Análisis de algoritmos

Algoritmos de Ordenación

Índice

- ¿ Por qué es importante la ordenación ?
- Algunos ejemplos
  - InsertionSort
  - SelectionSort
  - MergeSort
  - QuickSort
  - BubbleSort (Lab)
- Para cada algoritmo
  - En que consiste
  - Ejemplos de implementación
  - Eficiencia. Casos extremos

2

¿ Por qué es importante ?

- Facilita la búsqueda
- La ***información devuelta por un cómputo*** suele hacerse de forma ordenada para facilitar su manejo. Ej: Palabras de un diccionario, ficheros en un directorio, catalogo en una biblioteca, etc.
- Una ordenación de datos inicial puede ***facilitar el rendimiento de un algoritmo***. Ej. Encontrar valores repetidos
- Muchos de los cómputos invocan internamente un método de ordenación y el ***coste del método de ordenación utilizado determina el coste global***.

Problema inicial

- Partimos de una estructura de datos lineal que contiene un determinado número de elementos (N)
- Esos elementos son ordenables siguiendo algún criterio (es decir, son comparables).
- Para comprobar si un algoritmo de ordenación es correcto y calcular su eficiencia es necesario estudiar los casos más favorables y desfavorables y casos como:
  - Si la estructura está ordenada (en orden creciente o decreciente) antes de empezar a ejecutar el algoritmo
  - si existen elementos repetidos
  - etc.

4

InsertionSort
Ordenación por inserción

1. Partimos de un array de elementos, a[]
2. Suponemos que el primer elemento a[0] está ordenado
3. Seleccionamos el siguiente elemento, a[1] y lo guardamos en una variable temporal, tmp
4. Comparamos uno por uno con los elementos previamente ordenados para localizar su sitio correcto, desplazándolos a la derecha en el caso de que sean mayores
5. Lo ubicamos en su posición correcta y
6. Repetimos el proceso con el siguiente elemento suponiendo ahora que a[0] y a[1] están ordenados y añadimos a[2].
   a) siempre suponemos que la matriz a[0]....a[p-1] está ordenada
   b) cogemos el elemento a[p] y lo ubicamos correctamente
   c) cogemos el elemento a[p+1]
   d) repetimos el proceso hasta añadir el último elemento

5

InsertionSort
Ordenación por inserción. Ejemplo

- Partimos del siguiente array **a**

8	5	9	2	6	3

- Una simple ejecución del algoritmo sobre el array a (la parte coloreada está ordenada)

Posiciones del array a	**0**	**1**	**2**	**3**	**4**	**5**
Estado inicial	8	5	9	2	6	3
Después de ordenar a[0..1]	5	8	9	2	6	3
Después de ordenar a[0..2]	5	8	9	2	6	3
Después de ordenar a[0..3]	2	5	8	9	6	3
Después de ordenar a[0..4]	2	5	6	8	9	3
Después de ordenar a[0..5]	2	3	5	6	8	9

6

InsertionSort

Ordenación por inserción. Ejemplo

- Partimos del siguiente array **a**

8	5	9	2	6	3

- Una simple ejecución del algoritmo sobre el array *a*, viendolo más detenidamente

(la parte coloreada está ordenada y el color gris indica la posición dónde se ha insertado el nuevo elemento)

Posiciones del array a	**0**	**1**	**2**	**3**	**4**	**5**
Estado inicial	8	**5**				
Después de ordenar a[0..1]	5	8	**9**			
Después de ordenar a[0..2]	5	8	9	**2**		
Después de ordenar a[0..3]	2	5	8	9	**6**	
Después de ordenar a[0..4]	2	5	6	8	9	**3**
Después de ordenar a[0..5]	2	3	5	6	8	9

7

InsertionSort

Implementación

```
public static void insertionSort (int a[]) {
    for(int p=1;p<a.length;p++) {// recorre desde p hasta fin del array
            int tmp=a[p];
            int j;
            for(j=p;j>0 && tmp < a[j-1]; j--) { // recorre desde p hasta principio del array
                    a[j]=a[j-1]; // desplazo hacia la derecha los elementos del array mayores que tmp
            }
            a[j]=tmp; // inserto tmp en la posición en la que se para j (cuando dejo de desplazar elementos)
    }
}
```

8

InsertionSort
Complejidad. Casos extremos

- Cuando el array está ordenado en orden inverso
  - Caso peor
  - El bucle interno se ejecuta p+1 veces por cada valor de p:
    - 1+2+3+ .+n
  - $t_{ejecución}$ sería $O(n^2)$
- Cuando el array de entrada está ordenado
  - Caso mejor
  - El bucle interno nunca se ejecuta y $t_{ejecución}$ sería $O(n)$
- En problemas de ordenación el $t_{ejecución}$ depende no sólo del tamaño de los datos, sino también del orden específico de estos

9

SelectionSort
Ordenación por selección

1. Partimos de un array de elementos, a[]
2. Recorremos todo el array desde la posición 0 hasta el final seleccionando el elemento mas pequeño.
3. Intercambiamos el elemento seleccionado con el que esta en la posición 0.
4. Repetimos el proceso, seleccionando el elemento mas pequeño desde la posición 1 hasta el final e intercambiándolo con el elemento de la posición 1.
5. El proceso se repite:
   - Seleccionando subarrays mas pequeños desde la posición i hasta el final
   - intercambiando el elemento mas pequeño, con el de la posición i-esima.
   - Repetimos el proceso hasta que el subarray conste de un sólo elemento.

10

SelectionSort
Ordenación por inserción. Ejemplo

- Partimos del siguiente array **a**

8	5	9	2	6	3

- Una simple ejecución del algoritmo sobre el array a

(la parte coloreada está ordenada, el color gris indica la posición dónde se encuentra el mínimo, el cuál se intercambia con el primer elemento de la parte desordenada, marcado en negrita)

Posiciones del array a	**0**	**1**	**2**	**3**	**4**	**5**
Estado inicial	**8**	5	9	2	6	3
Después de ordenar a[0..0]	2	**5**	9	8	6	3
Después de ordenar a[0..1]	2	3	**9**	8	6	5
Después de ordenar a[0..2]	2	3	5	**8**	6	9
Después de ordenar a[0..3]	2	3	5	6	**8**	9
Después de ordenar a[0..4]	2	3	5	6	8	9

11

SelectionSort
Implementación

```
public static void selectionSort (int a[]) {
    for(int i=0; i < a.length-1; i++) {     // recorre todo el array
        int current = a[i]; // selecciona un elemento
        int k=i; // la variable k indica, la posición del minimo actual
        for(int j=i+1; j < a.length; j++) { // recorre desde la posición siguiente al seccionado
                                            // hasta el final del array
            if (current >a[j]){ // compara el seleccionado con los elementos del subarray restante
                k=j;            // se queda con la posición del mas pequeño
                current=a[j];   // y su contenido
            }
        }
        a[k]=a[i]; // guarda el elemento seleccionado en la posición donde estaba el mas pequeño
        a[i]=current; // guarda el elemento mas pequeño en la posición donde estaba el seleccionado
    }
}
```

SelectionSort
Complejidad. Casos extremos

- Este algoritmo funciona similar al *insertionSort*
- Existen dos ciclos anidados, por tanto

$t_{ejecución}$ será $O(n^2)$

13

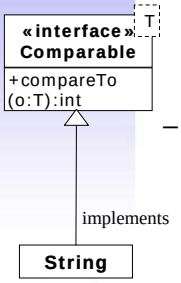
Problema

- Cómo podríamos ordenar una lista
  - de números reales?
  - de cadena de caracteres?
  - de personas?
    - por nombre?
    - o por edad?

14

Problema
Ordenando objetos de cualquier tipo

- Cómo se comparan o ordenan los objetos en Java?
  - La forma natural es implementando el interfaz *Comparable (cuando un objeto es menor, igual o mayor que otro)*




```
interface Comparable<T>{
   public int compareTo(T o);
}
```

  - *Además de la clase `String`, todas las clases wrapper (envoltorio. p.ej. `Integer`) lo implementan. Por jemplo,*

```
class String implements Comparable<String>{
   public int compareTo(String o);
}
```

  - *Uso:*

```
String s1, s2;
s1.compareTo(s2)→  <0 si s1 < s2
                    0 si s1 = s2
                   >0 si s1 > s2
```

15

Ejercicios

1. Reescribe el algoritmo **InsertionSort** para ordenar un array de objetos de tipo `String` (de menor a mayor)

```
class String implements Comparable<String>{
    public int compareTo(String o);
}
String s1, s2;
s1.compareTo(s2)→  <0 si s1 < s2
                    0 si s1 = s2
                   >0 si s1 > s2
```

2. Reescribe el algoritmo **SelectionSort** para ordenar un array de objetos de tipo `Persona`. (por nombre)

```
class Persona implements Comparable<Persona>{
    String nombre;
    int edad;
   public int compareTo(Persona o)
}
```

16

Ejercicio 1

(no es el definitivo)

```
public static void insertionSort (String a[]) {
    for(int p=1; p < a.length; p++) {// recorre desde p hasta fin del array
        String tmp=a[p];
         int j;
         for(j=p; j>0 & tmp.compareTo(a[j-1])<0; j--) { // recorre desde p hasta el inicio del array
             a[j]=a[j-1]; // desplazo hacia la drcha lo elementos del vector mayores que tmp
          }
          a[j]=tmp; // inserto tmp en la posición en la que se para j (cuando dejamos de desplazar elementos)
    }
}
```

Lección aprendida:

1. Utilizar el tipo de dato adecuado
2. Implementar el método de comparación (i.e. *compareTo*)
3. Definir cual es el criterio de ordenación

17

Ejercicio 2

(no es el definitivo)

```
public static void selectionSort (Persona a[]) {
    for(int i=0; i < a.length-1; i++) {     // recorre todo el array
      Persona current = a[i]; // selecciona un elemento
      int k=i;
      for(int j=i+1; j < a.length; j++) { // recorre desde la posición siguiente al seccionado
                                          // hasta el final del array
          if (current.compareTo(a[j]) < 0) { // compara el seleccionado con los elementos del
                                             // subarray restante
              k=j;              // se queda con la posición del mas pequeño
              current=a[j];     // y su contenido
          }
       }
      a[k]=a[i]; // guarda el elemento seleccionado en el hueco donde estaba el mas pequeño
      a[i]=current; // guarda el elemento mas pequeño en el hueco donde estaba el seleccionado
    }
}
```

Lección aprendida

1. Utilizar un tipo que sea comparable
2. El interfaz *Comparable* garantiza la implementación del método de comparación (i.e. ***compareTo***)
   a) Cuando un objeto es *menor*, *mayor* o *igual* que otro
3. Definir cual es el criterio de ordenación
   a) Ascendente o descendente

(ver http://tim.oreilly.com/lpt/a/3286)

19

Algoritmo genérico

T indica cualquier tipo, pero ésta tiene que ser una subclase de Comparable<T>

```
public static <T extends Comparable<T>>
void insertionSort (T a[]) {
    for(int p=1; p < a.length; p++) {// recorre desde p hasta fin del array
        T tmp=a[p];
         // recorre desde p hasta el inicio del array
        for(int j=p; j>0 && (tmp.compareTo(a[j-1])<0); j--) {
          a[j]=a[j-1]; // desplazo hacia la derecha los elementos del vector mayores que tmp
        }
        a[j]=tmp; // inserto tmp en la posición en la que se para j (cuando dejamos de desplazar elementos)
    }
}
```

```
public static void main (String[] args) {
    String [] nombres = {"pepito", "ana", "carlos"};
     insertionSort(nombres); //el tipo T se infiere del tipo del array
     for (String n: nombres)  // visualiza en pantalla los nombres: ana, carlos, pepito
          System.out.println(n);
}
```

Ejercicio 2

implementar el interfaz Comparable (with generics)

```
public class Persona implements Comparable<Persona> {
    private String nombre;
    private int edad;

    public Persona(String nombre, int edad) { …}

    public int compareTo(Persona o) {
            return this.getNombre().toLowerCase().compareTo(
                                                    o.getNombre().toLowerCase());
    }
   …
}
```

```
public static void main (String[] args) {
     Persona [] personas = {new Persona("pepito",10), new Persona("ana",15),
                            new Perona("carlos",5)};
     insertionSort(personas);
     for (Persona p: personas)  // visualiza en pantalla las personas: ana, carlos, pepito
          System.out.println(p);
}
```

21

Ejercicios (cont.)

3. Reescribe el algoritmo **InsertionSort** para que sea posible:
   a) ordenar un array de objetos (p.ej *Persona*) por nombre
   b) y otro array con los mismos objetos por edad

```
class Persona {
     private String nombre;
     private int edad;
}
```

***Problema**: el interfaz `Comparable` no puede comparar dos objetos por distintos criterios en una misma implementación*

***Solución**: Definir cada criterio implementando el*

```
interface Comparator<T>{
   public int compare(T o1, T o2)
}
```

22

Ejercicio 3

(después del Java 5.0, con generics)

```
private class PersonaPorNombre implements Comparator<Persona> {
    public int compare(Persona o1, Persona o2) {
        return o1.getNombre().toLowerCase().compareTo(
                                        o2.getNombre().toLowerCase());
    }
}
```

```
private class PersonaPorEdad implements Comparator<Persona> {
    public int compare(Persona o1, Persona o2) {
        if (o1.getEdad() < o2.getEdad())
            return -1;
        if (o1.getEdad() == o2.getEdad())
            return 0;
        return 1; //o1.getEdad() > o2.getEdad()
    }
}
```

23

Ejercicio 3

(después del Java 5.0, con generics)

```
public class Persona {
    …
    public static Comparator<Persona> compPersonaPorNombre;
    public static Comparator<Persona> compPersonaPorEdad;
    public Persona(String nombre, int edad) {
            …
            compPersonaPorNombre = new PersonaPorNombre();
            compPersonaPorEdad = new PersonaPorEdad()
    }

    private class PersonaPorNombre implements Comparator<Persona> {…}
    private class PersonaPorEdad implements Comparator<Persona> {…}
}
```

24

Ejercicio 3 (otra alternativa)

implementar directamente el interfaz

```
public class Persona {
   …
    public static Comparator<Persona> compPersonaPorNombre;
    public static Comparator<Persona> compPersonaPorEdad;
    public Persona(String nombre, int edad) {
          …
          compPersonaPorNombre = new Comparator<Persona> () {
              public int compare(Persona o1, Persona o2) {
                    return o1.getNombre().toLowerCase().compareTo(
                                                o2.getNombre().toLowerCase());
              }}
          compPersonaPorEdad = new Comparator<Persona> () {
              public int compare(Persona o1, Persona o2) { …  }
    }
}
```

Algoritmo genérico

(después del Java 5.0, con generics)

```
public static <AnyType>
void insertionSort (AnyType a[], Comparator<AnyType> comp) {
    for(int p=1; p < a.length; p++) {  // recorre desde p hasta fin del array
        AnyType tmp=a[p];
        for(int j=p; j>0 && (comp.compare(tmp, a[j-1])<0); j--) {  // recorre desde p hasta el inicio
            a[j]=a[j-1]; // desplazo hacia la derecha los elementos del vector mayores que tmp
        }
        a[j]=tmp; // inserto tmp en la posición en la que se para j (cuando dejamos de desplazar elementos)
    }
}
```

```
public static void main (String[] args) {
    Persona [] personas = {new Persona("pepito",10), new Persona("ana",15),
                           new Perona("carlos",5)};
    insertionSort(personas, Persona.compPersonaPorNombre);
    for (Persona p: personas)  // visualiza en pantalla las personas: ana, carlos, pepito
        System.out.println(p);
}
```

26

Otra alternativa:

proveer los dos interfaces de comparación reutilizando la implementación

```
public class Persona implements Comparable<Persona> {
   …
   public static Comparator<Persona> defaultComparator;
   public static Comparator<Persona> compPersonaPorEdad;
   public Persona(String nombre, int edad) {
          …
           defaultComparator = new Comparator<Persona>() {
              public int compare(Persona o1, Persona o2) {
                    return o1.compareTo(o2);
              }}
          compPersonaPorEdad = new Comparator<Persona>() {
              public int compare(Persona o1, Persona o2) { …  }
   }
    public int compareTo(Persona o) { //comparar los nombres …}
}
```

Técnicas: Divide y Venceras

¿En qué consiste ?

- ***Dividir***
  - **Descomponer** el problema a resolver en un nº de subproblemas mas pequeños hasta llegar al caso base
- ***Vencer***
  - **Resolver** sucesiva e independientemente todos los **subproblemas**
  - **Combinar las soluciones** obtenidas de esta forma, para obtener la solución al problema original.

33

Técnicas: Divide y Venceras

¿Cuándo debe utilizarse ?

- Decimos que un algoritmo recursivo utiliza esta técnica cuando:
  - Contiene **al menos dos llamadas** recursivas (es lo que se denomina recursión en cascada)
  - Ambas llamadas son **disjuntas** (sin superposiciones) para no hacer cálculos redundantes.

34

MergeSort

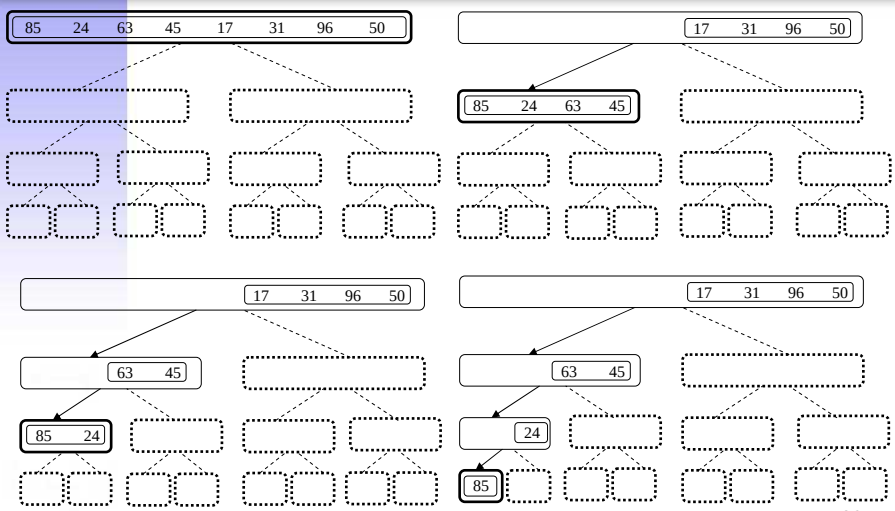
Ordenación por mezcla de vectores ordenados

- Utiliza la recursión para conseguir un algortimo subcuadrático, es decir de orden inferior a $O(n^2)$
- Partimos de un array de elementos a[]
  - Si el número de elementos es 0 o 1, termina
  - Si no es así, dividimos el array por la mitad
  - Ordenamos recursivamente los dos arrays resultantes
  - Mezclamos las dos mitades ordenadas en un array ordenado
- Eficiencia: $t_{ejecución}$ $O(N \log N)$ (demostrable)

35

MergeSort

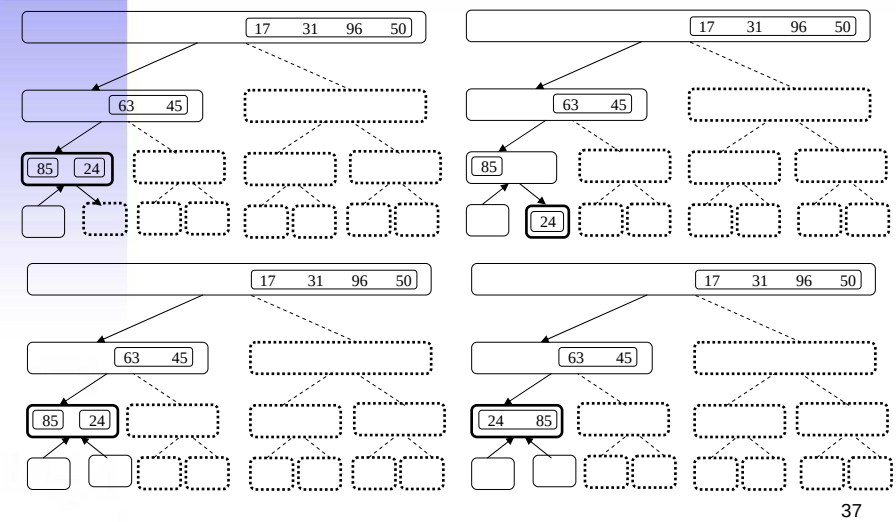
Ordenación por mezcla de vectores ordenados



36

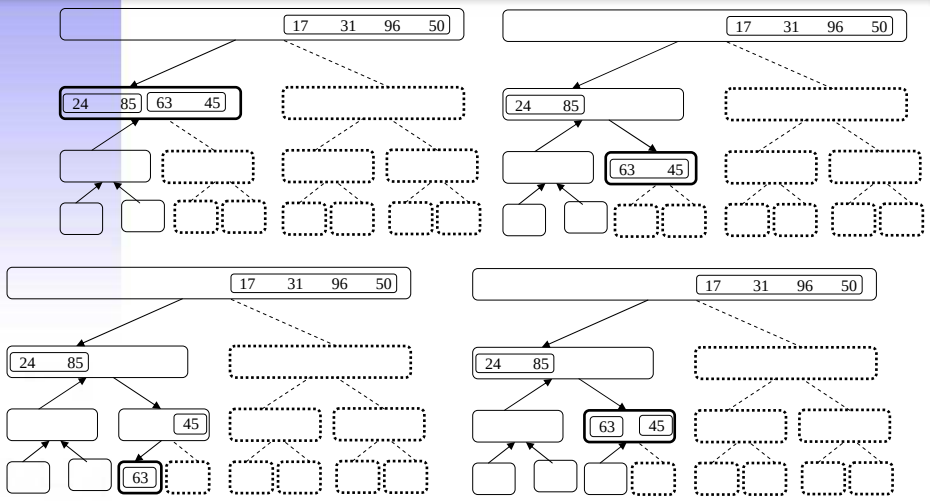
MergeSort

Ordenación por mezcla de vectores ordenados



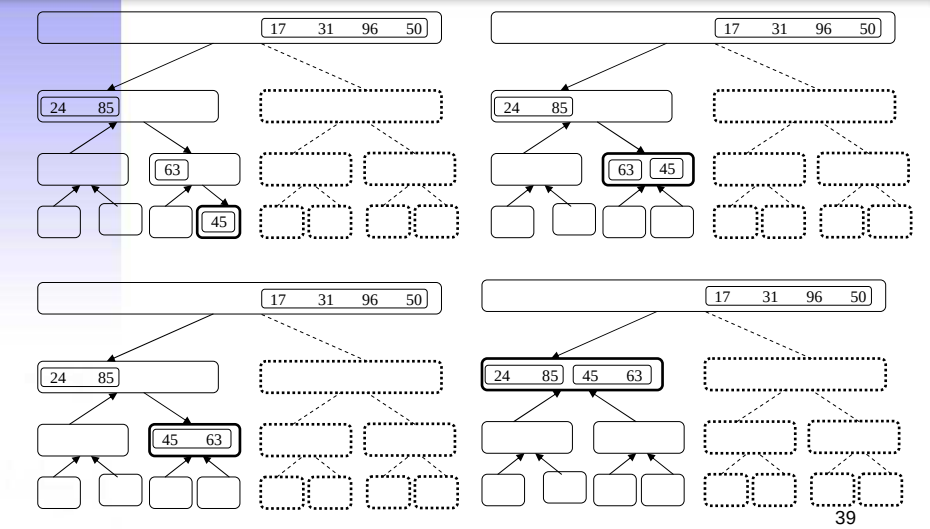
37

MergeSort
Ordenación por mezcla de vectores ordenados



38

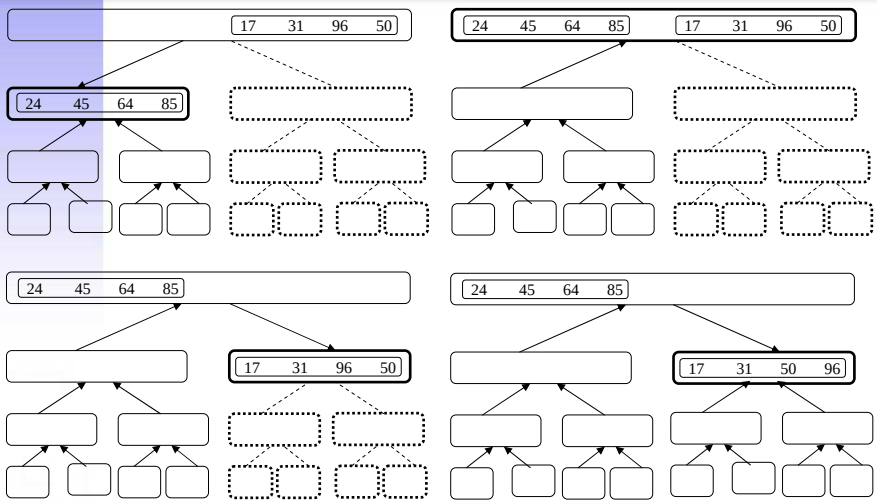
MergeSort
Ordenación por mezcla de vectores ordenados



39

MergeSort

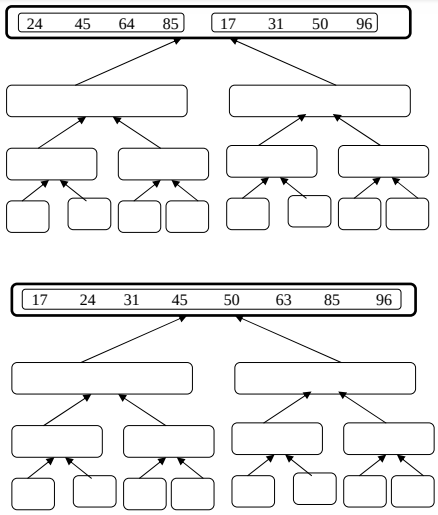
Ordenación por mezcla de vectores ordenados



40

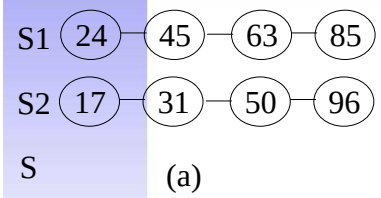
MergeSort

Ordenación por mezcla de vectores ordenados

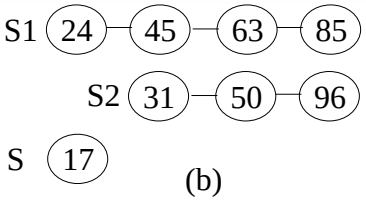


41

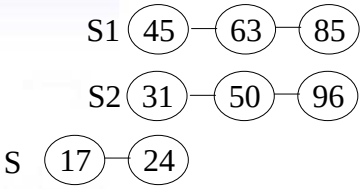
MergeSort
Ordenación por mezcla de vectores ordenados



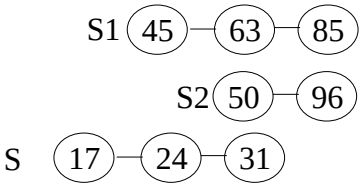
(a)



(b)



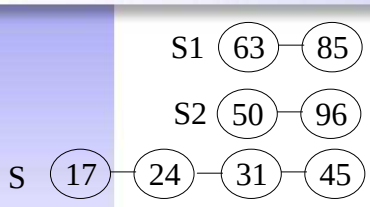
(c)



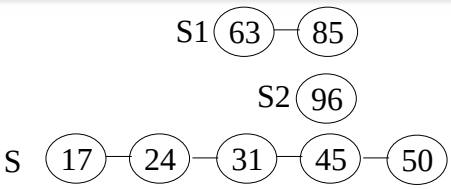
(d)

42

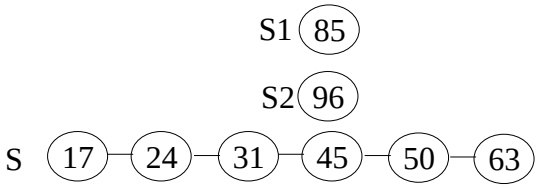
MergeSort
Ordenación por mezcla de vectores ordenados



(e)



(f)

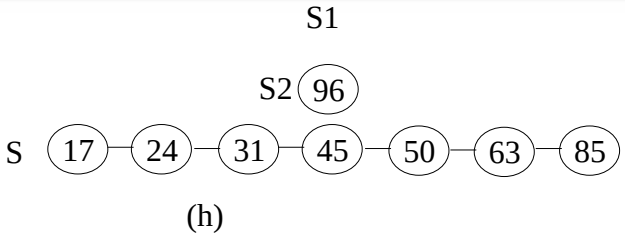


(g)

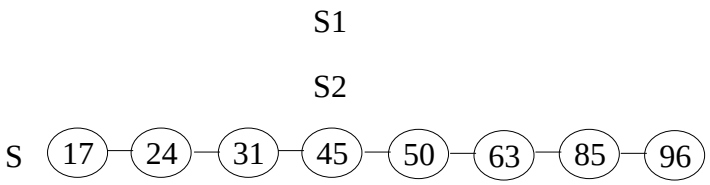
43

MergeSort

Ordenación por mezcla de vectores ordenados



(h)



(i)

44

MergeSort

Implementación recursiva

```
public static void mergeSort (int a[]) {
    int tmpArray[]=new int[a.length];
    mergeSort(a, tmpArray, 0, a.length-1);
}
```

```
public static void mergeSort (int a[], int tmpArray[], int left, int right) {
    if (left<right) {
            int center=(left+right)/2;
            mergeSort(a, tmpArray, left, center);
            mergeSort(a, tmpArray, center+1, right);
            merge(a, tmpArray, left, center+1, right);
    }
}
```

MergeSort
Implementación recursiva

```
private static void merge(int a[], int tmpArray[], int leftPos, int rightPos, int rightEnd) {
    int leftEnd=rightPos-1;
    int tmpPos=leftPos;
    int numElements=rightEnd-leftPos+1;
    while (leftPos<=leftEnd && rightPos<=rightEnd) {
        if (a[leftPos]< (a[rightPos] ) )
            tmpArray[tmpPos++]=a[leftPos++];
        else
            tmpArray[tmpPos++]=a[rightPos++];
    } //end first while
    while (leftPos<=leftEnd)  {  // Copy the rest of first half (middle)
        tmpArray[tmpPos++]=a[leftPos++];
    }
    while (rightPos<=rightEnd) { // Copy the rest of second half
        tmpArray[tmpPos++]=a[rightPos++];
    }
    for (int i=0; i<numElements;i++,rightEnd--) { Copy tmpArray in a
        a[rightEnd]=tmpArray[rightEnd];
    }
}
```

MergeSort
Complejidad

- Aunque $t_{ejecución}$ O(N logN) no se suele utilizar para ordenaciones internas (que ejecutan en memoria) porque:
  - La ordenación de un array **utiliza un espacio adicional en memoria**
  - El trabajo que supone copiar información entre el array de partida y el auxiliar ralentiza la ordenación
- Para ordenaciones internas (en memoria) se suele utilizar el QuickSort
- Para ordenaciones externas (utilizando espacio en disco) se suele utilizar el MergeSort.

Mergesort: referencias

- http://en.wikipedia.org/wiki/Mergesort
- http://es.wikipedia.org/wiki/Mergesort

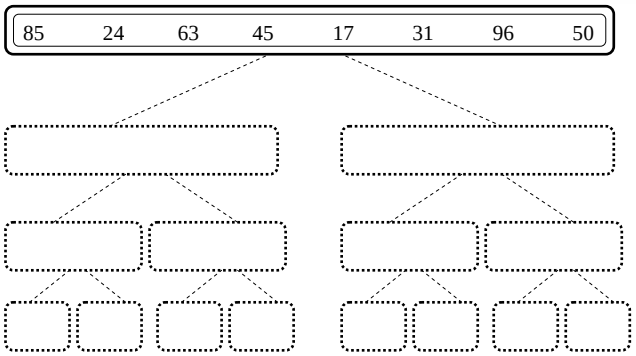
48

QuickSort
Ordenación rápida

- Partimos de un array a[]
- Cuando el número de elementos es 0 o 1 se para
- Escogemos un elemento cualquiera del array a[] (pivote p)
- Eliminamos el pivote y los elementos restantes se dividen en dos grupos
  - Los menores que el pivote a la izquierda (I)
  - Los mayores que el pivote a la derecha (D)
- Para cada uno de los grupos resultantes volvemos a aplicar el algoritmo: QuickSort(I), QuickSort(D)
- Devolvemos como resultado
  - QuickSort(I) seguido del pivote p seguido de QuickSort(D)

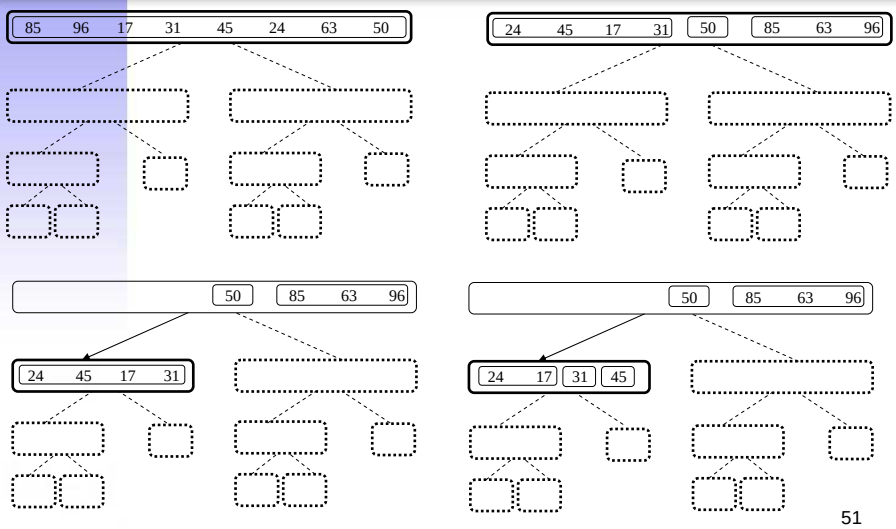
49

QuickSort
Ordenación rápida



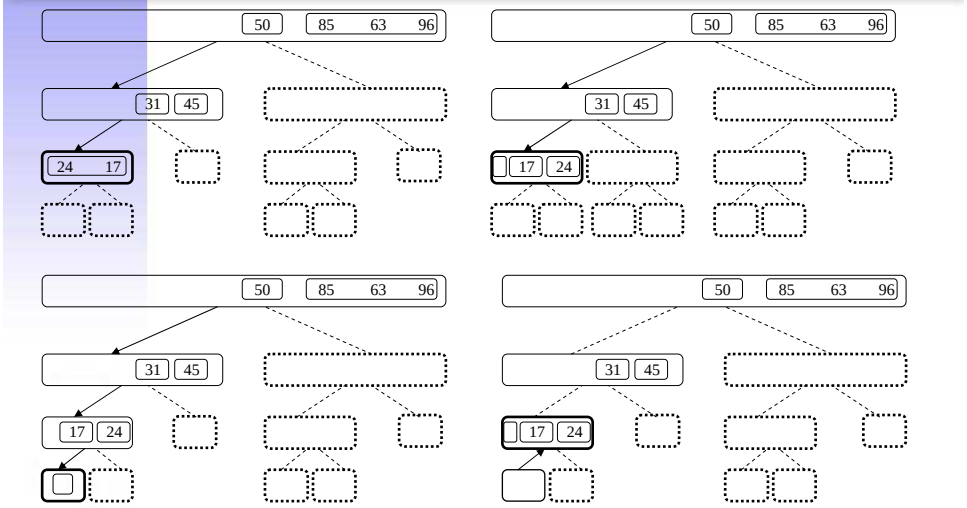
50

QuickSort
Ordenación rápida



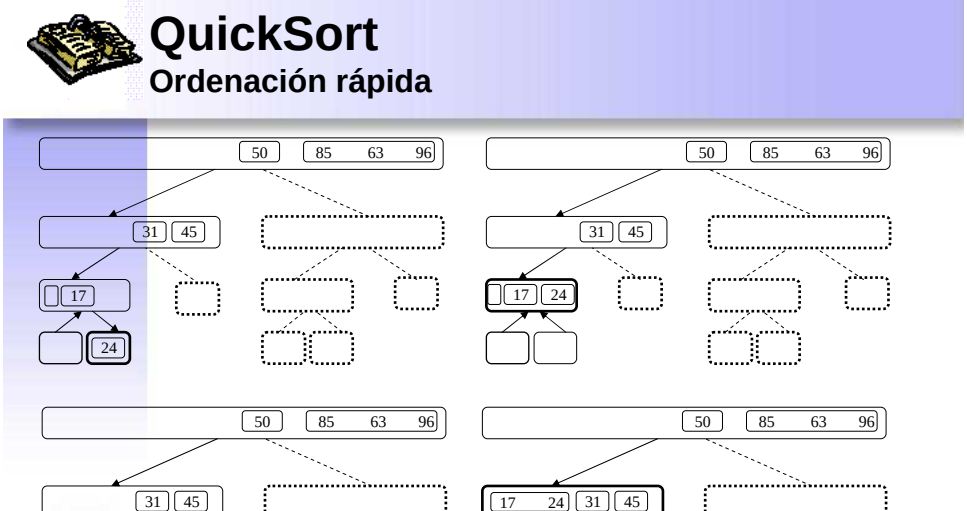
51

QuickSort
Ordenación rápida



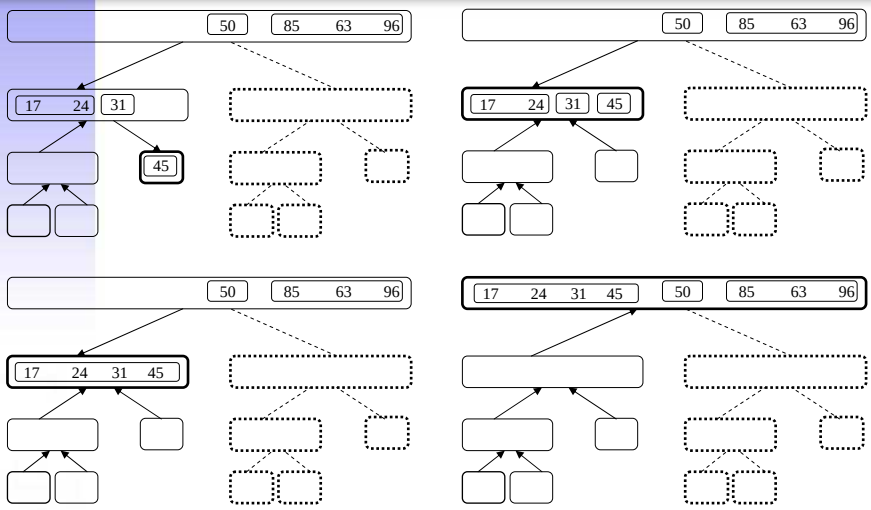
52

QuickSort
Ordenación rápida



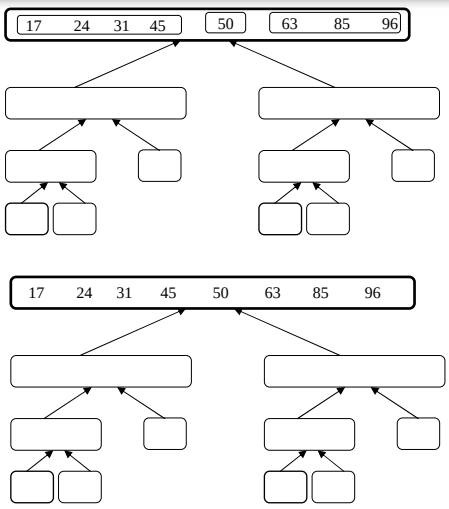
53

QuickSort
Ordenación rápida



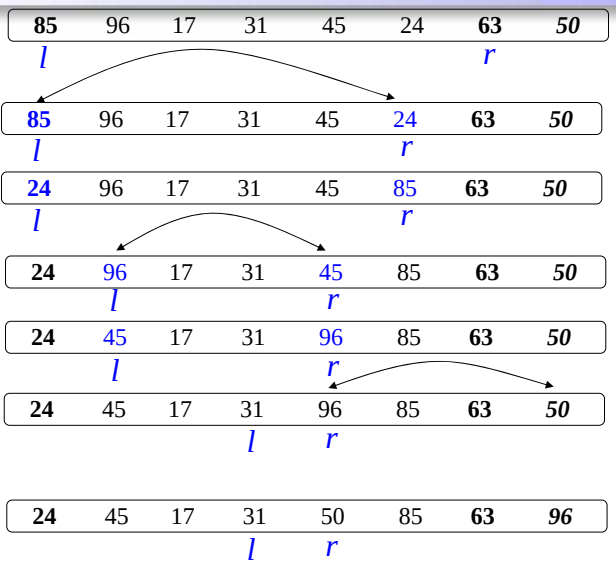
54

QuickSort
Ordenación rápida

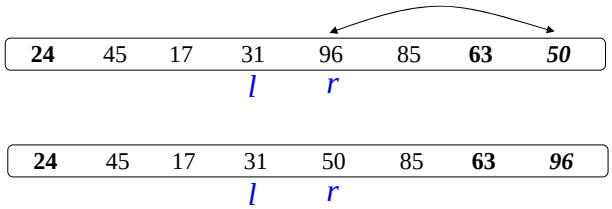


55

QuickSort
Ordenación rápida



QuickSort
Ordenación rápida



QuickSort
Implementación

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

```
public static void swapReferences (int a[], int index1, int index2) {
    int tmp=a[index1];
    a[index1]=a[index2];
    a[index2]=tmp;
}
```

58

QuickSort
Implementación recursiva. Pivote último elemento

```
private static void quickSort(int a[], int low, int high {
    if (low>=high) return;
    else {
      int pivot=a[high]; // pivote el último elemento
      int l=low;
      int r=high-1;
      do { // recorremos con l de izda a drcha con r de dcha a izda
        while (a[l] < pivot) l++; // Si estan bien colocados seguimos corriendo posiciones
        while (pivot < a[r]) r--; // Si estan bien colocados seguimos corriendo posiciones
        if (l<=r) {// Si no estan bien colocados los intercambiamos
          swapReferences(a, l, r);
          l++;
          r--;
        }
      }while (l<=r);
      if (low < r) quickSort(a, low, r);
      if (l < high) quickSort(a, l, high);
   }
}
```

QuickSort

Implementación recursiva. Pivote elemento central

```
private static void quickSort (int a[], int low, int high) {
  if (low>=high) return;
  else {
     int middle = (low+high)/2;
     int pivot=a[middle]; // pivote el elemento central
     int l=low;
     int r=high;
     do { // recorremos con l de izda a drcha con r de dcha a izda
       while (a[l] < pivot) l++; // Si estan bien colocados seguimos corriendo posiciones
       while (pivot < a[r]) r--; // Si estan bien colocados seguimos corriendo posiciones
       if (l<=r) {// Si no estan bien colocados los intercambiamos
          swapReferences(a, l, r);
          l++;
          r--;
       }
     }while (l<=r);
     if (low < r) quickSort(a, low, r);
     if (l < high) quickSort(a, l, high);
  }
}
```

QuickSort

Complejidad. Casos extremos

- **Caso Mejor**
  - **Cuando el pivote divide al array en dos subarrays del mismo tamaño**
  - **$t_{ejecución}$ $O(N \log_2 N)$**
- **Caso Peor**
  - **Cuando el pivote es el menor elemento**
  - **$t_{ejecución}$ $O(N^2)$**

QuickSort
Decisiones a tomar

- ¿ Cómo elegimos el pivote ?
  - **Elección incorrecta**: Para evitar caso peor, no coger nunca el primer o último elemento
  - **Elección segura**: El elemento central del array
  - **Elección más razonable**: Partición con la mediana de 3: Utiliza como pivote la mediana entre el primer elemento, el último y central.

67

QuickSort
Estrategia a seguir

- Seleccionamos el pivote utilizando la mediana de tres (ordenando los elementos primero, último y central) y seleccionando el del centro después de la ordenación
- Vamos recorriendo el array moviendo:
  - Los elementos menores a la izda del pivote
  - Los elementos mayores a la dcha del pivote
- Para ello buscamos
  - De izda a dcha elementos mayores con un contador i
  - De dcha a izda elementos menores con un contador j
  - Intercambiamos los elementos encontrados
  - Cuando lo índices se cruzan colocamos el pivote y hemos terminado

68

QuickSort

Implementación recursiva. Pivote mediana de tres

```
private static void quickSort(int a[], int low, int high) {
    if (low>=high) return;
    else { // ordenamos entre sí el primer elemento, el último y el del centro (mediana de tres)
            int middle=(low+high)/2;
            if (a[middle]<a[low])
                    swapReferences(a, low, middle);
            if (a[high]<a[low])
                    swapReferences(a, low, high);
            if (a[high]<a[middle])
                    swapReferences(a, middle, high);
            int pivot=a[middle]; // colocamos el pivote
         int l=low-1;
        int r=high-1;
        do { // recorremos con l de izda a drcha con r de dcha a izda
           while (a[l] < pivot) l++; // Si estan bien colocados sequimos corriendo posiciones
           while (pivot < a[r]) r--; // Si estan bien colocados sequimos corriendo posiciones
           if (l<=r) {// Si no estan bien colocados los intercambiamos
             swapReferences(a,l, r);
             l++; r--;
          }
        }while (l<=r);
        if (low < r) quickSort(a, low, r);
        if (l < high) quickSort(a, l, high);
   }
}
```

Quicksort: Referencias

- http://en.wikipedia.org/wiki/Quicksort
- http://es.wikipedia.org/wiki/Quicksort

70

Conclusiones

- **InsertionSort**: $O(N^2)$ es apropiado para pequeñas cantidades de datos
- **QuickSort** $O(N \log_2 N)$ tiene mejor rendimiento pero es complicado de implementar. Es apropiado para cantidades grandes de datos.
- Cuando el tamaño de entrada es tan grande que no cabe en memoria se utilizan técnicas distintas (algoritmos externos). Un algoritmo muy usado en este caso es **MergeSort** $O(N \log_2 N)$

71

Análisis de algoritmos

Algoritmos de Búsqueda

Índice

- ¿ Por qué es importante la búsqueda ?
- Algunos ejemplos
  - Búsqueda lineal
  - Búsqueda binaria
- Para cada algoritmo
  - En que consiste
  - Eficiencia. Casos extremos
  - Ejemplos de implementación

73

¿ Por qué es importante ?

Llamamos Búsqueda estática a la búsqueda que se hace sobre un array de elementos estáticos (no modifican su valor en el tiempo)

- Uno de los **usos mas frecuentes** de las computadoras es la búsqueda de información en estructuras de datos
- La eficiencia de un algoritmo de búsqueda depende de si el array sobre el que hacemos la búsqueda está o no **ordenado**: Ej. Palabras de un diccionario, ficheros en un directorio, catálogo en una biblioteca, etc.
- Muchos algoritmos de búsqueda invocan internamente un método de ordenación y el **coste del método de ordenación utilizado determina el coste global**.

Problema inicial

- Partimos de:
  - Una estructura de datos lineal que contiene determinado número de elementos, array **a[ ]**
  - Un valor **x** del mismo tipo que los elementos almacenados en el array.
- El resultado del algoritmo debe ser:
  - La posición del elemento **x** en el array **a[ ]** o
  - Una indicación de que no existe (p.ej -1)
  - Si **x** aparece mas de una vez devuelve cualquiera de las posiciones en la que está almacenado (es decir, depende de la implementación).

75

LinearSearch
Búsqueda Lineal

- Partimos de:
  - Un array de elementos **a[ ]** y
  - Un elemento **x**
- Recorremos el array de izda. a dcha. comparando cada uno de los elementos con x
  - Si son iguales devolvemos la posición del elemento y termina la búsqueda
  - Si son distintos seguimos buscando hasta llegar al final del array y si no lo encontramos lo notificamos con un mensaje de excepción

76

LinearSearch
Implementación

```
public static int LinearSearch(int a[], int x) {
      int i=0;
      while ((i<a.length)  && (a[i]!=x) )
        i++;
      if (i<a.length)
        return i;
      else
        return -1;
    }
```

77

LinearSearch (alternativa)
Implementación

```
public static int LinearSearch(int a[], int x) {
      int i=0;
      boolean encontrado = false;
      while ((i<a.length)  && (!encontrado)) {
         if (a[i]==x)
              encontrado = true;
        else
              i++;
      }
      if (encontrado)
         return i;
      else
         return -1;
    }
```

LinearSearch (sobre objetos)
Implementación

```
 public static <T extends Comparable<T>>
int LinearSearch(T a[], T x) {
       int i=0;
       while ((i<a.length)  && (a[i].compareTo(x)!=0) )
            i++;
       if (i<a.length)
         return i;
       else
         return -1;
     }
```

79

LinealSearch
Complejidad. Casos extremos

- Búsqueda sin éxito
  - $t_{ejecución}$ O(N) (Hay que recorrer todo el array)
- Búsqueda con éxito
  - Caso peor
    - $t_{ejecución}$ O(N) (Hay que recorrer todo el array)

80

BinarySearch
Búsqueda Binaria

- Sólo se puede aplicar con arrays ordenados previamente
  - Partimos de:
    - Un array **ordenado** de elementos **a[ ]** y
    - Un elemento **x**
  - Guardamos en una variable “central” el elemento que está en la posición intermedia del array
  - Comparamos el valor *central* con el valor de ***x***:
    - Si son iguales devolvemos la posición del elemento y termina la búsqueda
    - Si es menor repetimos la búsqueda entre la posición *inicial* y la del elemento *central-1*
    - Si es mayor repetimos la búsqueda entre la posición del elemento *central+1* y la *final*

82

BinarySearch
Implementación

```
public static int BinarySearch(int a[], int x) {
    int central;
    int inicio = 0;
    int fin = a.length -1;
    while (inicio <= fin) {
       central = (inicio + fin) / 2;
       if (a[central] == x)
         return central;
       if (x > a[central])
         inicio = central + 1;
       else   // (x < a[central])
         fin = central - 1;
    }
    return -1;
}
```

83

BinarySearch
Ejercicios

1. Adecua el algoritmo BinarySearch para buscar objetos de cualquier tipo, utilizando la comparación por defecto.

2. Adecua el algoritmo BinarySearch para buscar objetos de cualquier tipo, utilizando un comparador cualquiera.

84

BinarySearch
Complejidad. Casos extremos

En cada ciclo se considera una subtabla, la mitad de grande que en el caso anterior (marcada por inicio y fin)

n, n/2, n/4, n/8 ..... esto es: $n/2^i$

Ultima vuelta (caso peor) cuando $n/2^i=1$ → $2^i=n$

→ Esto es cuando $i=\log_2 n$

Luego se daran $\log_2 n$ vueltas (caso peor)

(definición de **$\log_2$ n**: nº al que hay que elevar 2 para que de n)

87

Conclusiones

- Para valores pequeños de N (ej N<10)
  - no merece la pena la búsqueda binaria porque utiliza aproximadamente el mismo número de comparaciones que la búsqueda secuencial
- Las últimas iteraciones de la búsqueda binaria progresan lentamente
- Puede interesar una aproximación **híbrida**
  - Aplicamos búsqueda binaria hasta que el rango es pequeño
  - Apartir de ese momento aplicamos búsqueda lineal

88

Consulta de libros

- Recomendable consultar:
  - [Joyanes07, cap. 17]
  - [Weiss10, cap. 8]
  - Cualquier otro libro que analiza los algoritmos de ordenación

89

Videos ilustrativos

- InsertionSort:
  - http://www.youtube.com/watch?v=c4BRHC7kTaQ&feature=relmfu
- MergeSort:
  - http://www.youtube.com/watch?v=GCae1WNvnZM&feature=relmfu
- QuickSort:
  - http://www.youtube.com/watch?v=y_G9BkAm6B8&feature=relmfu
  - http://www.youtube.com/watch?v=TeP1tG3HADw&feature=related

90

Links de interés

- java.util.Arrays:http://www.java2s.com/Tutorial/Java/0140__Collections/0150__Arrays-Utilities.htm
- http://www.it.uc3m.es/java/ordenacion/
- http://www.cs.ubc.ca/spider/harrison/Java/sorting-demo.html
- http://es.wikipedia.org/wiki/Categor%C3%ADa:Algoritmos_de_ordenamiento
- http://www.cs.brockport.edu/cs/javasort.html (no funciona)
- http://www.lfcia.org/~sjorge/palg/ (desaparecido)

91