Análisis y Diseño de Algoritmos

Tema 3: Divide y Vencerás

UNIVERSIDAD DE MÁLAGA
E.T.S. INGENIERÍA INFORMÁTICA

Contenido

- Introducción
  - Ejemplos iniciales
  - Esquema de la solución
  - Ventajas de la estrategia
- Divide y Vencerás
  - El teorema maestro
  - Ordenación por mezcla
  - Ordenación rápida
  - Búsqueda Binaria
  - Multiplicación de números grandes
  - Recorridos en un grafo DFS y BFS
- Referencias

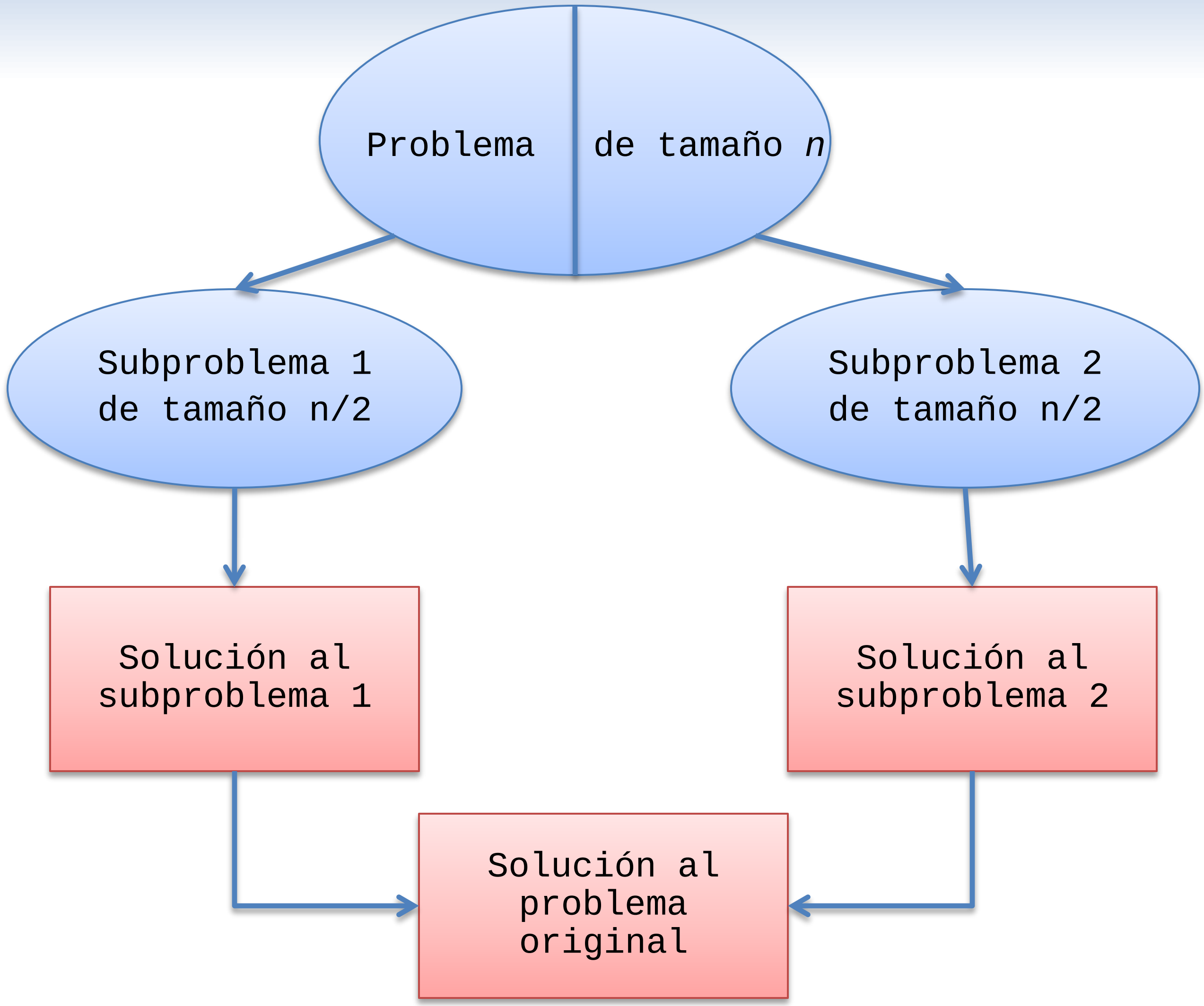
Introducción: Ejemplos iniciales

- Calcular x^n con x real y n potencia de 2.
- Acertar un número que he pensado entre 0 y 1000.
  - ¿Cuántas preguntas son necesarias en el caso peor?
  - Y si el rango es de 0 a n, ¿Cuántas preguntas?
- Array con exactamente un pico: encontrar el pico.
  - En los tres casos:
    - A) Fuerza bruta
    - B) Método más eficiente (pero igual de eficaz)

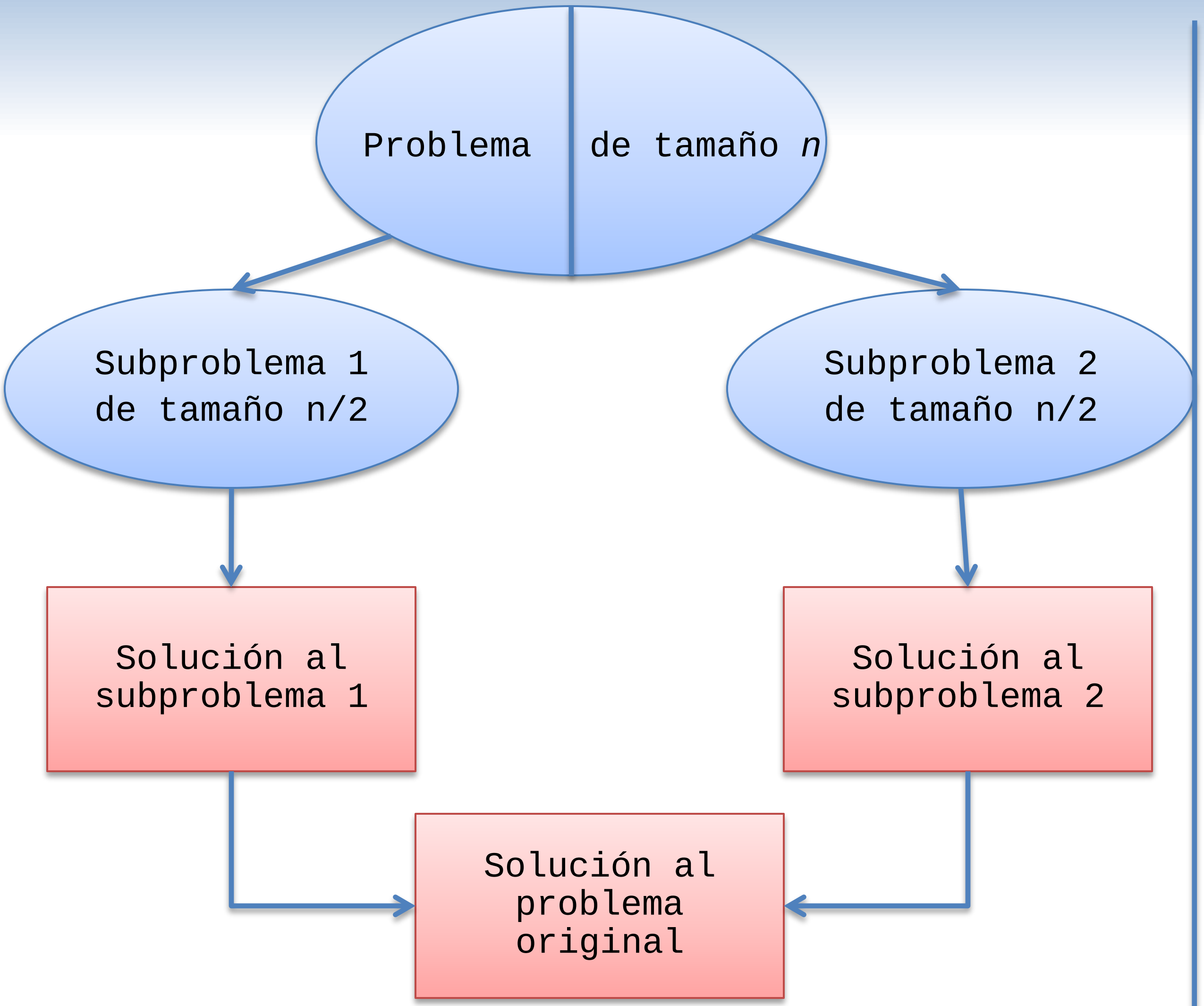
Introducción

- La estrategia Divide y Vencerás es una de las técnicas de resolución de problemas más conocidas. Se basa en
  - Para resolver una instancia de un problema, se divide en instancias más pequeñas del mismo problema (que tengan más o menos el mismo tamaño)
  - Se resuelven las instancias más pequeñas (normalmente utilizando recursividad)
  - Si es necesario se combinan las soluciones obtenidas para obtener la solución del problema original.

Introducción: esquema de la solución



Introducción: Ventajas de la estrategia



$$T(n) = n^2$$

$$T(\tfrac{n}{2}) = \frac{n^2}{4}$$

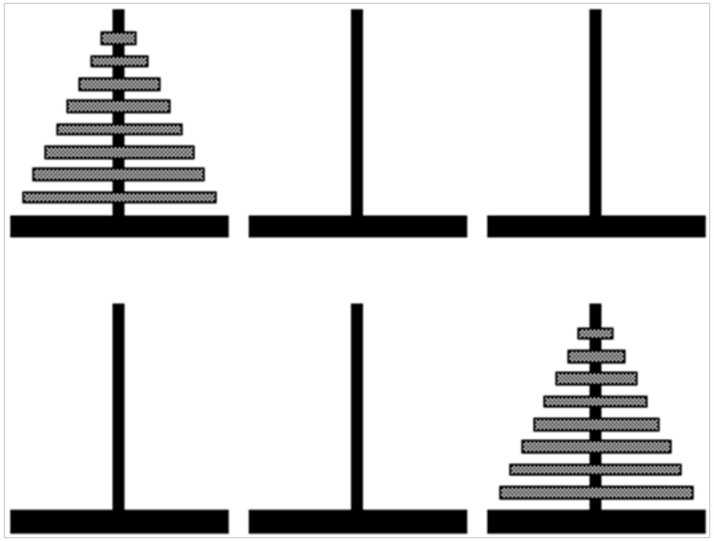
$$2 * T(\tfrac{n}{2}) = \frac{n^2}{2}$$

$$\frac{n^2}{2} < n^2$$

La solución resultante suele ser más rápida, que si se intenta resolver el problema directamente con todos los datos

Introducción: Ventajas de la estrategia

- Ejemplo: las torres de Hanoi



```java
/**
 *
 * @param n numero de discos
 * @param a varilla origen
 * @param b varilla intermedia
 * @param c varilla destino
 */

public static void hanoi(int n, char a,char b,char c){
  // n >=1
  if (n>0){
    hanoi(n-1,a,c,b);
    System.out.println("Pasar un disco de "+a+" a "+c);
    hanoi(n-1,b,a,c);
  }
}
```

Partir el problema, en muchos casos hace que la solución sea más fácil de encontrar sobre todo si se utiliza la recursividad

Divide y Vencerás:El teorema maestro

Teorema (**Teorema maestro**)Supón que del análisis de la complejidad de un algoritmo se obtiene la expresión $T(n) = aT(\frac{n}{b}) + f(n)$ entonces si $f(n) \in \Theta(n^d)$ con $d \geq 0$ se tiene que

$$T(n) = \begin{cases} \Theta(n^d) & si\ a < b^d \\ \Theta(n^d \log n) & si\ a = b^d \\ \Theta(n^{\log_b a}) & si\ a > b^d \end{cases}$$

El teorema maestro resuelve el problema de encontrar la complejidad de un algoritmo resuelto mediante DyV.

1. $\frac{n}{b}$ representa el tamaño de cada subproblema
2. a es el numero de subproblemas
3. $f(n)$ es la complejidad de combinar los subproblemas para encontrar la solucion al original

Divide y Vencerás

```java
/**
 * @param l lista de números a sumar
 * @param min y max, índices máximo y mínimo a suma
 * @return Suma de los elementos de la lista
 * Implementación recursiva
 */
public static int suma(List<Integer> l,int min,int max){
      if (max < min) return 0;
      if (max == min) return l.get(min);
      else {
         int s1 = suma(l, min,(min+max)/2);
         int s2 = suma(l, (min+max)/2+1,max);
         return s1+s2;
      }
}
```

$$T(n) = 2T(\tfrac{n}{2}) + 1$$

Divide y Vencerás

Teorema (**Teorema maestro**)Supón que del análisis de la complejidad de un algoritmo se obtiene la expresión $T(n) = aT(\frac{n}{b}) + f(n)$ entonces si $f(n) \in \Theta(n^d)$ con $d \geq 0$ se tiene que

$$T(n) = \begin{cases} \Theta(n^d) & si\ a < b^d \\ \Theta(n^d \log n) & si\ a = b^d \\ \Theta(n^{\log_b a}) & si\ a > b^d \end{cases}$$

$$T(n) = 2T(\tfrac{n}{2}) + 1$$

$$\begin{cases} \mathrm{a} = 2 \\ \mathrm{b} = 2 \\ \mathrm{d} = 0 \end{cases}$$

$$\boxed{2 > 2^0 \Rightarrow T(n) \in \Theta(n^{\log_2 2}) = \Theta(n)}$$

Búsqueda Binaria iterativa

```
• /**
•   * @param a array en el que se busca
•   *       suponemos que está ordenado
•   * @param x elemento buscado
•   * @return posición en la que aparece x, si
•   * está, o -1, si no está
•   */
• public static int busqBinaria(int[] a,int x){
•     int izda = 0,dcha = a.length-1;
•     int medio = (izda + dcha)/2;
•     while (izda <= dcha && a[medio]!=x){
•       if (a[medio]>x) dcha = medio - 1;
•       else izda = medio + 1;
•       medio = (izda + dcha)/2;
•     }
•     if (izda <= dcha) return medio;
•     else return -1;
• }
```

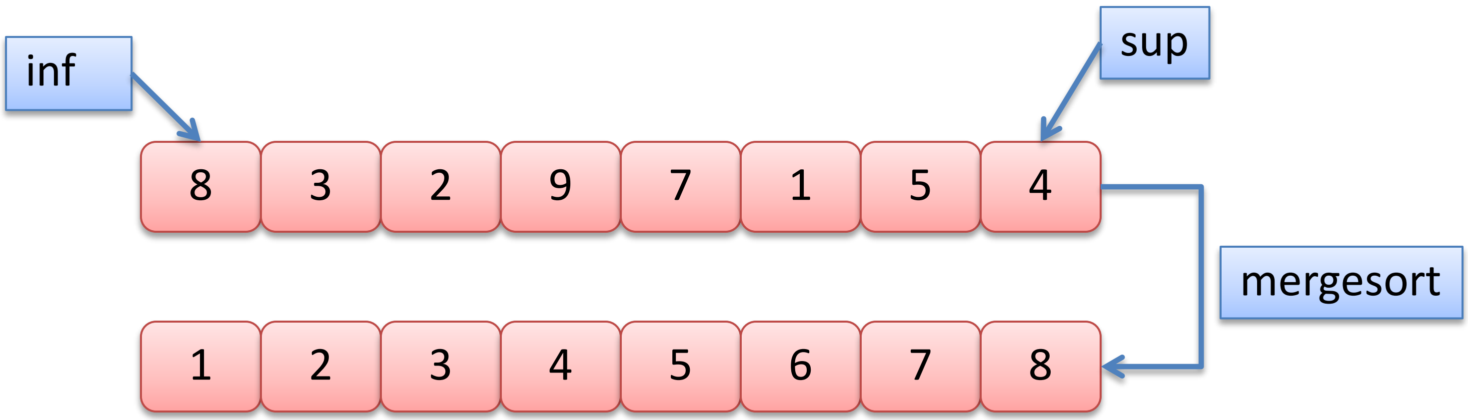
Busqueda binaria recursiva

```
static int busquedaBinariaRecursion(int a[],int n,int Iabajo,int Iarriba)
  {
        int Icentro=-1;
        if(Iarriba<Iabajo) {
              return -1;
        } else {
              Icentro=(Iabajo+Iarriba)/2;
              if (n<a[Icentro]) {
                    return(busquedaBinariaRecursion(a,n,Iabajo,Icentro-1));
              } else {
                if (n>a[Icentro]) {
                    return(busquedaBinariaRecursion(a,n,Icentro+1,Iarriba));
               } else {
                    return Icentro;
              }
              }
        }
  }
```

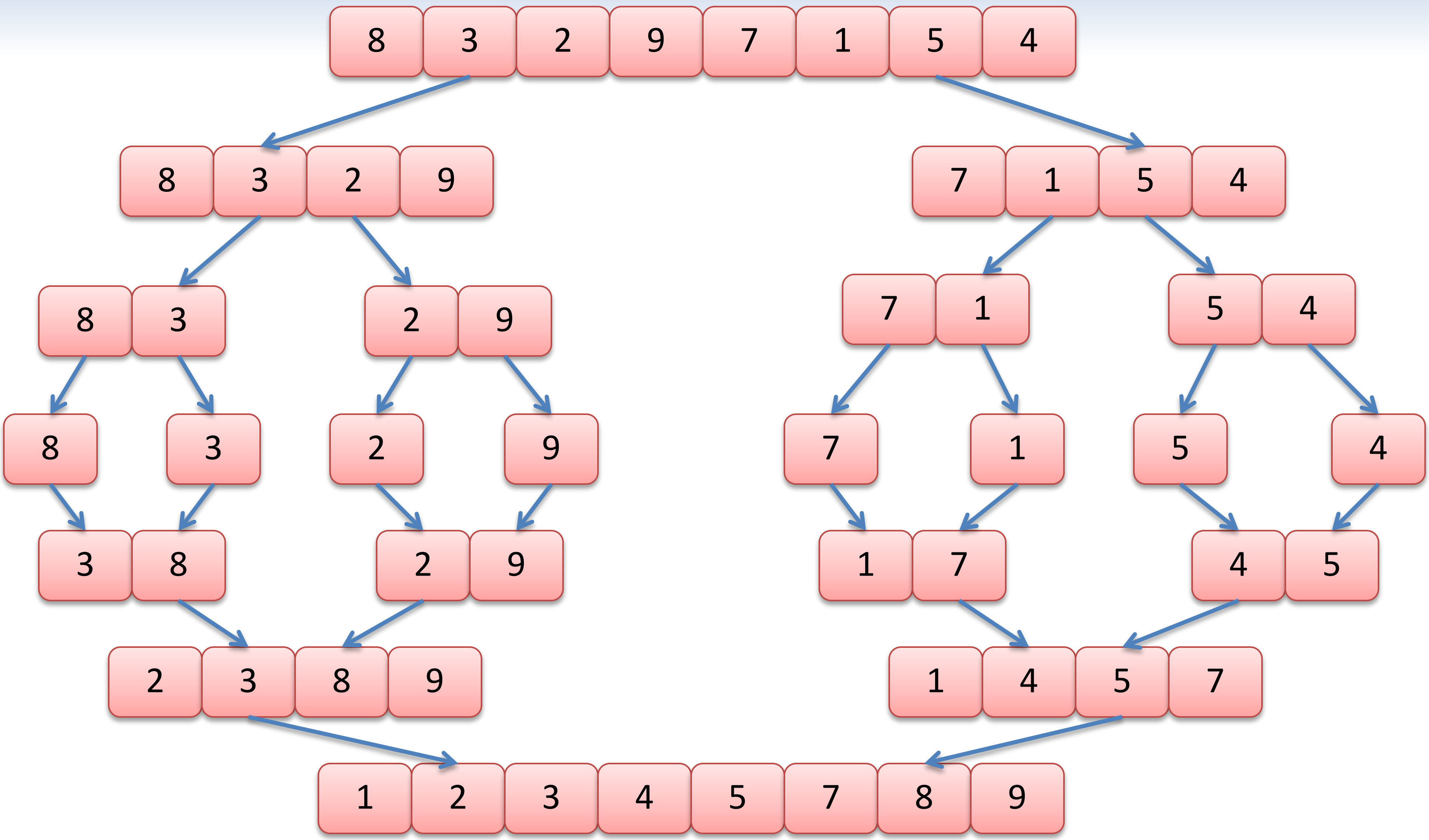
- **MÉTODOS DE ORDENACIÓN AVANZADOS**

Divide y Vencerás: Mergesort

```
/**
 *
 * @param a array con elementos desordenados
 * @param inf ordena mediante la ordenación por mezcla
 * @param sup el array a[inf..sup]
 */
public static void mergeSort(int[] a,int inf, int sup){
    if (inf < sup){
            mergeSort(a,inf,(inf+sup)/2);
            mergeSort(a,(inf+sup)/2+1,sup);
            mezclar(a,inf,(inf+sup)/2,sup);
     }
}
```



Divide y Vencerás: Mergesort



Divide y Vencerás: Mergesort

```
/**
 * mezcla las dos mitades de a[inf..sup]
 * de forma ordenada.
 * Utiliza un array intermedio
 */
public static void mezclar(int[] a,int inf,int medio,int sup){
    int i = inf; int j = medio+1;
    int[] b = new int[sup-inf+1];
    int k = 0;
    while (i<=medio && j <=sup){
      if (a[i]<=a[j]){
         b[k] = a[i];i++;
      }else{
         b[k] = a[j];j++;
      } k++;
    }
    while (i<=medio){
      b[k] = a[i];
      i++; k++;
    }
    while (j<=sup){
      b[k] = a[j];
      j++; k++;
    }
    k=0;
    for (int f=inf; f<= sup; f++){
      a[f] = b[k];k++;
     }
}
```

$$T_{merge}(n) = 2n - 1$$

(en el caso peor)

Divide y Vencerás: Mergesort

```java
/**
 *
 * @param a array con elementos desordenados
 * @param inf ordena mediante la ordenación por mezcla
 * @param sup el array a[inf..sup]
 */
public static void mergeSort(int[] a,int inf, int sup){
    if (inf < sup){
           mergeSort(a,inf,(inf+sup)/2);
           mergeSort(a,(inf+sup)/2+1,sup);
           mezclar(a,inf,(inf+sup)/2,sup);
     }
}
```

$$T(n) = 2T(\tfrac{n}{2}) + T_{merge}(n) = 2T(\tfrac{n}{2}) + 2n - 1$$

$$T(1) = 0$$

$$\boxed{T(n) \in \Theta(n \log n)}$$

Divide y Vencerás: Quicksort

```
/**
 * @param a array con elementos desordenados
 * ordena mediante la ordenación rápida
 * el array a[inf..sup]
 */
public static void quickSort(int[] a,int inf,int sup){
        if (inf < sup){
           int s = partir(a,inf,sup);
           quickSort(a,inf,s-1);
           quickSort(a,s+1,sup);
         }
}
```

método partir:

parte el array **a[inf..sup]** en dos subarrays (que pueden ser vacíos) **a[inf..q-1], a[q+1..sup]** tal que

-todos los elementos de **a[inf..q-1] son menores o iguales que a[q]**

-todos los elementos de **a[q+1..sup] son mayores o iguales que a[q]**

y devuelve el índice **q**

método quickSort:

-parte el array **a[inf..sup]** en dos subarrays (que pueden ser vacíos) **a[inf..q-1], a[q+1..sup]** con el método **partir**

-Ordena **a[inf..q-1] y a[q+1..sup]** llamando de forma recursiva a **quickSort**

Divide y Vencerás: QuickSort (Hoare)

```java
/**
 * Parte el array[inf..sup] en dos subarrays a[inf..j]
 * y a[j+1..sup], de forma que todos los elementos de
 * a[inf..j] son menores o iguales que un pivote y todos los elementos
 * de a[j+1..sup] son mayores o iguales que el pivote
 * @return el índice j
 */
private static  int partir(int[] a,int inf, int sup){
    int posPivote = inf, i = inf+1;
    int j = sup;
    // @inv a[inf..i-1] <= a[posPivote], a[j+1..sup] >= a[posPivote]
    while (i < j){
        while (a[j]>a[posPivote])
            j--;
        while (a[i]<a[posPivote])
            i++;
        if (i<j)
            intercambiar(a, i, j);
    }
    if(a[posPivote] > a[j])
        intercambiar(a, posPivote, j);
    return j;
}
```

```java
private static void intercambiar(int[] a,
int p, int d) {
    int aux = a[p];
    a[p] = a[d];
    a[d] = aux;
}
```

Divide y Vencerás: QuickSort (Hoare)

```
private static  int partir(int[] a,int inf, int sup){
    int posPivote = inf, i = inf+1;
    int j = sup;
    // @inv a[inf..i-1] <= a[posPivote], a[j+1..sup] >= a[posPivote]
    while (i < j){
        while (a[j]>a[posPivote])
            j--;
        while (a[i]<a[posPivote])
            i++;
        if (i<j)
            intercambiar(a, i, j);
    }
    if(a[posPivote] > a[j])
        intercambiar(a, posPivote, j);
    return j;
}
```

```
private static void intercambiar(int[] a, int p,
int d) {
    int aux = a[p];
    a[p] = a[d];
    a[d] = aux;
}
```

¿Por qué no hace falta la condición ***j >= inf*** en el primer bucle anidado?

En la primera iteración del bucle externo el pivote está a la izquierda de *j*, luego en el caso peor, el decremento de *j* terminará en la posición del pivote

En iteración *k>1* del bucle externo se sabe que en la iteración *k-1* se han intercambiado dos elementos, y **un elemento menor o igual que el pivote ha quedado a la izquierda de *j*,** por lo que, en el caso peor, la iteración del bucle anidado terminará en esa posición.

Divide y Vencerás: QuickSort (Hoare)

```java
private static  int partir(int[] a,int inf, int sup){
    int posPivote = inf, i = inf+1;
    int j = sup;
    // @inv a[inf..i-1] <= a[posPivote], a[j+1..sup] >= a[posPivote]
    while (i < j){
        while (a[j]>a[posPivote])
            j--;
        while (a[i]<a[posPivote])
            i++;
        if (i<j)
            intercambiar(a, i, j);
    }
    if(a[posPivote] > a[j])
        intercambiar(a, posPivote, j);
    return j;
}
```

```java
private static void intercambiar(int[] a, int p,
int d) {
    int aux = a[p];
    a[p] = a[d];
    a[d] = aux;
}
```

¿Por qué termina el bucle más externo?

En cada iteración j se decrementa al menos una vez, y i se incrementa al menos una vez, luego la distancia entre i y j se decrementa al menos dos unidades.

Divide y Vencerás: QuickSort (Hoare)

```
private static  int partir(int[] a,int inf, int sup){
    int posPivote = inf, i = inf+1;
    int j = sup;
    // @inv a[inf..i-1] <= a[posPivote], a[j+1..sup] >= a[posPivote]
    while (i < j){
        while (a[j]>a[posPivote])
            j--;
        while (a[i]<a[posPivote])
            i++;
        if (i<j)
            intercambiar(a, i, j);
    }
    if(a[posPivote] > a[j])
        intercambiar(a, posPivote, j);
    return j;
}
```

```
private static void intercambiar(int[] a, int p,
int d) {
    int aux = a[p];
    a[p] = a[d];
    a[d] = aux;
}
```

¿Es cierto el invariante?

Inicialmente: trivial.

Supongamos que es cierto antes de la iteración k>1, entonces
-j se decrementa hasta una posición pos1 tal que a[pos1] <= pivote
-i se incrementa hasta una posición pos2 tal que a[pos2] >= pivote

entonces
-si i < j, a[pos1] y a[pos2] se intercambian, y el invariante se cumple de nuevo antes de la iteración k+1
-si i >= j, los elementos no se intercambian, y el invariante se cumple al final del bucle

Divide y Vencerás: QuickSort (Hoare)

```
public static  int partir(int[] a,int inf, int sup){
  int pivote = a[inf]; int i = inf - 1;
  int j = sup + 1; // @inv a[inf..i-1] <= pivote, a[j+1..sup] >= pivote
  while (i < j){
    do{
      j--;
    }while (a[j]>pivote);
    do{
      i++;
    }while (a[i]<pivote);
……
  }
  return j;}
```

¿se cumple la postcondición?

Cuando el bucle termina i >= j
-si i = j, a[i] = a[j] = pivote, luego a[inf..j] <= pivote y a[j+1]>= pivote, y el array queda partido correctamente.
-si i = j + 1, luego a[inf..j] <= pivote y a[j+1..sup]>= pivote, y el array queda partido correctamente.

Divide y Vencerás: QuickSort (Hoare)

```
/**
 * @param a array con elementos desordenados
 * ordena mediante la ordenación rápida
 * el array a[inf..sup]
 */
public static void quickSortH(int[] a,int inf,int sup){
         if (inf < sup){
            int s = partir(a,inf,sup);
            quickSortH(a,inf,s);
            quickSortH(a,s+1,sup);
         }
}
```

Divide y Vencerás: QuickSort (análisis)

caso peor

Dado un array de tamaño *n*, el caso peor ocurre cuando el método partir produce un subarray de tamaño *n-1* y otro de tamaño *0*

$$T(n) = T(n-1) + T(0) + n = T(n-1) + n \in \Theta(n^2)$$

caso mejor

Dado un array de tamaño *n*, el caso mejor ocurre cuando el método partir produce dos subarrays de tamaño *n/2*

$$T(n) = 2T(\tfrac{n}{2}) + n \in \Theta(n \log n)$$

Divide y Vencerás: Búsqueda Binaria

El algoritmo de Búsqueda Binaria es un ejemplo de la metodología Divide y Vencerás

```java
/**
 * Busqueda binaria recursiva
 */
public static int busquedaBinariaRec(int[] a,int inf,int sup, int x){
   if (inf < sup){
       int medio = (inf + sup)/2;
       if (x == a[medio]) return medio;
       else if (x < a[medio])
              return busquedaBinariaRec(a,inf,medio-1,x);
            else
              return busquedaBinariaRec(a,medio+1,sup,x);
    } else return -1;
}
```

Divide y Vencerás: Multiplicación de números grandes

Supongamos que queremos multiplicar 23 por 14:
$23 = 2 * 10^1 + 3 * 10^0$, y $14 = 1 * 10^1 + 4 * 10^0$,
entonces, utilizando el algoritmo clasico,

$$\begin{aligned} 23 * 14 &= (2 * 10^1 + 3 * 10^0) * (1 * 10^1 + 4 * 10^0) \\ &= (2 \boxed{*} 1) * 10^2 + (2 \boxed{*} 4 + 3 \boxed{*} 1) * 10^1 + (3 \boxed{*} 4) * 10^0 \\ &= 322 \end{aligned}$$

Sin embargo, el numero de productos se puede reducir si el valor $2 * 4 + 3 * 1$ se calcula de otra forma

$$2 * 4 + 3 * 1 = (2 + 3) * (1 + 4) - 2 * 1 – 3 * 4$$

$$\begin{aligned} 23 * 14 \quad &= \quad (2 * 10^1 + 3 * 10^0) * (1 * 10^1 + 4 * 10^0) \\ &= \quad (2 \boxed{*} 1) * 10^2 + ((2 + 3) \boxed{*} (1 + 4) - 2 * 1 – 3 * 4) * 10^1 \\ &\quad\quad + (3 \boxed{*} 4) * 10^0 \\ &= \quad 322 \end{aligned}$$

Divide y Vencerás: Multiplicación de números grandes

Supongamos que queremos multiplicar dos números a y b de n cifras, siendo n un numero par:

$$a = a_1 * 10^{n/2} + a_0, b = b_1 * 10^{n/2} + b_0$$

entonces

$$c = a * b = c_2 * 10^n + c_1 * 10^{n/2} + c_0$$

donde

$$\begin{aligned} c_2 = & \quad a_1 * b_1 \\ c_1 = & \quad (a_1 + a_0) * (b_1 + b_0) - a_1 * b_1 - a_0 * b_0 \\ c_0 = & \quad a_0 * b_0 \end{aligned}$$

Divide y Vencerás: Multiplicación de números grandes

Suponiendo que $n = 2^k$, la complejidad del algoritmo es

$M(n) = 3M(n/2), \quad si\ n > 1$

$M(1) = 1$

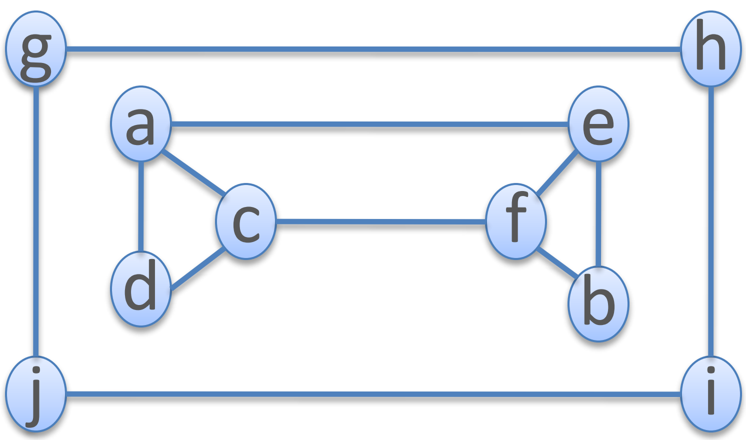
$$M(n) = 3M(n/2) = 3^2 M(n/4) = \cdots = 3^k M(1) = 3^k$$

como $k = \log_2 n$,

$$M(n) = 3^{\log_2 n} = n^{\log_2 3} \approx n^{1.585}$$

Recorridos en un grafo

- Un grafo G es un par (V,E) donde V es el conjunto de vértices, y E es el conjunto de arcos que unen dichos vértices.
- Los grafos pueden ser de muchos tipo
  - Dirigidos: si los arcos tienen dirección
  - Acíclicos: si no hay ningún camino en el grafo que una un mismo vértice.

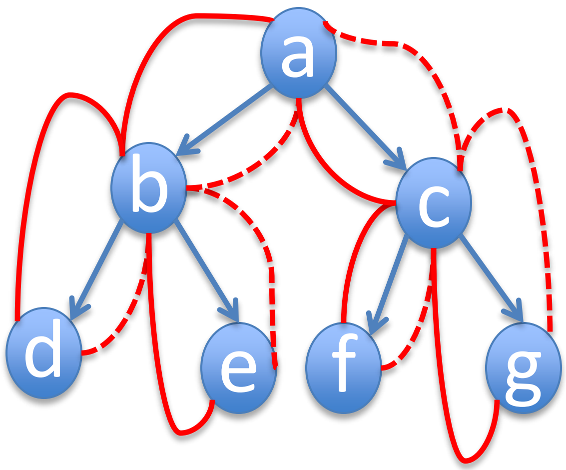


Recorridos en un grafo

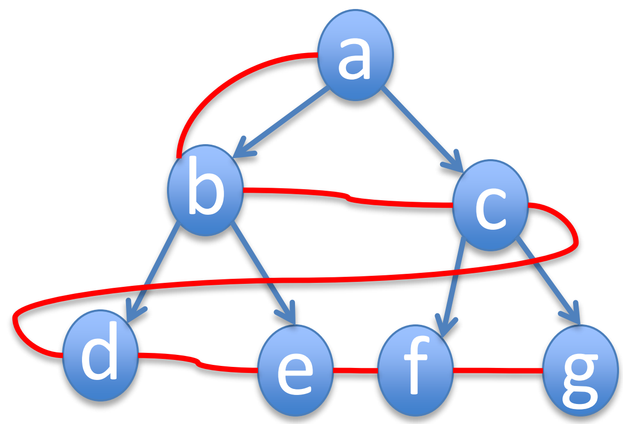
- Hay dos algoritmos para recorrer de forma sistemática todos los vértices de un grafo
  - El recorrido primero en profundidad, que utiliza una pila (depth-first search)
  - El recorrido primero en amplitud, que utiliza una cola fifo (breadth-first search)
  - En ambos casos, los vértices visitados se marcan de manera que, en cada iteración, el número de vértices a ser visitados disminuye en 1.

Recorridos en un grafo

- Si el grafo es un árbol, los algoritmos lo recorrerían así:



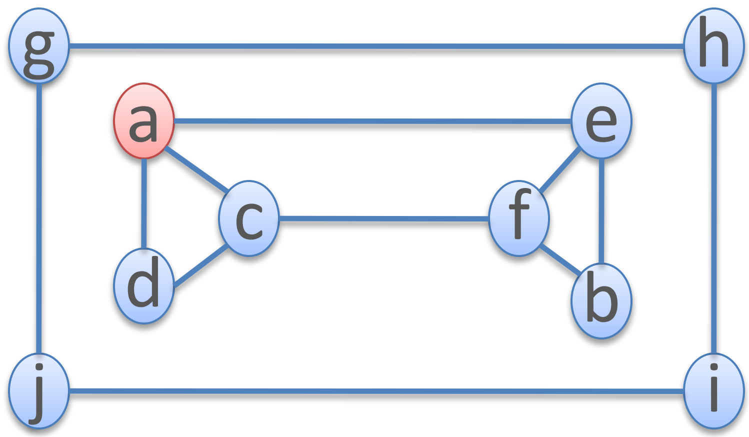
dfs----abdecfg



bfs----abcdefg

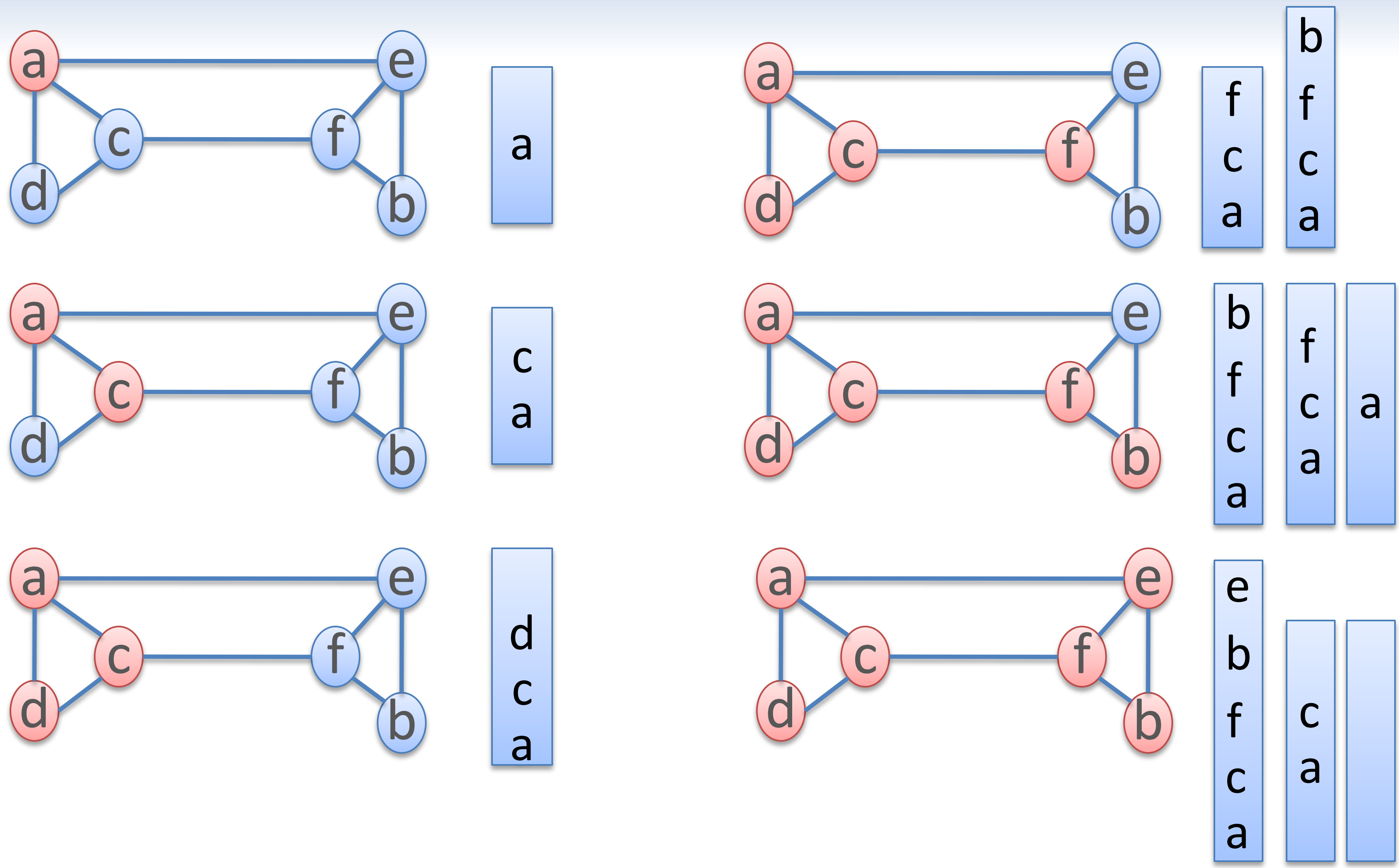
Recorridos en un grafo DFS

- Escoge un primer vértice de forma arbitraria y lo marca como visitado.
- En cada iteración, el algoritmo continúa por uno de los vértices adyacentes al actual, que no haya sido visitado.
- El proceso continúa hasta que el vértice actual no tenga ningún adyacente no visitado.
- Cuando esto ocurre el algoritmo vuelve hacia atrás hacia el vértice anterior visitado, buscando otro camino desde él.
- El algoritmo termina cuando se vuelve al vértice inicial.



- Si todos los vértices se han visitado, terminamos, sino empezamos de nuevo con alguno de los que no se han visitado todavía

Recorridos en un grafo
DFS-pila



Recorridos en un grafo
DFS

```
algoritmo dfs()
   marcar todos los vértices como
   no visitados
   para cada vértice en V
      si v no está marcado
        dfs(v);
```

La aparente simplicidad del algoritmo no debe engañarnos. DFS es un algoritmo complejo y potente.

La complejidad procede sobre todo de la combinación de iteración y recursividad que contiene.

```
algoritmo dfs(v)
//visita de forma recursiva todos
// los vértices conectados con v
// por algún camino
  marcar v como visitado
  para cada vértice w adyacente a v
     si w no está marcado
        dfs(w);
```

Recorridos en un grafo DFS- implementación Java

```
public class Vertice<T> {
        private final T vertice;
        private int visitado = 0;
        public Vertice(T vertice){
            this.vertice = vertice;
        }
        public void visitado(int b){
            visitado = b;
        }
        public int visitado(){
                return visitado;
        }
```

```
    public T vertice(){
        return vertice;
    }
    public boolean equals(Object o){
        return (o instanceof Vertice &&
        ((Vertice)o).vertice.equals(vertice));

    }
    public int hashCode(){
        return vertice.hashCode();
    }
    public String toString(){
        return "("+vertice.toString()+","+visitado+")";
    }
}
```

Recorridos en un grafo
DFS-complejidad

```
algoritmo dfs()
  marcar todos los vértices como
  no visitados
  para cada vértice en V
    si v no está marcado
      dfs(v);
```

$$\Theta(|V|)$$

```
algoritmo dfs(v)
//visita de forma recursiva todos
// los vértices conectados con v
// por algún camino
  marcar v como visitado
  para cada vértice w adyacente a v
    si w no está marcado
      dfs(w);
```

Recorridos en un grafo
DFS-complejidad

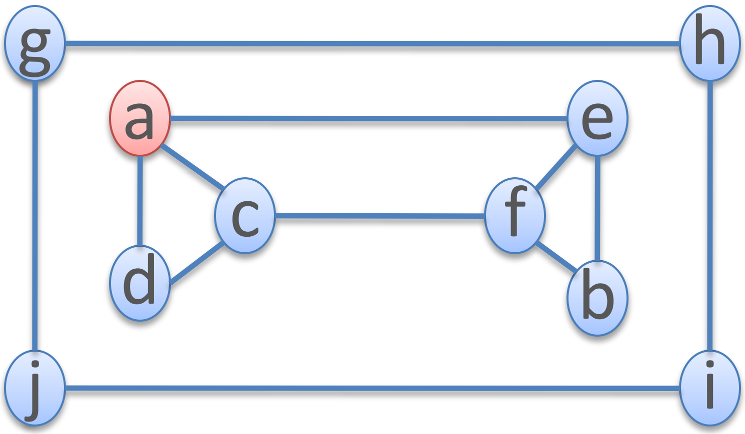
```
algoritmo dfs()
  marcar todos los vértices como
  no visitados
  para cada vértice en V
    si v no está marcado
      dfs(v);
```

Para cada vértice *V* sólo se llama a *dsf(v)* una vez, y para cada vértice la complejidad *dsf(v)* es a lo sumo *O(|E|)*, el número de arcos del grafo.

Ahora usamos los que se llama el ***análisis agregado:***
la complejidad de cada iteración es de media *O(|E|)/|V|*,
por lo que en total la complejidad de todas las iteraciones es *O(|E|).*
Por lo tanto, la complejidad del algoritmo es *O(|V|+|E|)*

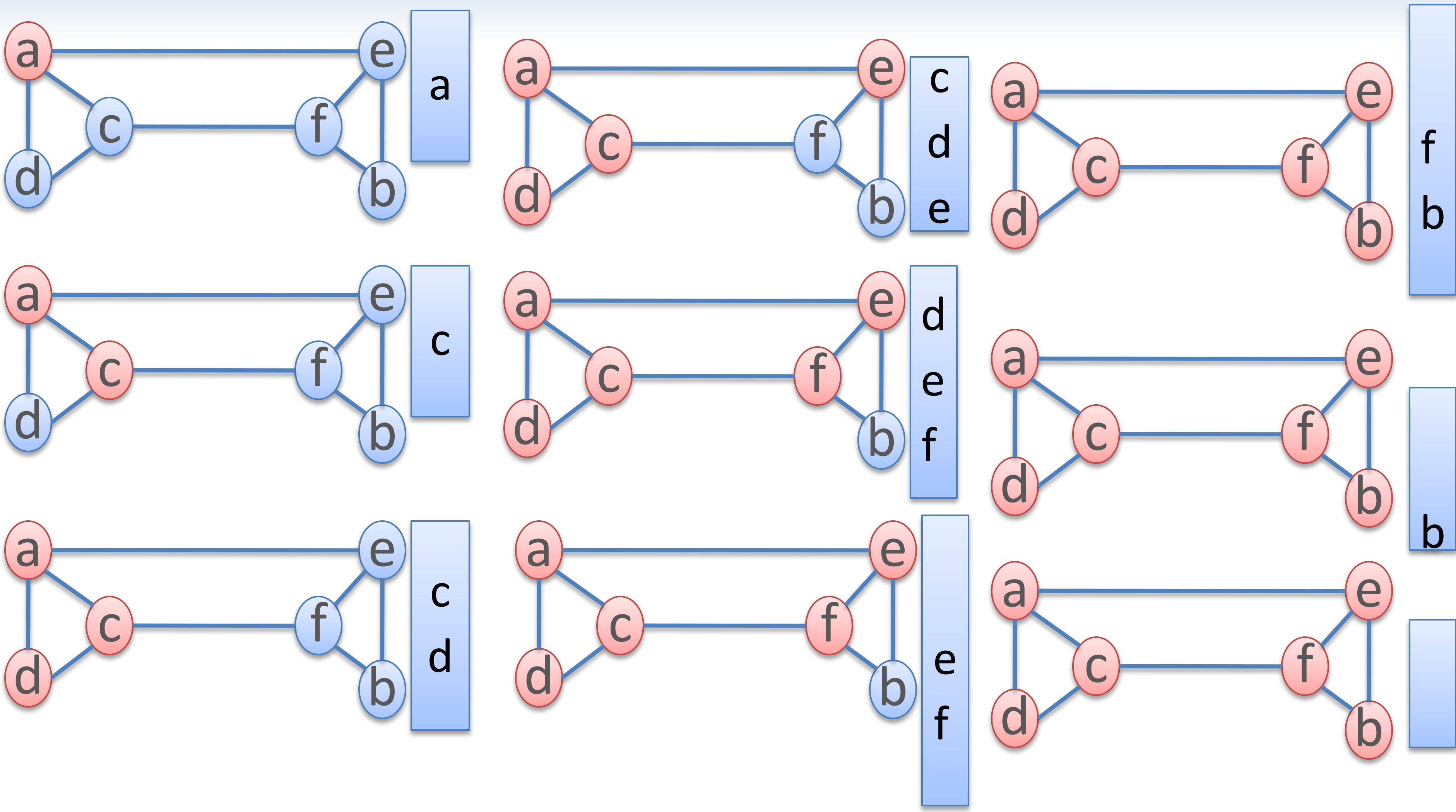
Recorridos en un grafo
BFS

- Escoge un primer vértice de forma arbitraria y lo marca como visitado.
- En cada iteración, el algoritmo continúa por uno de los vértices adyacentes al actual, que no haya sido visitado.
- El proceso continúa hasta que se ha visitado todos los vértices que se encuentran a distancia 1 del primero.
- Cuando esto ocurre el algoritmo, itera sobre todos los vértices que se encuentran a distancia 1, buscando los que se encuentran a distancia 2.
- El algoritmo termina cuando los vértices más lejanos al primer vértice han sido encontrados.



- Si todos los vértices se han visitado, terminamos, sino empezamos de nuevo con alguno de los que no se han visitado todavía

Recorridos en un grafo
BFS-cola



Recorridos en un grafo
BFS

```
algoritmo bfs()
  marcar todos los vértices como
  no visitados
  para cada vértice en V
    si v no está marcado
      bfs(v);
```

```
algoritmo bfs(v)
//visita de forma recursiva todos
// los vértices conectados con v
// por algún camino
  marcar v como visitado
  crear lista l
   añadir a la lista v
  mientras la lista no está vacía
    sacar el primer elemento x de la lista
    para cada w adyacente a x
        si w no está marcado
            marcar w como visitado
            añadir w al final de la lista
```

Recorridos en un grafo
BFS- implementación Java

```java
public class Grafo<T> {

    private Map<Vertice<T>,Set<Vertice<T>>> grafo;
    …..
    /**
    /*   @return el grafo (V,E) this, con todos los vértices
    /*   numerados según el orden de recorrido bfs
    /**
    public int dfs(){
        int cont = 0;
        for (Vertice<T> v:grafo.keySet()){
            if (v.visitado()==0)
                cont = bfs(v,++cont);
        }
        return cont;
    }
    ….
```

```java
public int bfs(Vertice<T> v,int cont){
      v.visitado(cont);cont++;
      List<Vertice<T>> lista
           = new LinkedList<Vertice<T>>();
       lista.add(v);
       while (lista.size() != 0){
           Vertice<T> x = lista.remove(0);
            for (Vertice<T> w: grafo.get(x)){
               if (w.visitado()==0){
                 w.visitado(cont);cont++;
                 lista.add(w);
               }
            }
          }
          return cont;
}
```

Recorridos en un grafo
BFS-Complejidad

```
algoritmo bfs()
  marcar todos los vértices como
  no visitados
  para cada vértice en V
    si v no está marcado
      bfs(v);
```

Razonando como en BFS, la complejidad de BFS es del orden de O(|V|+|E|)

```
algoritmo bfs(v)
//visita de forma recursiva todos
// los vértices conectados con v
// por algún camino
  marcar v como visitado
  crear lista l
   añadir a la lista v
  mientras la lista no está vacía
    sacar el primer elemento x de la lista
    para cada w adyacente a x
        si w no está marcado
           marcar w como visitado
           añadir w al final de la lista
```

Referencias

- *Introduction to The Design & Analysis of Algorithms*. A. Levitin. Ed. Adison-Wesley
- *Introduction to Algorithms*. T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein. Ed. The MIT Press