



TEMA 6

Algoritmos sobre Listas

V1.1

Manuel Pereira González



Agenda

- **Introducción**
- Algoritmos de Búsqueda
 - Búsqueda Secuencial
 - Búsqueda Binaria
 - Análisis de Eficiencia
- Algoritmos de Inserción
- Algoritmos de Ordenación
 - Métodos Directos
 - Métodos Avanzados
 - Medición Experimental de la Eficiencia
- Resumen



Introducción



- Importancia de Organizar la Información
- Ej: Buscar número de teléfono en la guía
 - Rápido, gracias a que está ordenado alfabéticamente
- En informática, igual que en la vida real, frecuentemente es necesario buscar información entre mucha disponible, para lo que también se ordena.
- Existen diversos algoritmos para ordenar una lista de elementos y buscar dentro de una lista ordenada

Agenda



- Introducción
- **Algoritmos de Búsqueda**
 - Búsqueda Secuencial
 - Búsqueda Binaria
 - Análisis de Eficiencia
- Algoritmos de Inserción
- Algoritmos de Ordenación
 - Métodos Directos
 - Métodos Avanzados
 - Medición Experimental de la Eficiencia
- Resumen



Algoritmos de Búsqueda: Búsqueda Secuencial



- No es necesario que los elementos estén ordenados dentro del array
- Poco eficiente:
 - n es el número de elementos del Array
 - Eficiencia: $O(n)$ – “O grande de n ”
 - El tiempo que tarda el algoritmo en ejecutarse es **aproximadamente proporcional a n** (al número de elementos)
- Consiste en ir mirando elemento a elemento del array hasta que se encuentre el indicado

Algoritmos de Búsqueda: Búsqueda Secuencial



```
/**
 * Busca un determinado elemento elem entre las posiciones from y to del vector
 * @param vector Array de números en el que buscar
 * @param elem Elemento a buscar
 * @param from Posición desde la que buscar
 * @param to Posición hasta la que buscar
 * @return Devuelve la posición en la que se ha encontrado, o -1 si no se encuentra
 */
public static int secuencial(int[] vector, int elem,
                             int from, int to) {
    for(int i=from; i<to; i++) {
        if(vector[i] == elem) {
            return i;
        }
    }
    return -1;
}
```

Algoritmos de Búsqueda:

Búsqueda Binaria



- El array debe estar ordenado
- Muy eficiente:
 - Eficiencia: $O(\log_2 n)$
 - El tiempo que tarda el algoritmo en ejecutarse es **aproximadamente proporcional al logaritmo en base 2 de n**
- Ejecución:
 - Se mira el elemento del centro de la lista
 - Si es el buscado, se devuelve
 - Si es mayor que el buscado, se repite la búsqueda sobre la primera mitad de la lista
 - Si es menor que el buscado, se repite la búsqueda sobre la segunda mitad de la lista
 - Se repite el algoritmo hasta que se encuentra el elemento o se llega a una lista de longitud cero (el elemento no existe).

Algoritmos de Búsqueda:

Búsqueda Binaria



```
/**
 * Busca un determinado elemento elem entre las posiciones from y to del vector
 * @param vector Array de números en el que buscar. El array debe estar ordenado
 * @param elem Elemento a buscar
 * @param from Posición desde la que buscar
 * @param to Posición hasta la que buscar
 * @return Devuelve la posición en la que se ha encontrado, o -1 si no se encuentra
 */
public static int binaria(int[] vector, int elem,
                          int from, int to) {
    for(;;) {
        int medio = (from + to) / 2;
        if (vector[medio] == elem) {
            return medio; // Encontrado
        }
        else if (from > to) {
            return -1; // No encontrado
        }
        else {
            // Divide
            if (vector[medio] < elem) { // En mitad superior
                from = medio + 1;
            }
            else { // En mitad inferior
                to = medio - 1;
            }
        }
    }
}
```

Algoritmos de Búsqueda:

Búsqueda Binaria



```
/**
 * Busca un determinado elemento elem en el vector
 * @param vector Array de números en el que buscar. El array debe estar ordenado
 * @param elem Elemento a buscar
 * @return Devuelve la posición en la que se ha encontrado, o -1 si no se encuentra
 */
public static int binaria(int[] vector, int elem) {
    return binaria(vector, elem, 0, vector.length - 1);
}

public static void main(String[] args) {
    int[] array = new int[] {2, 5, 18, 23, 54, 112};
    System.out.println("Buscando el 17: " + binaria(array, 17));
    System.out.println("Buscando el 18: " + binaria(array, 18));
    System.out.println("Buscando el 2: " + binaria(array, 2));
    System.out.println("Buscando el 1: " + binaria(array, 1));
    System.out.println("Buscando el 113: " + binaria(array, 113));
    System.out.println("Buscando el 112: " + binaria(array, 112));
}
```

```
C:\Practicas\Programacion>java BusquedaBinaria
Buscando el 17: -1
Buscando el 18: 2
Buscando el 2: 0
Buscando el 1: -1
Buscando el 113: -1
Buscando el 112: 5
C:\Practicas\Programacion>
```

Algoritmos de Búsqueda:

Análisis de Eficiencia



- Estudio de la eficiencia en la peor situación: cuando el elemento no se encuentra en la lista
- Búsqueda Secuencial: El número de comparaciones a realizar es n
- Búsqueda Binaria: En cada comparación se puede eliminar la mitad de los elementos del array.
 - Primera comparación: array de n elementos
 - Segunda comparación: array de $n/2$ elementos
 - Tercera comparación: array de $n/4$ elementos
 - Cuarta comparación: array de $n/8$ elementos
 - ...
 - C-ésima comparación: array de $n/2^C$ elementos

Algoritmos de Búsqueda: Análisis de Eficiencia



- Búsqueda Binaria: C-ésima comparación:
array de $n/2^C$ elementos
 - Para llegar a que el array tenga 1 sólo elemento: $n/2^C = 1$ por tanto $n = 2^C$ por lo que $C = \log_2(n)$

n	Número de Comparaciones	
	Búsqu. secuencial	Búsqu. binaria
10	10	4
100	100	7
1000	1000	10
10000	10000	14
100000	100000	17
1000000	1000000	20

Agenda



- Introducción
- Algoritmos de Búsqueda
 - Búsqueda Secuencial
 - Búsqueda Binaria
 - Análisis de Eficiencia
- **Algoritmos de Inserción**
- Algoritmos de Ordenación
 - Métodos Directos
 - Métodos Avanzados
 - Medición Experimental de la Eficiencia
- Resumen



Algoritmos de Inserción



- Inserción en un Array no ordenado
 - Necesario indicar la posición en la que se desea insertar el elemento (posición p)
 - Hay que “abrir hueco” en la posición p, desplazando una posición a la derecha los elementos desde p hasta el final.
- Inserción en un Array ordenado
 - No es necesario indicar la posición, el elemento a insertar ocupará la posición que le corresponde según su valor

Algoritmos de Inserción



- Inserción en un Array ordenado. Dos posibles situaciones:
 - 1) Los elementos **no pueden encontrarse repetidos en la lista**. Se busca la posición p en la que insertar (búsqueda binaria). Si el elemento a insertar ya se encuentra en la lista, se genera un código de error, en caso contrario se inserta en la posición p.
 - 2) Los elementos **pueden repetirse**. Se busca la posición p en la que insertar (búsqueda binaria) y se inserta el nuevo elemento en la posición p

Algoritmos de Inserción



```
/**
 * Inserta un determinado elemento elem en el vector ordenado
 * @param vector Array de números en el que insertar. El array debe estar ordenado
 * @param elem Elemento a insertar
 * @return Devuelve un nuevo array con el elemento insertado, o null si ya existía
 */

public static int[] insertarEnArrayOrdenado(int[] vector, int elem) {
    int from = 0;
    int to = vector.length;
    for(;;) {
        int medio = (from + to) / 2;
        if (medio < vector.length && vector[medio] == elem) {
            return null; // Ya existía, devuelve null
        }
        else if (from >= to) {
            // Encontrada la posición lo inserta
            int[] devolver = new int[vector.length + 1];
            for(int i=0; i<from; i++) {
                devolver[i] = vector[i];
            }
            devolver[from] = elem;
            for(int i=from+1; i<devolver.length; i++) {
                devolver[i] = vector[i-1];
            }
            return devolver;
        }
        else // Divide
        {
            if (vector[medio] < elem) { // En mitad superior
                from = medio + 1;
            }
            else { // En mitad inferior
                to = medio - 1;
            }
        }
    }
}
```

Agenda



- Introducción
- Algoritmos de Búsqueda
 - Búsqueda Secuencial
 - Búsqueda Binaria
 - Análisis de Eficiencia
- Algoritmos de Inserción
- **Algoritmos de Ordenación**
 - **Métodos Directos**
 - Métodos Avanzados
 - Medición Experimental de la Eficiencia
- Resumen



Métodos Directos: Burbuja o BubbleSort



- Basado en intercambio de pares adyacentes
- Se hacen varias pasadas sobre el array comparando todos los elementos adyacentes, intercambiándolos si no están ordenados
- En el peor caso se realizan $n-1$ pasadas sobre el array

Métodos Directos: Burbuja o BubbleSort



```
public class Burbuja {  
    public static void main(String[] args) {  
        int[] vector = new int[] {54, 61, 1, 23, 14, 112, 4};  
        burbuja(vector);  
        for(int i=0; i<vector.length; i++) {  
            System.out.print(vector[i] + " ");  
        }  
    }  
  
    public static void burbuja(int[] vector) {  
        int num = vector.length;  
        for(int i=1; i<num; i++) {  
            for(int j=0; j<num-i; j++) {  
                if(vector[j] > vector[j+1]) {  
                    int aux = vector[j];  
                    vector[j] = vector[j+1];  
                    vector[j+1] = aux;  
                }  
            }  
        }  
    }  
}
```

```
Command Prompt  
C:\Practicas\Programacion>java Burbuja  
1, 4, 14, 23, 54, 61, 112.  
C:\Practicas\Programacion>
```

Métodos Directos: Inserción Directa



- Insertar cada elemento, comenzando desde el segundo y hasta el final, en el lugar que le corresponde en la secuencia ordenada que se va formando a su izquierda
 - En la primera pasada se selecciona el segundo elemento y se inserta en la secuencia ordenada formada por el primer elemento
 - En la segunda pasada se inserta el tercer elemento en la secuencia ordenada formada por los dos primeros
 - ...
- Consiste en n-1 inserciones en una lista ordenada

Métodos Directos: Inserción Directa



```
public class InsercionDirecta {  
    public static void main(String[] args) {  
        int[] vector = new int[] {54, 61, 1, 23, 14, 112, 4};  
        insercionDirecta(vector);  
        for(int i=0; i<vector.length; i++) {  
            System.out.print(vector[i] + " ");  
        }  
    }  
  
    public static void insercionDirecta(int[] vector) {  
        for(int i=1; i<vector.length; i++) {  
            int aux = vector[i];  
            int j = i - 1;  
            while(j >= 0 && aux < vector[j]) {  
                vector[j+1] = vector[j];  
                j--;  
            }  
            vector[j+1] = aux;  
        }  
    }  
}
```

Métodos Directos: Selección Directa



- Se selecciona de la lista completa el elemento con menor valor y se intercambia con el primero (el primer elemento queda definitivamente ordenado).
- A continuación, se busca el elemento de menor valor de la sublista comprendida entre el segundo y el último, y se intercambia con el segundo elemento
- Así sucesivamente...

Métodos Directos: Selección Directa



```
public class SeleccionDirecta {  
    public static void main(String[] args) {  
        int[] vector = new int[] {54, 61, 1, 23, 14, 112, 4};  
        seleccionDirecta(vector);  
        for(int i=0; i<vector.length; i++) {  
            System.out.print(vector[i] + " ");  
        }  
    }  
  
    public static void seleccionDirecta(int[] vector) {  
        for(int i=0; i<vector.length-1; i++) {  
            int menor = vector[i];  
            int pos = i;  
            for(int j=i+1; j<vector.length; j++) {  
                if(vector[j] < menor) {  
                    menor = vector[j];  
                    pos = j;  
                }  
            }  
            vector[pos] = vector[i];  
            vector[i] = menor;  
        }  
    }  
}
```

```
Command Prompt  
C:\Practicas\Programacion>java SeleccionDirecta  
1, 4, 14, 23, 54, 61, 112.  
C:\Practicas\Programacion>
```

Agenda



- Introducción
- Algoritmos de Búsqueda
 - Búsqueda Secuencial
 - Búsqueda Binaria
 - Análisis de Eficiencia
- Algoritmos de Inserción
- Algoritmos de Ordenación
 - Métodos Directos
 - **Métodos Avanzados**
 - Medición Experimental de la Eficiencia
- Resumen



Métodos Avanzados: Ordenación Rápida o QuickSort



- El mejor método para ordenar arrays conocido hasta el momento.
- En la mayoría de los casos su eficiencia es de orden $O(n \log(n))$
- En los peores casos, su eficiencia puede ser de orden $O(n^2)$, aunque es muy improbable
- Su eficiencia depende de la elección del pivote

Métodos Avanzados: Ordenación Rápida o QuickSort



- Algoritmo:
 - Se elige un elemento del array (**pivote**), y se dejan a su izquierda todos los valores menores o iguales que él, y a la derecha los mayores
 - Se realiza la misma operación con las partes resultantes de la izquierda y la derecha del pivote
 - Así sucesivamente, hasta que las partes tengan tamaño 1 (array completamente ordenado)

Métodos Avanzados: Ordenación Rápida o QuickSort



```
private static void quickSort( int vector[], int izq, int der) {
    int pivote = vector[(izq + der) / 2];
    int i = izq, j = der;
    do {
        while(vector[i] < pivote) {
            i++;
        }
        while(vector[j] > pivote) {
            j--;
        }
        if(i <= j) {
            int aux = vector[i];
            vector[i] = vector[j];
            vector[j] = aux;
            i++;
            j--;
        }
    } while (i <= j);
    if(izq < j) {
        quickSort(vector, izq, j);
    }
    if(i < der) {
        quickSort(vector, i, der);
    }
}

public static void main(String[] args) {
    int[] vector = new int[] {54, 61, 1, 23, 14, 112, 4};
    quickSort(vector);
    for(int i=0; i<vector.length; i++) {
        System.out.print(vector[i] + " ");
    }
}

public static void quickSort(int[] vector){
    quickSort(vector, 0, vector.length - 1 );
}
```

```
C:\Practicas\Programacion>java QuickSort
1. 4. 14. 23. 54. 61. 112.
```

Métodos Avanzados: Método del Montículo o HeapSort

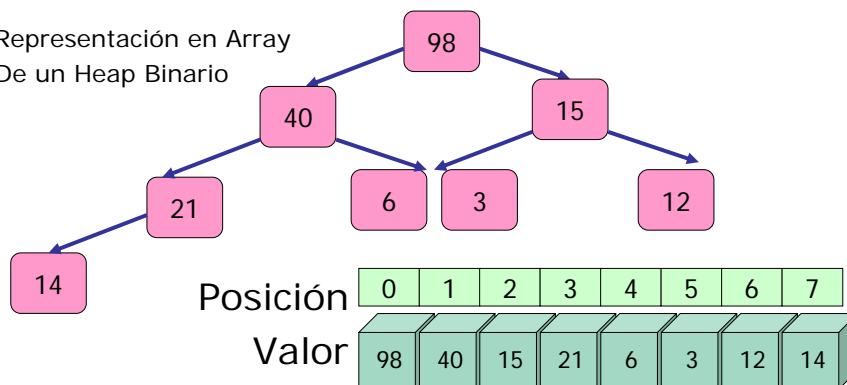


- Un **montículo** (heap) es una estructura de Árbol con información perteneciente a un conjunto ordenado.
- Característica de un montículo: cada nodo tiene un valor mayor que el de todos sus nodos hijos -> **El nodo raíz contiene el mayor elemento.**
- Algoritmos HeapSort:
 - 1) Construir un montículo
 - 2) Sacar el nodo raíz (mayor)
 - 3) Reconstruir el montículo con los elementos restantes
 - 4) Volver al paso 2) mientras queden elementos en el montículo

Métodos Avanzados: Método del Montículo o HeapSort



Representación en Array
De un Heap Binario



Posición del Hijo Derecho = $2 * (\text{Posición del Padre} + 1)$

Posición del Hijo Izquierdo = $(2 * (\text{Posición del padre Padre} + 1)) - 1$

Métodos Avanzados: Método del Montículo o HeapSort



- **Fase 1:** Construcción inicial del montículo
 - Se construye el montículo inicial a partir del array original
- **Fase 2:** Ordenación. Repetir hasta que sólo quede un elemento en el montículo:
 - A) Se intercambia la raíz con el último elemento del montículo. Este último elemento queda definitivamente ordenado y ya no pertenece al montículo.
 - B) Se restaura el montículo haciendo que se “hunda” el primer elemento hasta la posición que le corresponda. La raíz entonces vuelve a ser el mayor elemento, se vuelve al punto A).

Métodos Avanzados: Método del Montículo o HeapSort



```
public static void main(String[] args) {
    int[] vector = new int[] {54, 61, 1, 23, 14, 112, 4};
    heapSort(vector);
    for(int i=0; i<vector.length; i++) {
        System.out.print(vector[i] + " ");
    }
}

public static void heapSort(int[] vector)
{
    int izq = vector.length;
    for (int v=izq/2-1; v>=0; v--) {
        criba (vector, izq, v);
    }
    while (izq>1)
    {
        izq--;
        int aux=vector[0];
        vector[0]=vector[izq];
        vector[izq]=aux;
        criba (vector, izq, 0);
    }
}

private static void criba(int[] vector,
    int izq, int der)
{
    int w=2*der+1;
    while (w<izq)
    {
        if (w+1<izq) {
            if (vector[w+1]>vector[w]) {
                w++;
            }
        }
        if (vector[der]>vector[w]) {
            return;
        }
        int aux=vector[der];
        vector[der]=vector[w];
        vector[w]=aux;
        der=w;
        w=2*der+1;
    }
}
```

Command Prompt

```
C:\Practicas\Programacion>java HeapSort
1. 4. 14. 23. 54. 61. 112.
C:\Practicas\Programacion>
```

Agenda



- Introducción
- Algoritmos de Búsqueda
 - Búsqueda Secuencial
 - Búsqueda Binaria
 - Análisis de Eficiencia
- Algoritmos de Inserción
- Algoritmos de Ordenación
 - Métodos Directos
 - Métodos Avanzados
 - **Medición Experimental de la Eficiencia**
- Resumen



Medición Experimental de la Eficiencia



- Tiempos (en milliseg) que se han tardado experimentalmente en ordenar arrays generados aleatoriamente según todos los algoritmos estudiados

	10.000	15.000	20.000	25.000	30.000	35.000	40.000
Burbuja	41.924	116.538	170.098	266.772	391.321	534.290	697.952
Inserción	13.311	31.966	55.043	86.794	127.112	197.123	272.311
Selección	16.962	39.192	69.083	108.254	157.849	226.166	324.750
QuickSort	76	123	163	212	267	449	360
HeapSort	176	290	389	493	692	720	844

Agenda



- Introducción
- Algoritmos de Búsqueda
 - Búsqueda Secuencial
 - Búsqueda Binaria
 - Análisis de Eficiencia
- Algoritmos de Inserción
- Algoritmos de Ordenación
 - Métodos Directos
 - Métodos Avanzados
 - Medición Experimental de la Eficiencia
- **Resumen**



Resumen



- | | |
|--|--|
| <ul style="list-style-type: none">▪ Introducción▪ Algoritmos de Búsqueda<ul style="list-style-type: none">▪ Búsqueda Secuencial▪ Búsqueda Binaria▪ Análisis de Eficiencia▪ Algoritmos de Inserción▪ Algoritmos de Ordenación<ul style="list-style-type: none">▪ Métodos Directos<ul style="list-style-type: none">▪ BubbleSort<ul style="list-style-type: none">▪ Intercambios adyacentes▪ Inserción directa<ul style="list-style-type: none">▪ Inserciones sucesivas▪ Selección directa<ul style="list-style-type: none">▪ Buscar el menor de una sublista | <ul style="list-style-type: none">▪ Métodos Avanzados<ul style="list-style-type: none">▪ QuickSort<ul style="list-style-type: none">▪ Más rápido▪ Divide y Vencerás▪ HeapSort<ul style="list-style-type: none">▪ Montículo▪ Dos fases▪ Medición Experimental de la Eficiencia<ul style="list-style-type: none">▪ $O(n^2)$▪ $O(n \log(n))$ |
|--|--|

Resumen: Para más información



- Algoritmos de ordenación en Java:
 - <http://www.cs.ubc.ca/spider/harrison/Java/sorting-demo.html>
- Algoritmos de ordenación animados
 - <http://www.csc.depauw.edu/~bhoward/courses/0304Fall/csc222/sort/>
 - <http://www2.hig.no/~algmet/animate.html>
- Heap Sort Animated
 - <http://www2.hawaii.edu/~copley/665/HSApplet.html>