primero, central, ultimo);
        }
    }
// mezcla de dos sublistas ordenadas
    public static void mezcla(int[] a, int izda, int medio, int drcha) {
        int[] tmp = new int[a.length];
        int i, k, z;
        i = z = izda;
        k = medio + 1;
        // bucle para la mezcla, utiliza tmp[] como array auxiliar
        while (i <= medio && k <= drcha) {
            if (a[i] <= a[k]) {
                tmp[z++] = a[i++];
            } else {
                tmp[z++] = a[k++];
            }
        }
        // se mueven elementos no mezclados de sublistas
        while (i <= medio) {
            tmp[z++] = a[i++];
        }
        while (k <= drcha) {
            tmp[z++] = a[k++];
        }
        // Copia de elementos de tmp[] al array a[]
        System.arraycopy(tmp, izda, a, izda, drcha - izda + 1);
    }
```

Pregunta

- ¿Cuál es el orden de Merge Sort: Notación Big O, mejor y peor caso?

	Mejor	Peor
A.	O(NlogN)	$O(N^2)$
B.	$O(N^2)$	$O(N^2)$
C.	$O(N^2)$	O(N!)
D.	O(NlogN)	O(NlogN)
E.	O(N)	O(NlogN)

Merge sort

```java
    public static void mergesort(int a[]) {
        mergesort(a, 0, a.length - 1);
    }
    public static void mergesort(int[] a, int primero, int ultimo) {
        int central;
        if (primero < ultimo) {
            central = (primero + ultimo) / 2;
            mergesort(a, primero, central);
            mergesort(a, central + 1, ultimo);
            mezcla(a, primero, central, ultimo);
        }
    }
// mezcla de dos sublistas ordenadas
    public static void mezcla(int[] a, int izda, int medio, int drcha) {
        int[] tmp = new int[a.length];
        int i, k, z;
        i = z = izda;
        k = medio + 1;
        // bucle para la mezcla, utiliza tmp[] como array auxiliar
        while (i <= medio && k <= drcha) {
            if (a[i] <= a[k]) {
                tmp[z++] = a[i++];
            } else {
                tmp[z++] = a[k++];
            }
        }
        // se mueven elementos no mezclados de sublistas
        while (i <= medio) {
            tmp[z++] = a[i++];
        }
        while (k <= drcha) {
            tmp[z++] = a[k++];
        }
        // Copia de elementos de tmp[] al array a[]
        System.arraycopy(tmp, izda, a, izda, drcha - izda + 1);
    }
```

Tamaño de la entrada: n = cantidad de componentes de a

Recurrencia para n:

$$T(n) = \begin{cases} c_1 & si \quad n = 1 \\ c_2 n + 2T(n/2) & si \quad n > 1 \end{cases}$$

$T(n) = c_2n + 2T(n/2) = c_2n + 2(c_2(n/2) + 2T(n/2/2))$

$= 2c_2n + 4T(n/4) = 2c_2n + 4(c_2(n/4) + 2T(n/4/2))$

$= 3c_2n + 8T(n/8) = \ldots$

$= ic_2n + 2^iT(n/2^i)$

Termina cuando $n/2^i = 1$, es decir $n=2^i$. Luego, $i = \log_2(n)$.

$T(n) = \log_2(n)c_2n + nT(1) = \log_2(n)c_2n + nc_1$

es $O(n\log_2(n))$

Teorema Maestro

Si tenemos un algoritmo cuya ecuación de recurrencia es:

$$\mathcal{T}(n) = A\mathcal{T}\left(\frac{n}{B}\right) + \mathcal{O}\left(n^{C}\right)$$

A: cantidad de llamados recursivos

B: proporción del tamaño original con el que llamamos recursivamente

$\mathcal{O}(n^{C})$: el costo de *partir y juntar* (todo lo que no son llamados recursivos)

$$\text{Si } \log_B(A) \begin{cases} < & \rightarrow \mathcal{T}(n) = \mathcal{O}\left(n^{C}\right) \\ = C & \rightarrow \mathcal{T}(n) = \mathcal{O}\left(n^{C}\log_B n\right) = \mathcal{O}\left(n^{C}\log n\right) \\ > & \rightarrow \mathcal{T}(n) = \mathcal{O}\left(n^{\log_B A}\right) \end{cases}$$

ShellSort

- Creado por Donald Shell en 1959
- Quería dejar de mover datos a pequeñas distancias (en el caso del ordenamiento por inserción y burbuja) y dejar de hacer intercambios que no son útiles (en el caso de la ordenación por selección)
- Comience con subarreglos creados al observar datos que están muy separados y luego reduzca el tamaño del salto o brecha (gap)

ShellSort en la práctica

46 2 83 41 102 5 17 31 64 49 18

Salto de cinco. Ordenar subarreglo de 46, 5 y 18

5 2 83 41 102 18 17 31 64 49 46

Salto todavía de cinco. Ordenar subarreglo de 2 y 17 (no se intercambia)

5 2 83 41 102 18 17 31 64 49 46

Salto todavía de cinco. Ordenar subarreglo de 83 y 31

5 2 31 41 102 18 17 83 64 49 46

Salto todavía de cinco. Ordenar subarreglo de 41 y 64 (no se intercambia)

5 2 31 41 102 18 17 83 64 49 46

Salto todavía de cinco. Ordenar subarreglo de 102 y 49

5 2 31 41 49 18 17 83 64 102 46

Continúa en la siguiente diapositiva:

Shellsort completo

5 2 31 41 49 18 17 83 64 102 46
Salto ahora de 2: Ordenar subarreglo de 5 31 49 17 64 46
5 2 17 41 31 18 46 83 49 102 64
Salto aún de 2: Ordenar subarreglo de 2 41 18 83 102
5 2 17 18 31 41 46 83 49 102 64
Salto de 1 (ordenamiento por inserción)
2 5 17 18 31 41 46 49 64 83 102

Arreglo ordenado!!

Shellsort sobre otro conjunto de datos

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
44	68	191	119	119	37	83	82	191	45	158	130	76	153	39	25

Salto inicial = length / 2 = 16 / 2 = 8

Indices del subarreglo:

{0, 8}, {1, 9}, {2, 10}, {3, 11}, {4, 12}, {5, 13}, {6, 14}, {7, 15}

próximo salto = 8 / 2 = 4

{0, 4, 8, 12}, {1, 5, 9, 13}, {2, 6, 10, 14}, {3, 7, 11, 15}

próximo salto = 4 / 2 = 2

{0, 2, 4, 6, 8, 10, 12, 14}, {1, 3, 5, 7, 9, 11, 13, 15}

Salto final = 2 / 2 = 1

Código Shellsort

```java
    public static void ordenacionShell(int a[]) {
        int intervalo, i, j, k;
        int n = a.length;

        intervalo = n / 2;
        while (intervalo > 0) {
            for (i = intervalo; i < n; i++) {
                j = i - intervalo;
                while (j >= 0) {
                    k = j + intervalo;
                    if (a[j] <= a[k]) {
                        j = -1;    // par de elementos ordenado
                    } else {
                        intercambiar(a, j, j + 1);
                        j -= intervalo;
                    }
                }
            }
            intervalo = intervalo / 2;
        }
    }
```

Algoritmos de Ordenamiento

https://www.youtube.com/watch?v=PgBzjlCcFvc

https://www.youtube.com/watch?v=JSceec-wEyw

https://www.youtube.com/watch?v=SHcPqUe2GZM

QUICK SORT

MERGE SORT

SHELL SORT

GeeksforGeeks

A computer science portal for geeks

Comparación de varios Algoritmos de ordenamiento

Num Items	Seleccion	Insercion	Shellsort	Quicksort
1000	16	5	0	0
2000	59	49	0	6
4000	271	175	6	5
8000	1056	686	11	0
16000	4203	2754	32	11
32000	16852	11039	37	45
64000	Espera demasiado?	Espera demasiado?	100	68
128000	Espera demasiado?	Espera demasiado?	257	158
256000	Espera demasiado?	Espera demasiado?	543	335
512000	Espera demasiado?	Espera demasiado?	1210	722
1024000	Espera demasiado?	Espera demasiado?	2522	1550

Tiempo en milisegundos

Comparación de varios tipos de ordenamiento (2011)

Num Items	Selección	Inserción	Quicksort	Merge	Arrays.sort
1000	0.002	0.001	-	-	-
2000	0.002	0.001	-	-	-
4000	0.006	0.004	-	-	-
8000	0.022	0.018	-	-	-
16000	0.086	0.070	0.002	0.002	0.002
32000	0.341	0.280	0.004	0.005	0.003
64000	1.352	1.123	0.008	0.010	0.007
128000	5.394	4.499	0.017	0.022	0.015
256000	21.560	18.060	0.035	0.047	0.031
512000	86.083	72.303	0.072	0.099	0.066
1024000	???	???	0.152	0.206	0.138
2048000			0.317	0.434	0.287
4096000			0.663	0.911	0.601
8192000			1.375	1.885	1.246

Comparación de varios tipos de ordenamiento (2020)

Num Items	Selección	Inserción	Quicksort	Mergesort	Arrays. sort(int)	Arrays.sort(Integer)	Arrays. parallelSort
1000	<0.001	<0.001	-	-	-	-	
2000	0.001	<0.001	-	-	-	-	
4000	0.004	0.003	-	-	-	-	
8000	0.017	0.010	-	-	-	-	
16000	0.065	0.040	0.002	0.002	0.003	0.011	0.007
32000	0.258	0.160	0.002	0.003	0.002	0.008	0.003
64000	1.110	0.696	0.005	0.008	0.004	0.011	0.001
128000	4.172	2.645	0.011	0.015	0.009	0.024	0.002
256000	16.48	10.76	0.024	0.034	0.018	0.051	0.004
512000	70.38	47.18	0.049	0.68	0.040	0.114	0.008
1024000	-	-	0.098	0.143	0.082	0.259	0.017
2048000	-	-	0.205	0.296	0.184	0.637	0.035
4096000	-	-	0.450	0.659	0.383	1.452	0.079
8192000	-	-	0.941	1.372	0.786	3.354	0.148

Algunas consideraciones

- Las librerías de los Lenguajes de programación incluyen a menudo algoritmos de ordenamiento
  - Las clases Arrays y Collections de Java
  - C++ Standard Template Library
  - Python sort and sorted functions
- Ordenamiento híbrido (Hybrid sorts)
  - cuando el tamaño de la lista no ordenada o una parte del arreglo es pequeño, use la ordenación por inserción, de lo contrario use la ordenación O (N log N) como Quicksort o Mergesort

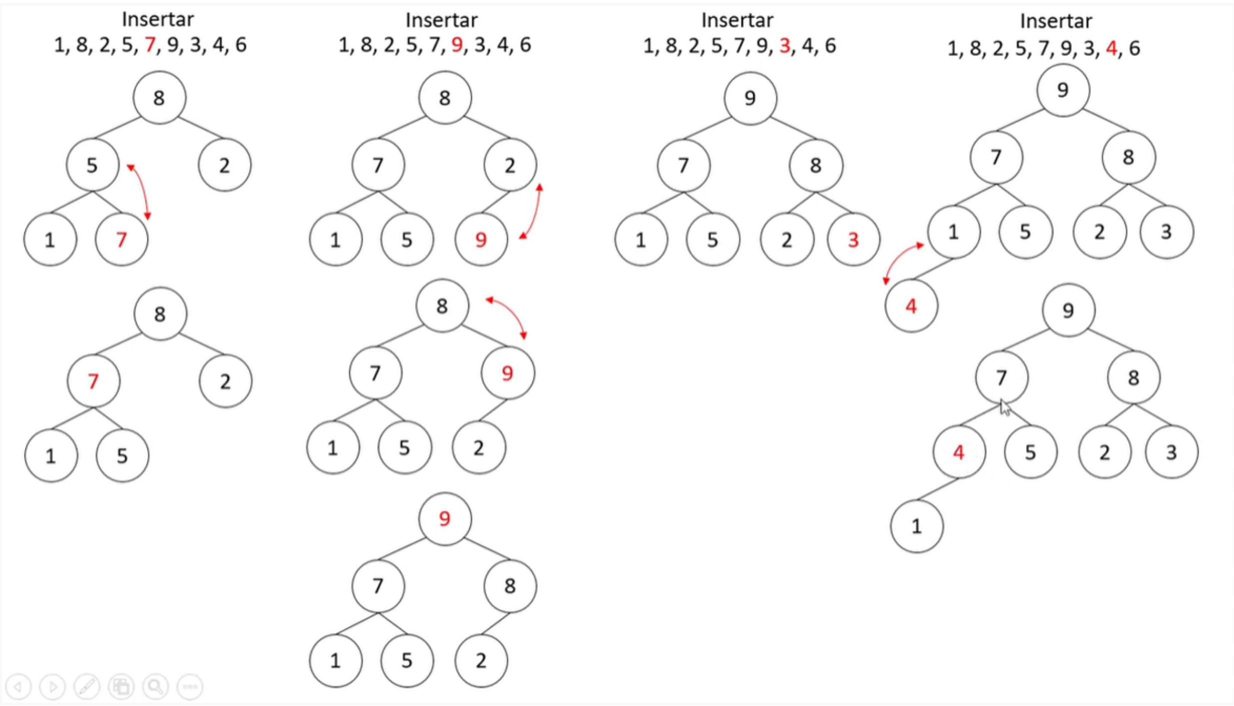
Algunas consideraciones

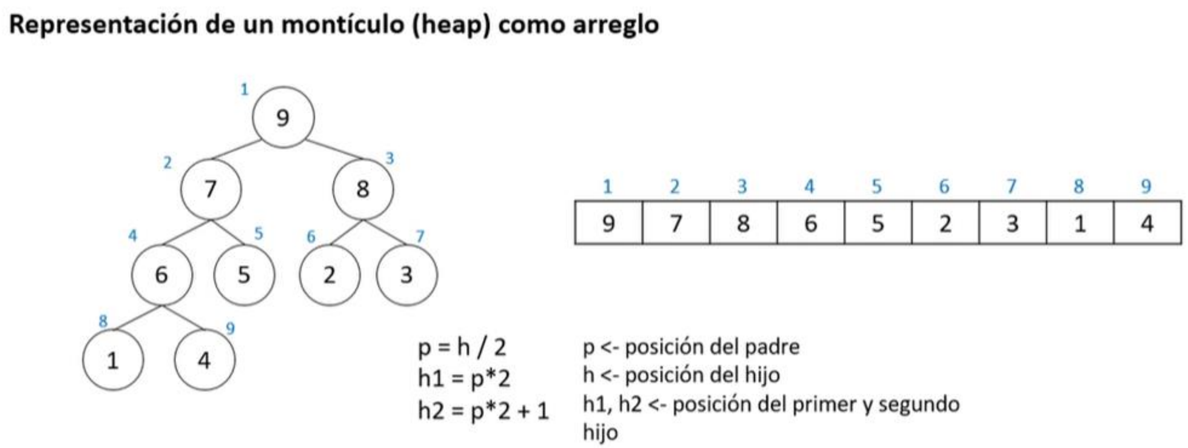
- ¡Aún se están creando tipos!
- Timsort (2002)
  - Creado para python version 2.3
  - Ahora usado en Java version 7.0+
  - Toma ventaja de los datos del mundo real
  - Los datos del mundo real suelen estar parcialmente ordenados, no totalmente aleatorios
- Library Sort (2006)
  - Como la ordenación por inserción, pero deja espacios para elementos posteriores.

Algoritmo Heap Sort

https://www.youtube.com/watch?v=Hq3OD8dUTCM

https://www.youtube.com/watch?v=x4J5Mcyzdxk



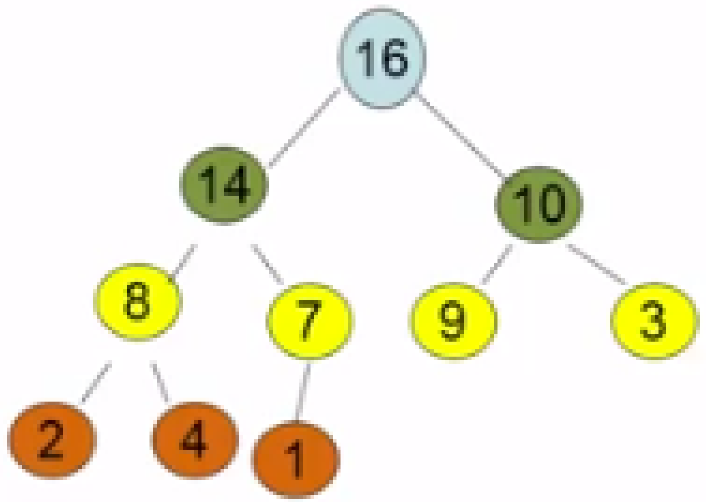


Algoritmo Heap Sort

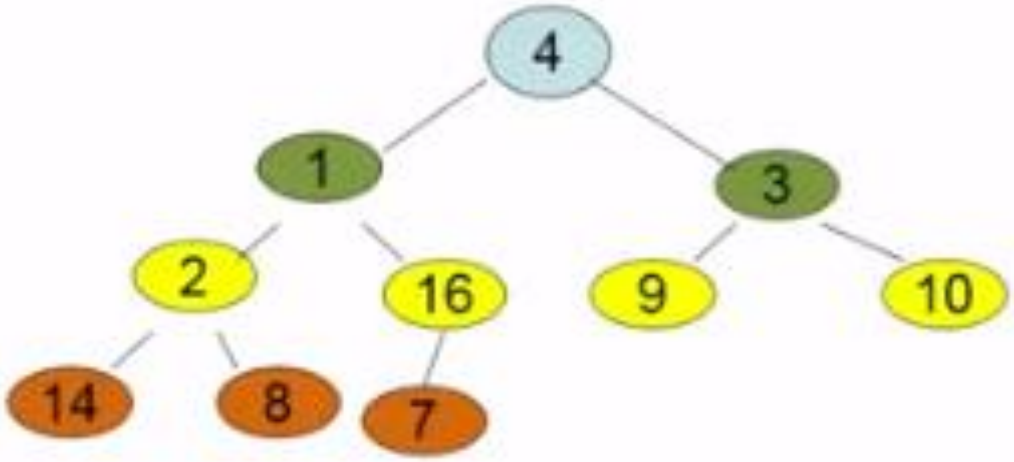
Heap sort

La filosofía de este método de ordenación consiste en aprovechar la estructura particular de los **montículos (heaps)**, que son **árboles binarios completos (todos sus niveles están llenos salvo a lo sumo el último, que se rellena de izquierda a derecha)** y cuyos nodos verifican la propiedad del montículo: *todo nodo es mayor o igual que cualquiera de sus hijos. En consecuencia, en la raíz se encuentra siempre el elemento mayor.*

	1	2	3	4	5	6	7	8	9	10
K	16	14	10	8	7	9	3	2	4	1



	0	1	2	3	4	5	6	7	8	9
K	4	1	3	2	16	9	10	14	8	7



N= cantidad de elementos

PP = (N / 2)-1

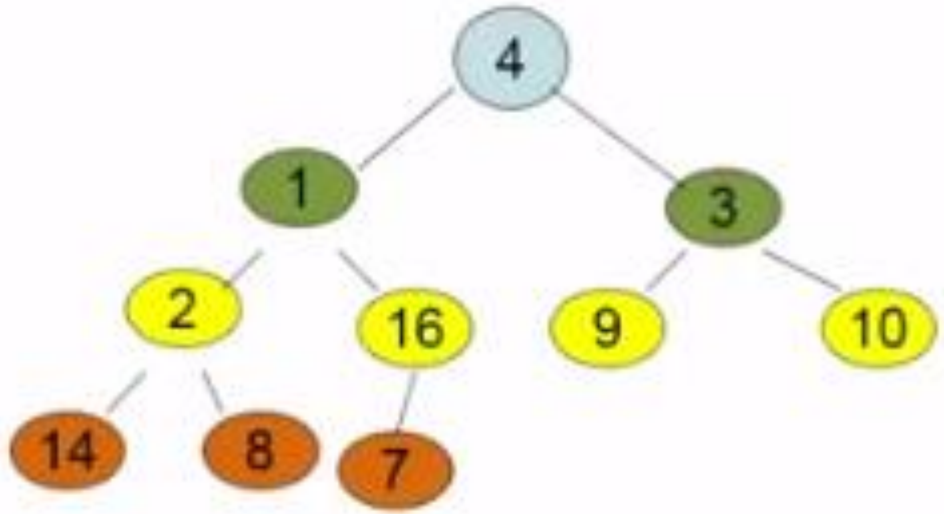
Posición de Hijos

Hijo Izquierda : HI = (PP * 2) +1

Hijo Derecha : HD = (PP * 2) + 2

* posiciones debe ser igual o menor a N.

Algoritmo Heap Sort



En el Padre se coloca el valor mas alto del montículo

	0	1	2	3	4	5	6	7	8	9
K	4	1	3	2	16	9	10	14	8	7

N= 10

PP = 4

HI = 9

HD = No tiene

N= cantidad de elementos

PP = (N / 2)-1

Posición de Hijos

Hijo Izquierda : HI = (PP * 2) +1

Hijo Derecha : HD = (PP * 2) + 2

* posiciones debe ser igual o menor a N.

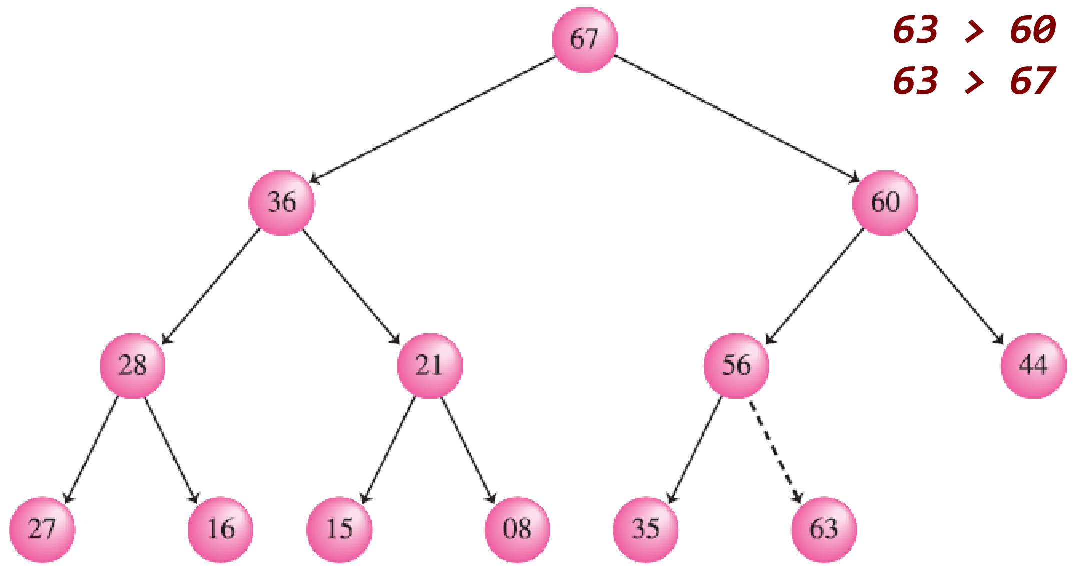
Inserción de un elemento en un montículo

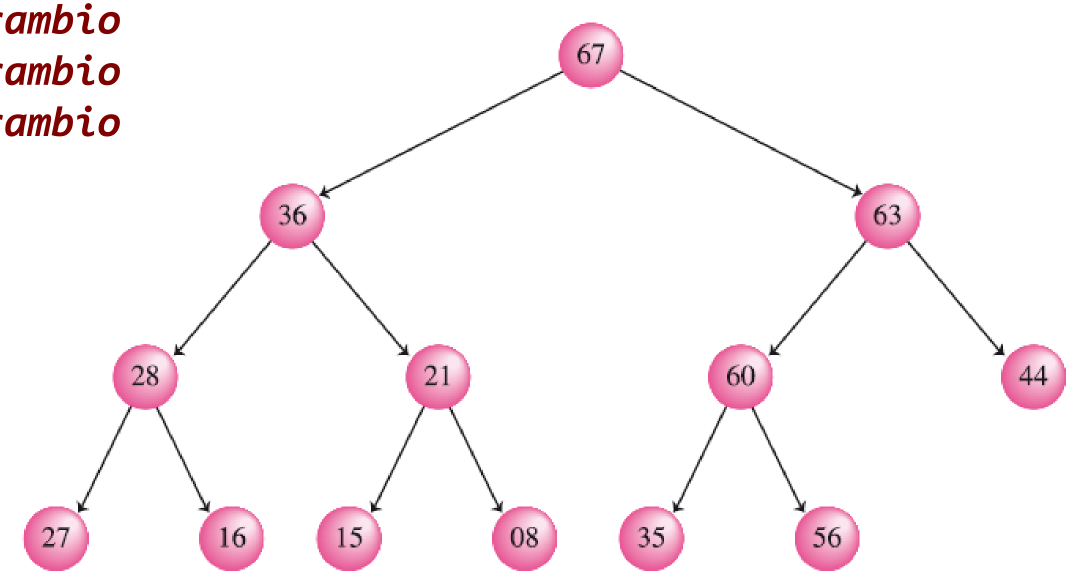
La inserción de un elemento en un montículo se lleva a cabo por medio de los siguientes pasos:

1. *Se inserta el elemento en la primera posición disponible.*
2. *Se verifica si su valor es mayor que el de su padre. Si se cumple esta condición entonces se efectúa el intercambio. Si no se cumple esta condición entonces el algoritmo se detiene y el elemento queda ubicado en la posición correcta en el montículo.*

Supongamos que se quiere incorporar al montículo de la figura el elemento 63, las comparaciones que realizamos son:

63 > 56 si hay intercambio
63 > 60 si hay intercambio
63 > 67 no hay intercambio

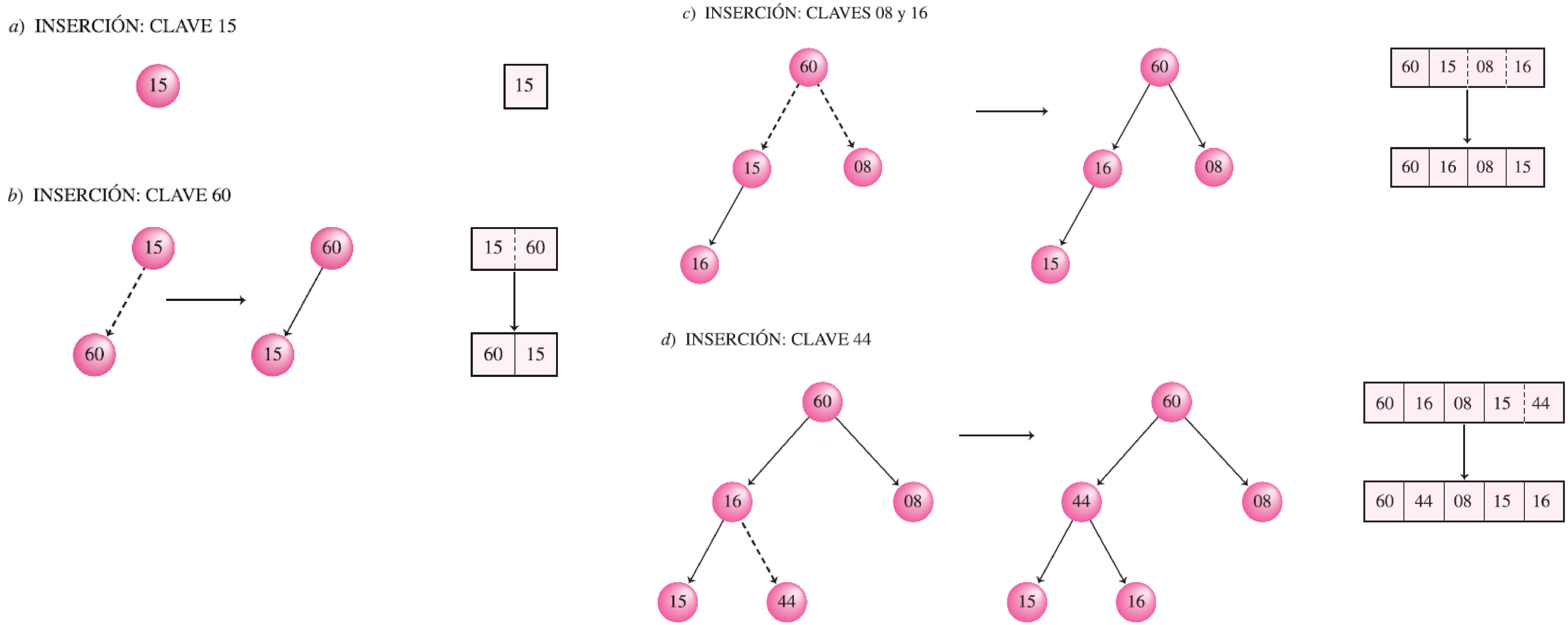




Inserción de un elemento en un montículo

Supongamos que se desea insertar las siguientes claves en un montículo que se encuentra vacío:

15 60 08 16 44 27 12 35

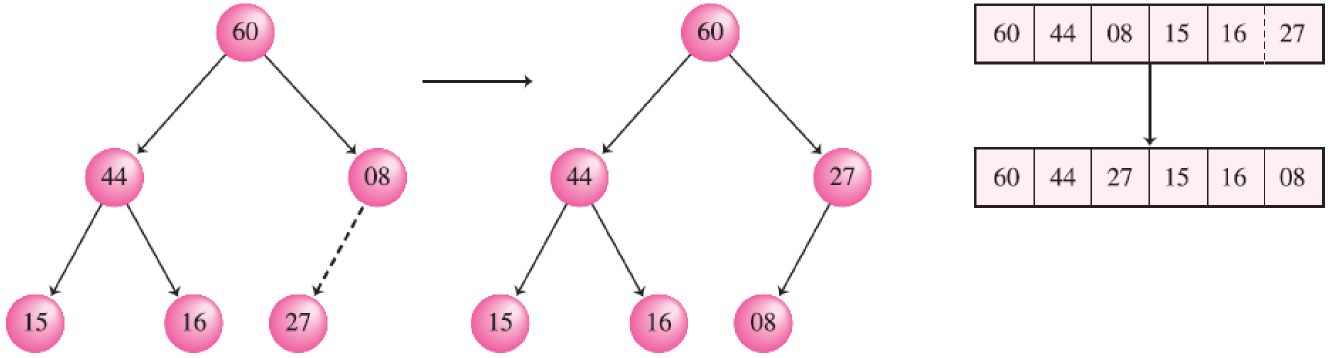


Inserción de un elemento en un montículo

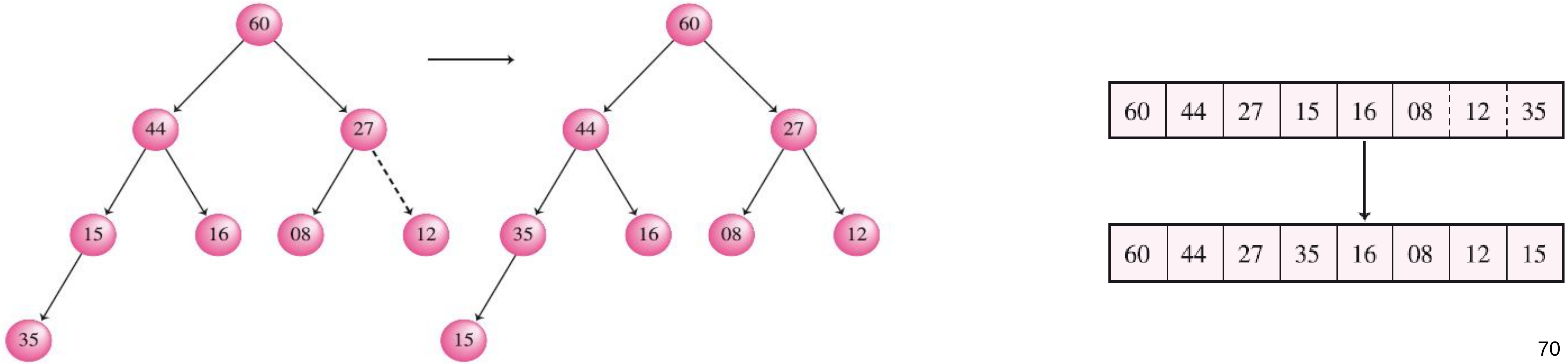
Supongamos que se desea insertar las siguientes claves en un montículo que se encuentra vacío:

15 60 08 16 44 27 12 35

e) INSERCIÓN: CLAVE 27



f) INSERCIÓN: CLAVES 12 y 35

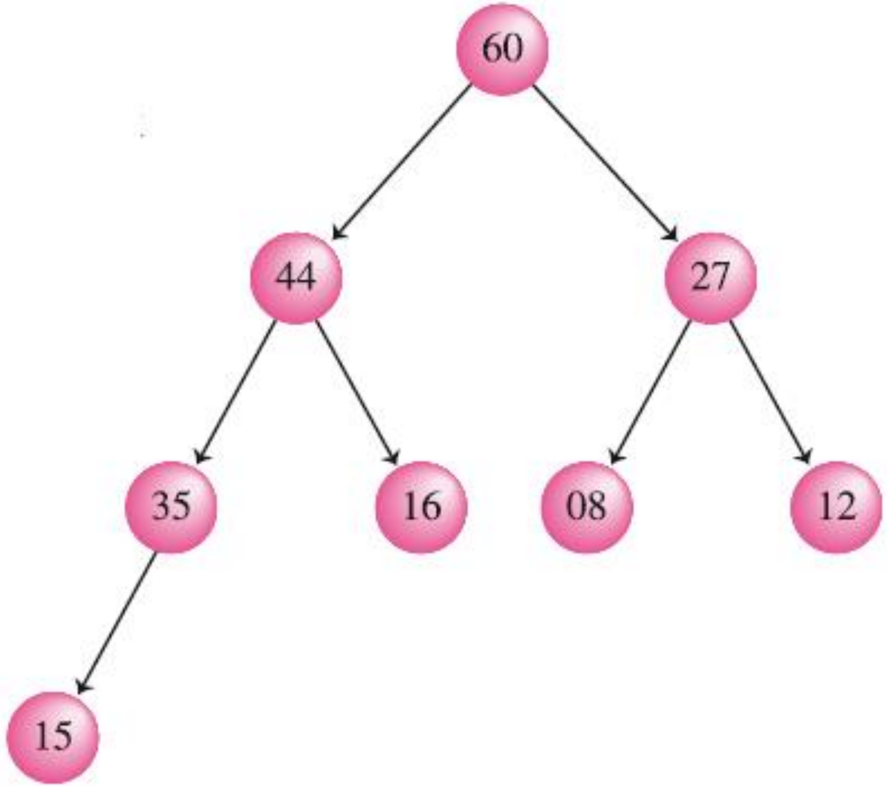


Inserción de un elemento en un montículo

Ejercicios

Dado el montículo de la figura, verifique como queda luego de insertar las siguientes claves

56 21 13 28 67 36 07 10



67	56	60	44	21	28	36	15	35	16	13	08	27	12	07	10

Algoritmo de Inserción de un elemento en un montículo

```java
    public static void insertamonticulo(int[] a) {
        int i, j;
        int aux;
        boolean band;
        for (i = 1; i <= a.length-1; i++) {
            j = i;
            band = true;
            while (j > 0 && band == true) {
                band = false;
                if (a[j] > a[j / 2]) {
                    aux = a[j / 2];
                    a[j / 2] = a[j];
                    a[j] = aux;
                    j = j / 2;
                    band = true;
                }
            }
        }
    }
```

Eliminación de un montículo

El proceso para obtener los elementos ordenados se efectúa eliminando la raíz del montículo en forma repetida. Los pasos necesarios para lograr la eliminación de la raíz de un montículo son:

1. Se reemplaza la raíz con el elemento que ocupa la última posición del montículo.

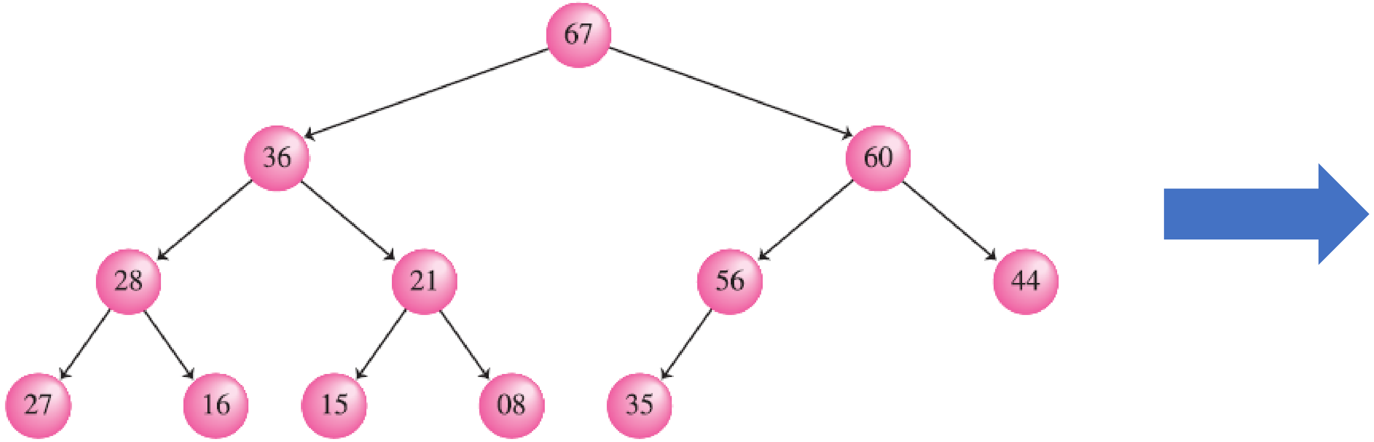
2. Se verifica si el nuevo valor de la raíz es menor que el valor mas grande de sus hijos. Si se cumple la condición, entonces se efectúa el intercambio. Si no se cumple la condición entonces el algoritmo se detiene y el elemento queda ubicado en la posición correcta en el montículo.

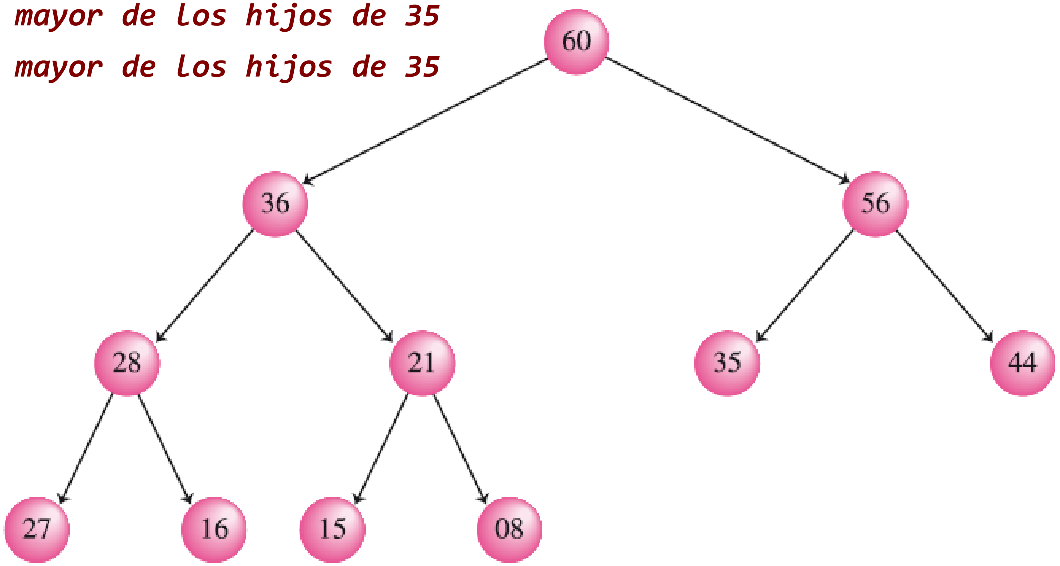
Se repite desde 1.

Supongamos que se desea eliminar la raíz del montículo (67) de la figura:

35 < 60 si hay intercambio, *60 es el mayor de los hijos de 35*

35 < 56 si hay intercambio, *56 es el mayor de los hijos de 35*





Eliminación de un montículo

Supongamos que se desea eliminar la raíz del montículo, presentado como arreglo, en forma repetida.

67	56	60	44	21	28	36	15	35	16	13	08	27	12	07	10
0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*	*15*

Cabe aclarar que al reemplazar la raíz por el último elemento del montículo, ésta se coloca en la posición del último elemento del arreglo. Es decir, la primera vez la raíz será colocada en el índice n-1 del arreglo, la segunda vez en el índice n-2, la tercera vez en el índice n-3 y así sucesivamente hasta llegar al índice 1 y 0.

Primera eliminación:

Se intercambia la raíz, 67 con el elemento que ocupa la última posición del montículo, 10. Las comparaciones que se realizan son:
A[0] < A[2] (10 < 60) sí hay intercambio, A[2] es el mayor de los hijos de A[0]
A[2] < A[6] (10 < 36) sí hay intercambio, A[6] es el mayor de los hijos de A[2]
A[6] < A[13] (10 < 12) sí hay intercambio, A[13] es el mayor de los hijos de A[6]

Luego de eliminar la primera raíz, el montículo queda así:

60	56	36	44	21	28	12	15	35	16	13	08	27	10	07	$\boxed{67}$
0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*	*15*

!El mayor se ubicó en la última posición!

Eliminación de un montículo

Segunda eliminación:

Se intercambia la raíz, 60 con el elemento que ocupa la última posición del montículo, 07. Las comparaciones que se realizan son:
A[0] < A[2] (07 < 56) sí hay intercambio, A[1] es el mayor de los hijos de A[0]
A[1] < A[3] (07 < 44) sí hay intercambio, A[3] es el mayor de los hijos de A[1]
A[3] < A[8] (07 < 35) sí hay intercambio, A[8] es el mayor de los hijos de A[3]

Luego de eliminar la segunda raíz, el montículo queda así:

56	44	36	35	21	28	12	15	07	16	13	08	27	10	60	67
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

!El mayor se ubicó en la última posición!

Tercera eliminación:

Se intercambia la raíz, 56 con el elemento que ocupa la última posición del montículo, 10. Las comparaciones que se realizan son:
A[0] < A[1] (10 < 44) sí hay intercambio, A[1] es el mayor de los hijos de A[0]
A[1] < A[3] (10 < 35) sí hay intercambio, A[3] es el mayor de los hijos de A[1]
A[3] < A[7] (10 < 15) sí hay intercambio, A[7] es el mayor de los hijos de A[3]

Luego de eliminar la tercera raíz, el montículo queda así:

44	35	36	15	21	28	12	10	07	16	13	08	27	56	60	67
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Eliminación de un montículo

Se presenta el resultado de las restantes eliminaciones. Luego de eliminar la raíz del montículo, en forma repetida, el arreglo queda ordenado

Eliminación	Montículo															
4	36	35	28	15	21	27	12	10	07	16	13	08	44	56	60	67
5	35	21	28	15	16	27	12	10	07	08	13	36	44	56	60	67
6	28	21	27	15	16	13	12	10	07	08	35	36	44	56	60	67
7	27	21	13	15	16	08	12	10	07	28	35	36	44	56	60	67
8	21	16	13	15	07	08	12	10	27	28	35	36	44	56	60	67
9	16	15	13	10	07	08	12	21	27	28	35	36	44	56	60	67
10	15	12	13	10	07	08	16	21	27	28	35	36	44	56	60	67
11	13	12	08	10	07	15	16	21	27	28	35	36	44	56	60	67
12	12	10	08	07	13	15	16	21	27	28	35	36	44	56	60	67
13	10	07	08	12	13	15	16	21	27	28	35	36	44	56	60	67
14	08	07	10	12	13	15	16	21	27	28	35	36	44	56	60	67
15	07	08	10	12	13	15	16	21	27	28	35	36	44	56	60	67

Algoritmo de eliminación de la raíz de un montículo

```java
public static void eliminamonticulo(int[] a) {
    int i, j;
    int aux, izq, der, ap,mayor;
    boolean bool;
    for (i = a.length - 1; i > 0; i--) {
        aux = a[i];
        a[i] = a[0];
        izq = 1;
        der = 2;
        j = 0;
        bool = true;
        while ((izq < i) && (bool == true)) {
            mayor = a[izq];
            ap = izq;
            if (mayor < a[der] && der != i) {
                mayor = a[der];
                ap = der;
            }
            if (aux < mayor) {
                a[j] = a[ap];
                j = ap;
            } else {
                bool = false;
            }
            izq = j * 2 ;
            der = izq + 1;
        }
        a[j] = aux;
    }
}
```

Algoritmo Heap Sort

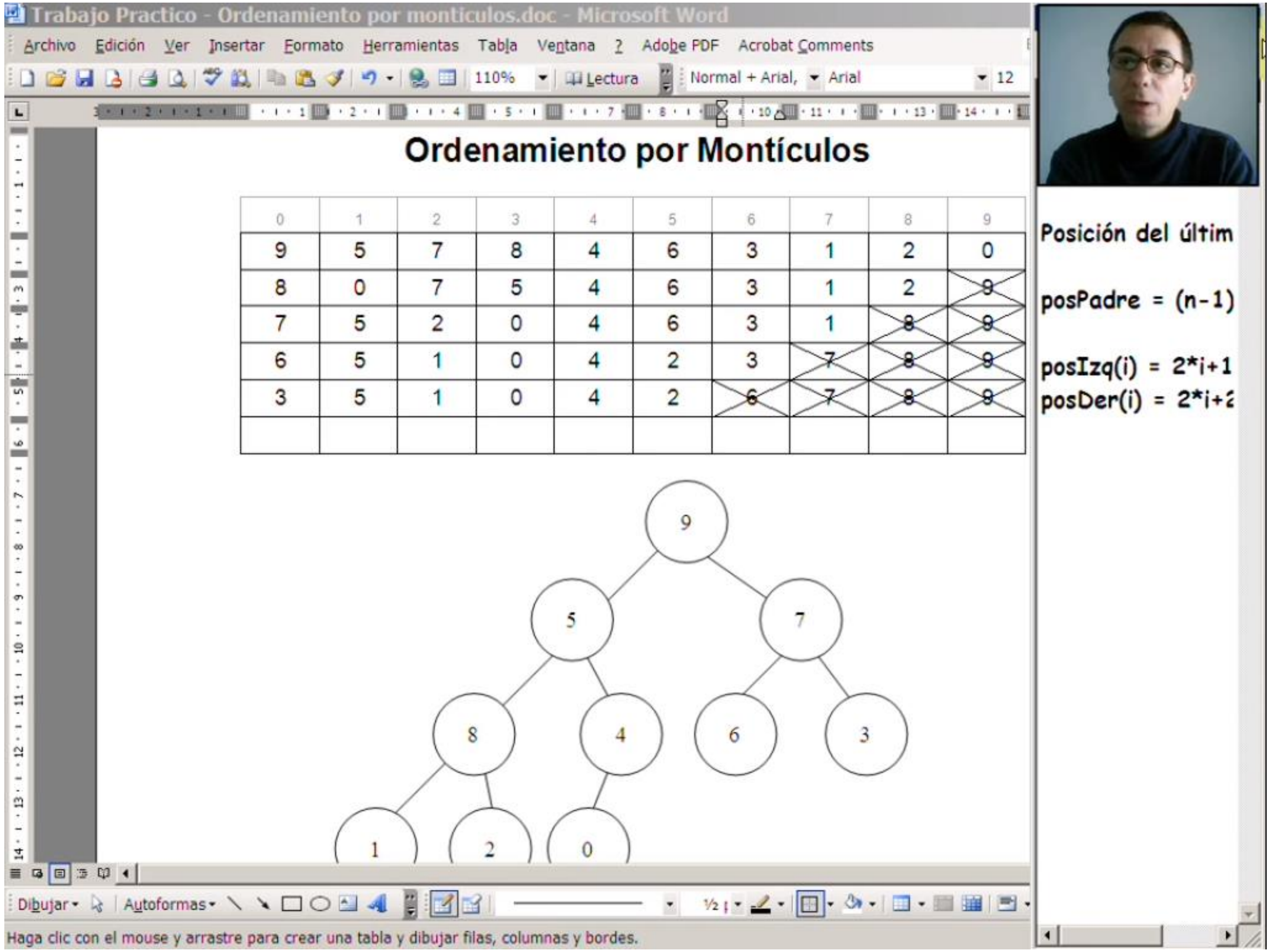
El proceso de ordenación por el método Heap Sort consta de dos partes:

1. Construir el montículo (insertar)
2. Eliminar repetidamente la raíz del montículo.

```
public static void heapsort(int a[]) {
    insertamonticulo(a);
    eliminamonticulo(a);
}
```

Algoritmo Heap Sort

https://www.youtube.com/watch?v=32jdzOmLsYQ



Tarea: Demostrar que la complejidad del Algoritmo Heapsort es O(nlogn)

Otros Algoritmos de Ordenamiento

https://www.youtube.com/watch?v=SHcPqUe2GZM

https://www.youtube.com/watch?v=VuXbEb5ywrU

https://www.youtube.com/watch?v=nu4gDuFabIM

https://www.youtube.com/watch?v=MtQL_ll5KhQ

SHELL SORT

BUCKET SORT

RADIX SORT

HEAP SORT

GeeksforGeeks

A computer science portal for geeks

¿Preguntas?

FIN