UNIVERSIDAD FRANCISCO DE VITORIA
VINCE IN BONO MALUM

Estructuras de Datos y Algoritmos

Tema 6. Introducción a los esquemas algorítmicos

Prof. Dr. P. Javier Herrera

Contenido

1 Conceptos básicos de grafos y su especificación algebraica

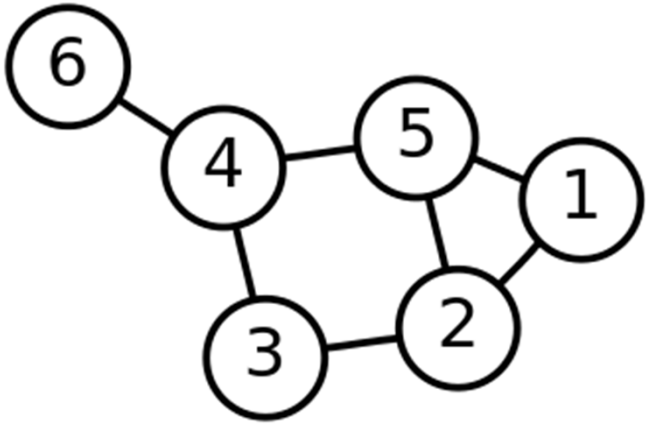
2 Algoritmos de ordenación

3 Algoritmos de búsqueda

1. Grafos

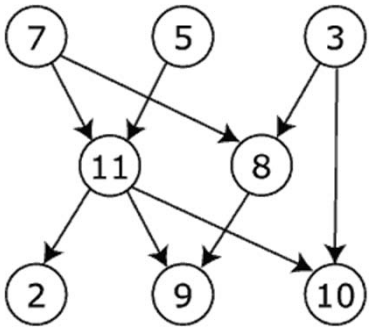
- Estructuras relacionales que generalizan las estructuras arbóreas.

- Un **grafo** G se define como un par de conjuntos finitos (V, A), donde V es el conjunto de **vértices** y $A \subseteq V \times V$ es un conjunto de pares de vértices llamados **aristas**.



1. Grafos

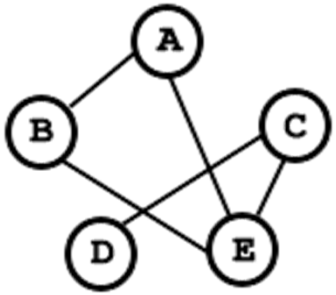
- Se distinguen diferentes tipos de grafos:



  - **Dirigidos**, cuando los pares de las aristas están ordenados.

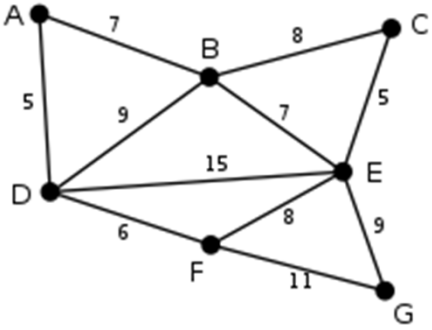
    En la arista (v,w), v es el **origen**, w es el **destino** y w es **adyacente** a v.

    El **grado de entrada** de un vértice v es el número de aristas de las cuales v es el destino. El **grado de salida** es el número de aristas de las cuales v es el origen.



  - **No dirigidos**, cuando los pares de las aristas no están ordenados.

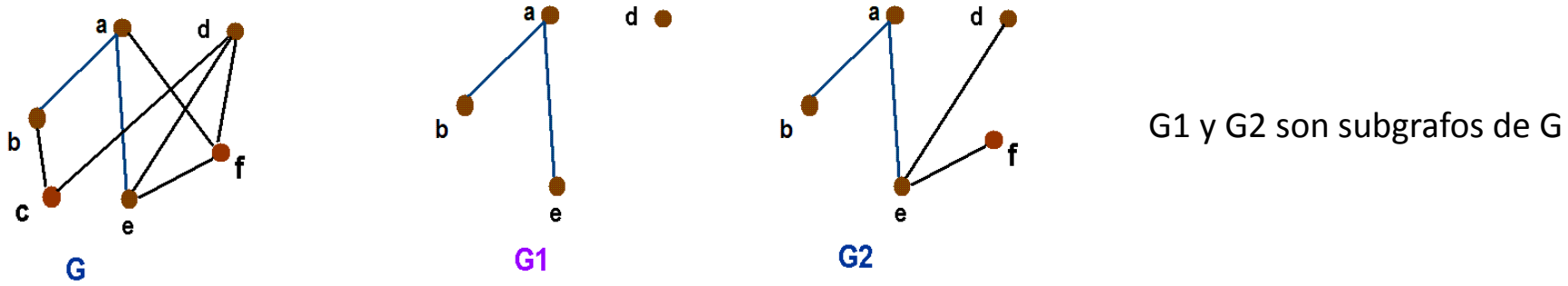
    Se habla simplemente de **grado** de un vértice y **extremos** de una arista, siendo ambos adyacentes entre sí.



  - **Valorados**, cuando las aristas tienen asociado algún valor (peso, longitud, etc.). Sus vértices pueden estar ordenados o no.

1.1 Terminología

- **Subgrafo**: $G' = (V', A')$ es **subgrafo** de $G = (V, A)$, denotado por $G' \subseteq G$, si $V' \subseteq V, A' \subseteq A$ y G' está bien definido como grafo.

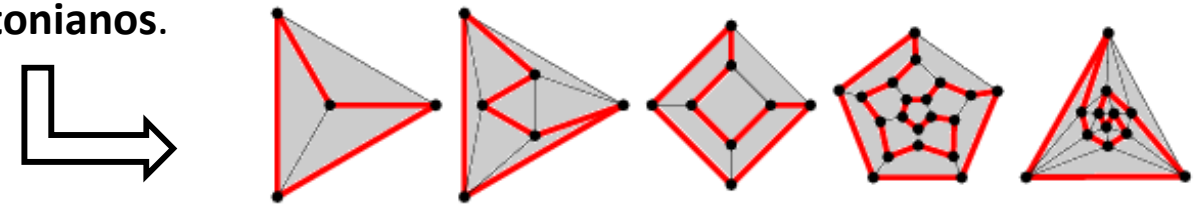


G1 y G2 son subgrafos de G

- **Camino**: Para un grafo $G = (V, A)$ la secuencia $(v_1, v_2, \ldots, v_n)$ con $n \geq 1$ y $v_i \in V$ es un **camino** en G si $\forall i : 1 \leq i < n : (v_i, v_{i+1}) \in A$.
  - La **longitud** de un camino es el número de aristas que lo forman.
  - La **distancia mínima** entre dos vértices de un grafo se define como la longitud del camino más corto que parte de *v* y llega hasta *w*.
  - Para grafos valorados, el **coste** de un camino es la suma total de los costes asociados a las aristas que lo forman.

1.1 Terminología

- **Ciclo**: Es un camino de longitud no nula que comienza y termina en el mismo vértice.
  - Se dice que un grafo es **acíclico** cuando no tiene ciclos.
  - Los ciclos que pasan exactamente una vez por cada vértice se denominan **hamiltonianos**.



- **Conexión**: Se dice que un grafo no dirigido es **conexo** si para cada par de vértices diferentes v y w existe un camino desde v a w.
  - Si un grafo no es conexo, los mayores conjuntos de vértices tales que los subgrafos inducidos son conexos se denominan **componentes conexas** del grafo.
  - Se dice que un grafo dirigido es **fuertemente conexo** si para cada par de vértices diferentes v y w existe un camino desde v a w y un camino desde w a v.
  - A los grafos no dirigidos, conexos y acíclicos se los denomina **árboles libres**.
  - Un **árbol de recubrimiento** de un grafo no dirigido es un árbol libre que es subgrafo del grafo original y que incluye todos sus vértices.

1.2 TAD de los grafos

- El TAD de los grafos contiene las siguientes operaciones:
  - crear el grafo vacío,
  - añadir un vértice,
  - añadir una arista,
  - eliminar un vértice y todas las aristas de las que es origen o destino,
  - eliminar una arista,
  - determinar si un vértice pertenece a un grafo,
  - determinar si una arista pertenece a un grafo,
  - determinar si un grafo es vacío, y
  - obtener el conjunto de vértices adyacentes a uno dado.
- Las ecuaciones son las que hacen que el grafo sea dirigido o no.

1.3 Especificación de grafos dirigidos

especificación *GRAFOS*[*VÉRTICES*]
usa *BOOLEANOS*, *CONJUNTOS*[*VÉRTICES*]
tipos *grafo*
operaciones

grafo-vacío	:			→ *grafo*	{ constructora }
añ-vértice	:	*vértice*	*grafo*	→ *grafo*	{ constructora }
añ-arista	:	*vértice*	*vértice* *grafo*	$\rightarrow_p$ *grafo*	{ constructora }
elim-vértice	:	*vértice*	*grafo*	→ *grafo*	
elim-arista	:	*vértice*	*vértice* *grafo*	→ *grafo*	
está-vértice?	:	*vértice*	*grafo*	→ *bool*	
está-arista?	:	*vértice*	*vértice* *grafo*	→ *bool*	
es-grafo-vacío?	:	*grafo*		→ *bool*	
adyacentes	:	*vértice*	*grafo*	$\rightarrow_p$ *conjunto*[*vértice*]	

variables

v,*w*, *v*′ ,*w*′ : *vértice*
g : *grafo*

1.3 Especificación de grafos dirigidos

ecuaciones

añ-vértice(v, añ-vértice(v, g)) = añ-vértice(v, g)

añ-vértice(w, añ-vértice(v, g)) = añ-vértice(v, añ-vértice(w, g))

añ-arista(v, w, g) = error
$\Leftarrow \neg$está-vértice?(v, g) $\vee \neg$está-vértice?(w, g)

añ-arista(v,w, añ-arista(v, w, g)) = añ-arista(v, w, g)

añ-arista(v, w, añ-arista(v', w', g)) = añ-arista(v',w', añ-arista(v, w, g))

añ-arista(v, w, añ-vértice(v', g)) = añ-vértice(v', añ-arista(v, w, g))
$\Leftarrow (v' \neq v \wedge v' \neq w) \vee$ (está-vértice?(v, g) $\wedge$ está-vértice?(w, g))

elim-vértice(v, grafo-vacío) = grafo-vacío

elim-vértice(v, añ-vértice(v, g)) = elim-vértice(v, g)

elim-vértice(v, añ-vértice(w, g)) = añ-vértice(w, elim-vértice(v, g)) $\Leftarrow$ $v \neq w$

elim-vértice(v, añ-arista(v, w, g)) = elim-vértice(v, g)

elim-vértice(v, añ-arista(w, v, g)) = elim-vértice(v, g)

elim-vértice(v, añ-arista(v' ,w', g)) = añ-arista(v', w', elim-vértice(v, g))
$\Leftarrow v \neq v' \wedge v \neq w'$

1.3 Especificación de grafos dirigidos

elim-arista(v, w, grafo-vacío) = grafo-vacío

elim-arista(v, w, añ-vértice(v', g)) = añ-vértice(v', elim-arista(v, w, g))

elim-arista(v, w, añ-arista(v, w, g)) = elim-arista(v, w, g)

elim-arista(v, w, añ-arista(v', w', g)) = añ-arista(v', w', elim-arista(v, w, g))
$\Leftarrow v \neq v' \vee w \neq w'$

está-vértice?(v, grafo-vacío) = falso

está-vértice?(v, añ-vértice(w, g)) = $v == w \vee$ está-vértice?(v, g)

está-vértice?(v, añ-arista(v', w', g)) = está-vértice?(v, g)

está-arista?(v, w, grafo-vacío) = falso

está-arista?(v, w, añ-vértice(v', g)) = está-arista?(v, w, g)

está-arista?(v, w, añ-arista(v', w', g)) = $(v == v' \wedge w == w') \vee$ está-arista?(v, w, g)

1.3 Especificación de grafos dirigidos

es-grafo-vacío?(grafo-vacío) = cierto
es-grafo-vacío?(añ-vértice(v, g)) = falso
es-grafo-vacío?(añ-arista(v, w, g)) = falso

adyacentes(v, g) = error ⇐ ¬está-vértice?(v, g)
adyacentes(v, añ-vértice(v, g)) = cjto-vacío ⇐ ¬está-vértice?(v, g)
adyacentes(v, añ-vértice(w, g)) = adyacentes(v, g) ⇐ $v \neq w \vee$ está-vértice?(v, g)
adyacentes(v, añ-arista(v, w, g)) = añadir(w, adyacentes(v, g))
adyacentes(v, añ-arista(v', w', g)) = adyacentes(v, g) ⇐ $v \neq v'$

fespecificación

1.4 Especificación de grafos no dirigidos

añ-arista(v, w, g) = añ-arista(w, v, g)

elim-arista(v, w, grafo-vacío) = grafo-vacío
elim-arista(v, w, añ-vértice(v', g)) = añ-vértice(v', elim-arista(v, w, g))
elim-arista(v, w, añ-arista(v, w, g)) = elim-arista(v, w, g)
elim-arista(v, w, añ-arista(w, v, g)) = elim-arista(v, w, g)
elim-arista(v, w, añ-arista(v', w', g)) = añ-arista(v', w', elim-arista(v, w, g))
$\Leftarrow (v \neq v' \vee w \neq w') \wedge (v \neq w' \vee w \neq v')$

está-arista?(v, w, grafo-vacío) = falso
está-arista?(v, w, añ-vértice(v', g)) = está-arista?(v, w, g)
está-arista?(v, w, añ-arista(v', w', g)) = $(v == v' \wedge w == w') \vee (v == w' \wedge w == v') \vee$ está-arista?(v, w, g)

1.4 Especificación de grafos no dirigidos

adyacentes(v, g)	=	error	⇐ ¬está-vértice?(v, g)
adyacentes(v, añ-vértice(v, g))	=	cjto-vacío	⇐ ¬está-vértice?(v, g)
adyacentes(v, añ-vértice(w, g))	=	adyacentes(v, g)	⇐ $v \neq w \vee$ está-vértice?(v, g)
adyacentes(v, añ-arista(v, w, g))	=	añadir(w, adyacentes(v, g))	
adyacentes(v, añ-arista(w, v, g))	=	añadir(w, adyacentes(v, g))	
adyacentes(v, añ-arista(v', w', g))	=	adyacentes(v, g)	⇐ $v \neq v' \wedge v \neq w'$

1.5 Especificación de grafos valorados dirigidos

especificación *GRAFOS-VALORADOS*[*VÉRTICES*, *VALORES*]
usa *BOOLEANOS*, *CONJUNTOS*[*VÉRTICES*]
tipos *grafo-val*
operaciones

gv-vacío	:	→ *grafo-val*	{ constructora }
gv-añ-vértice	: *vértice* *grafo-val*	→ *grafo-val*	{ constructora }
gv-añ-arista	: *vértice* *vértice* *valor* *grafo-val*	$\rightarrow_p$ *grafo-val*	{ constructora }
gv-valor	: *vértice* *vértice* *grafo-val*	$\rightarrow_p$ *valor*	
gv-elim-vértice	: *vértice* *grafo-val*	→ *grafo-val*	
gv-elim-arista	: *vértice* *vértice* *grafo-val*	→ *grafo-val*	
gv-está-vértice?	: *vértice* *grafo-val*	→ *bool*	
gv-está-arista?	: *vértice* *vértice* *grafo-val*	→ *bool*	
gv-es-vacío?	: *grafo-val*	→ *bool*	
gv-adyacentes	: *vértice* *grafo-val*	$\rightarrow_p$ *conjunto*[*vértice*]	

variables

v, *w*, *v′*, *w′* : *vértice*
p, *q*, *p′* : *valor*
g : *grafo-val*

1.5 Especificación de grafos valorados dirigidos

ecuaciones

gv-añ-vértice(v, gv-añ-vértice(v, g)) = gv-añ-vértice(v, g)

gv-añ-vértice(w, gv-añ-vértice(v, g)) = gv-añ-vértice(v, gv-añ-vértice(w, g))

gv-añ-arista(v, w, p, g) = error
$\Leftarrow \neg$gv-está-vértice?(v, g) $\vee \neg$gv-está-vértice?(w, g)

gv-añ-arista(v, w, p, gv-añ-arista(v, w, q, g)) = gv-añ-arista(v, w, p, g)

gv-añ-arista(v, w, p, gv-añ-arista(v', w', p', g)) = gv-añ-arista(v', w', p', gv-añ-arista(v, w, p, g))
$\Leftarrow v \neq v' \vee w \neq w'$

gv-añ-arista(v, w, p, gv-añ-vértice(v', g)) = gv-añ-vértice(v', gv-añ-arista(v, w, p, g))
$\Leftarrow (v' \neq v \wedge v' \neq w) \vee$ (gv-está-vértice?(v, g) $\wedge$ gv-está-vértice?(w, g))

gv-valor(v, w, gv-vacío) = error

gv-valor(v, w, gv-añ-vértice(v', g)) = gv-valor(v, w, g)

gv-valor(v, w, gv-añ-arista(v, w, p, g)) = p

gv-valor(v, w, gv-añ-arista(v', w', p', g)) = gv-valor(v, w, g) $\Leftarrow v \neq v' \vee w \neq w'$

…

2. Algoritmos de ordenación

- La **ordenación** o **clasificación** de datos (*sort*, en inglés) es una operación consistente en disponer un conjunto (estructura) de datos en algún determinado orden con respecto a uno de los campos de elementos del conjunto.

- El orden de clasificación u ordenación puede ser *ascendente* o *descendente*. La lista se dice que está en orden ascendente si:

  ```
  i < j implica que k[i] <= k[j]
  ```

  y se dice que está en orden descendente si:

  ```
  i < j implica que k[i] >= k[j]
  ```

  para todos los elementos de la lista.

- Existen numerosos algoritmos de ordenación de arrays: inserción, burbuja, selección, rápido (*quick sort*), fusión (*merge*), montículo (*heap*), shell, etc.

2. Algoritmos de ordenación

- Los métodos de ordenación se suelen dividir en dos grandes grupos:
  - ***Directos***: burbuja, selección, inserción.
  - ***Indirectos* (*avanzados*)**: Shell, ordenación rápida (*quick sort*), ordenación por mezcla, Radixsort.

- En el caso de listas pequeñas, los métodos directos se muestran eficientes, sobre todo porque los algoritmos son sencillos; su uso es muy frecuente. Sin embargo, en listas grandes estos métodos se muestran ineficaces y es preciso recurrir a los métodos avanzados.

- ¿Cómo se sabe cuál es el mejor algoritmo? La **eficiencia** es el factor que mide la calidad y rendimiento de un algoritmo.

2. Algoritmos de ordenación

- En el caso de la operación de ordenación, dos criterios se suelen seguir a la hora de decidir qué algoritmo es el más eficiente:
  1. menor tiempo de ejecución en computadora;
  2. menor número de instrucciones.

- Sin embargo, no siempre es fácil efectuar estas medidas. Por esta razón, el mejor criterio para medir la eficiencia de un algoritmo es aislar una operación específica clave en la ordenación y contar el número de veces que se realiza.

- Así, en el caso de los algoritmos de ordenación, se utilizará como medida de su eficiencia el **número de comparaciones entre elementos** efectuados. El algoritmo de ordenación A será más eficiente que el B, si requiere menor número de comparaciones.

2.1 Algoritmos de ordenación básicos

- Los algoritmos básicos de ordenación más simples y clásicos son:
  - **Ordenación por burbuja** (*Bubble sort*)
  - **Ordenación por selección** (*Selection sort*)
  - **Ordenación por inserción** (*Insertion sort*)

- Los métodos más recomendados son: selección e inserción, aunque se estudiará el método de burbuja, por ser el más sencillo.

- Las técnicas que se analizarán a continuación considerarán, esencialmente, la ordenación de elementos de una lista (array) en orden ascendente.

2.2 Ordenación por burbuja

- La ordenación por burbuja es uno de los métodos más fáciles de ordenación.

- La técnica consiste en hacer varias pasadas a través del array. En cada pasada, se comparan parejas sucesivas de elementos. Si una pareja está en orden creciente (o los valores son idénticos), se dejan los valores como están. Si una pareja está en orden decreciente, sus valores se intercambian en el array.

6 5 3 1 8 7 2 4

- En el caso de un array con *n* elementos, la ordenación por burbuja requiere hasta *n−1* pasadas.

2.2 Ordenación por burbuja

- Los pasos del algoritmo son:
  1. En la primera pasada se comparan elementos adyacentes:

     `(A[0],A[1]),(A[1],A[2]),(A[2],A[3]),...(A[n-2],A[n-1])`

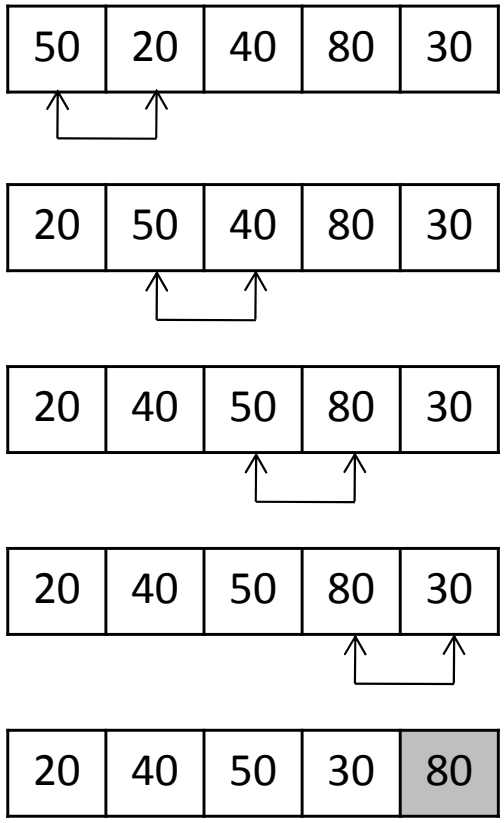
     Se realizan *n−1* comparaciones, por cada pareja `(A[i],A[i+1])` se intercambian los valores si `A[i+1]<A[i]`. Al final de la pasada, el elemento mayor de la lista está situado en `A[n-1]`.
  2. En la segunda pasada se realizan las mismas comparaciones e intercambios, terminando con el elemento segundo mayor valor en `A[n-2]`.
  3. El proceso termina con la pasada *n−1*, en la que el elemento más pequeño se almacena en `A[0]`.

- *Ejemplo*:

Lista desordenada	50	20	40	80	30

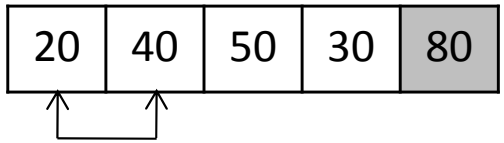
2.2 Ordenación por burbuja

- Primera pasada:

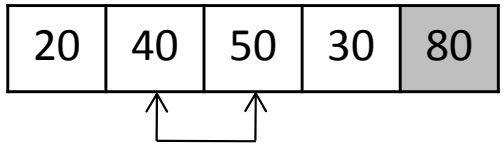


2.2 Ordenación por burbuja

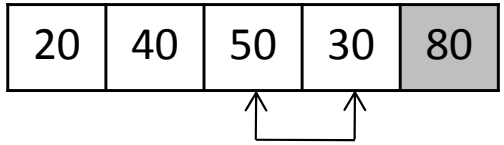
- Segunda pasada:



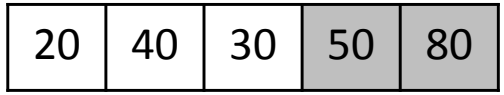
20 y 40 ya están ordenados



40 y 50 ya están ordenados



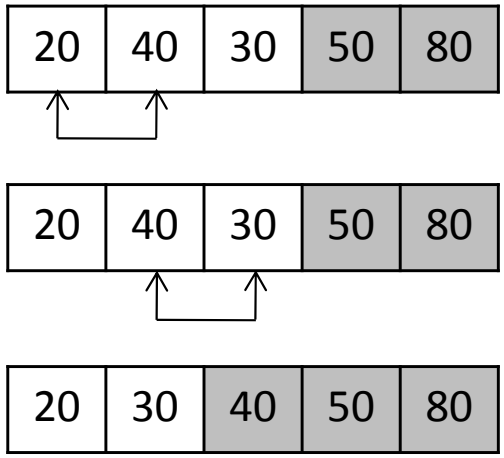
Se intercambian 50 y 30



50 y 80 son los elementos mayores y están ordenados

2.2 Ordenación por burbuja

- Tercera pasada:



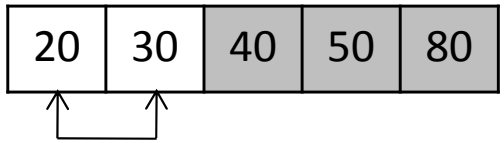
20 y 40 ya están ordenados

Se intercambian 40 y 30

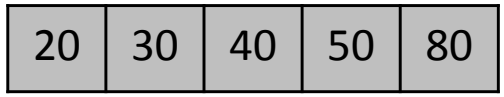
40, 50 y 80 están ordenados

2.2 Ordenación por burbuja

- Cuarta pasada:



20 y 30 ya están ordenados



Lista ordenada

2.2 Ordenación por burbuja

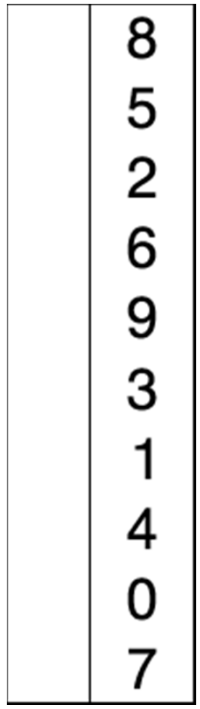
- Codificación del algoritmo de la burbuja:

$O(n^2)$

```
proc ordBurbuja (a[]: ent, n: ent)
    interruptor, pasada, j, aux : ent;
    interruptor := 1;
    para pasada := 1 hasta (n-1 and interruptor) paso 1 hacer
    // bucle externo controla la cantidad de pasadas
        interruptor := 0;
        para j := 0 hasta n-pasada-1 paso 1 hacer
            si (a[j] > a[j+1]) entonces
            // elementos desordenados, es necesario intercambio
                interruptor := 1;
                aux := a[j];
                a[j] := a[j+1];
                a[j+1] := aux;
            fsi
        fpara
    fpara
fproc
```

2.3 Ordenación por selección

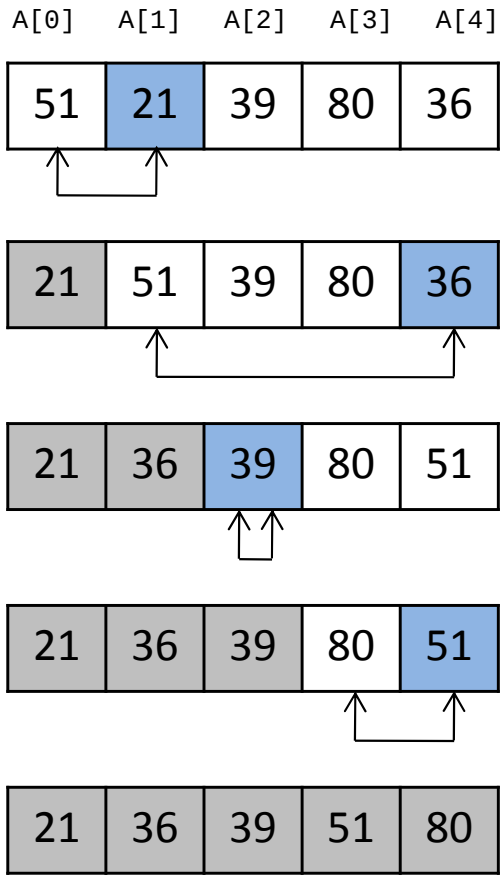
- El algoritmo se apoya en sucesivas pasadas que **intercambian el elemento más pequeño sucesivamente** con el primer elemento de la lista, `A[0]` en la primera pasada.



- Los pasos del algoritmo son:
  1. Seleccionar el elemento más pequeño de la lista A; intercambiarlo con el primer elemento `A[0]`. Ahora la entrada más pequeña está en la primera posición del vector.
  2. Considerar las posiciones de la sublista desordenada `A[1]`, `A[2]`, `A[3]`…, `A[n]`, seleccionar el elemento más pequeño e intercambiarlo con `A[1]`. Ahora las dos primeras entradas de A están en orden.
  3. Continuar este proceso encontrando o seleccionando el elemento más pequeño de los restantes elementos de la sublista desordenada, intercambiándolos adecuadamente.
- El proceso continúa *n − 1* pasadas y en ese momento la lista desordenada se reduce a un elemento (el mayor de la lista) y el array completo ha quedado ordenado.

2.3 Ordenación por selección

- Ejemplo:



Primera pasada. Seleccionar 21
Intercambiar 21 y `A[0]`

Segunda pasada. Seleccionar 36
Intercambiar 36 y `A[1]`

Tercera pasada. Seleccionar 39
Intercambiar 39 y `A[2]`

Cuarta pasada. Seleccionar 51
Intercambiar 51 y `A[3]`

Lista ordenada

2.3 Ordenación por selección

- Codificación del algoritmo de selección:

$O(n^2)$

```
proc ordSeleccion (a[]: ent, n: ent)
    indiceMenor, i, j, aux: ent;
    // ordenar a[1]..a[n-2] y a[n-1] en cada pasada
    para i := 0 hasta n-1 paso 1 hacer
        indiceMenor := i;          // comienzo de la exploración en índice i
        // j explora la sublista a[i+1]..a[n]
        para j := i+1 hasta n paso 1 hacer
            si (a[j] < a[indiceMenor]) entonces
                indiceMenor := j;
            fsi
        fpara
        //sitúa el elemento más pequeño en a[i]
        si (i != indiceMenor) entonces
            aux := a[i];
            a[i] := a[indiceMenor];
            a[indiceMenor] := aux ;
        fsi
    fpara
proc
```

2.4 Ordenación por inserción

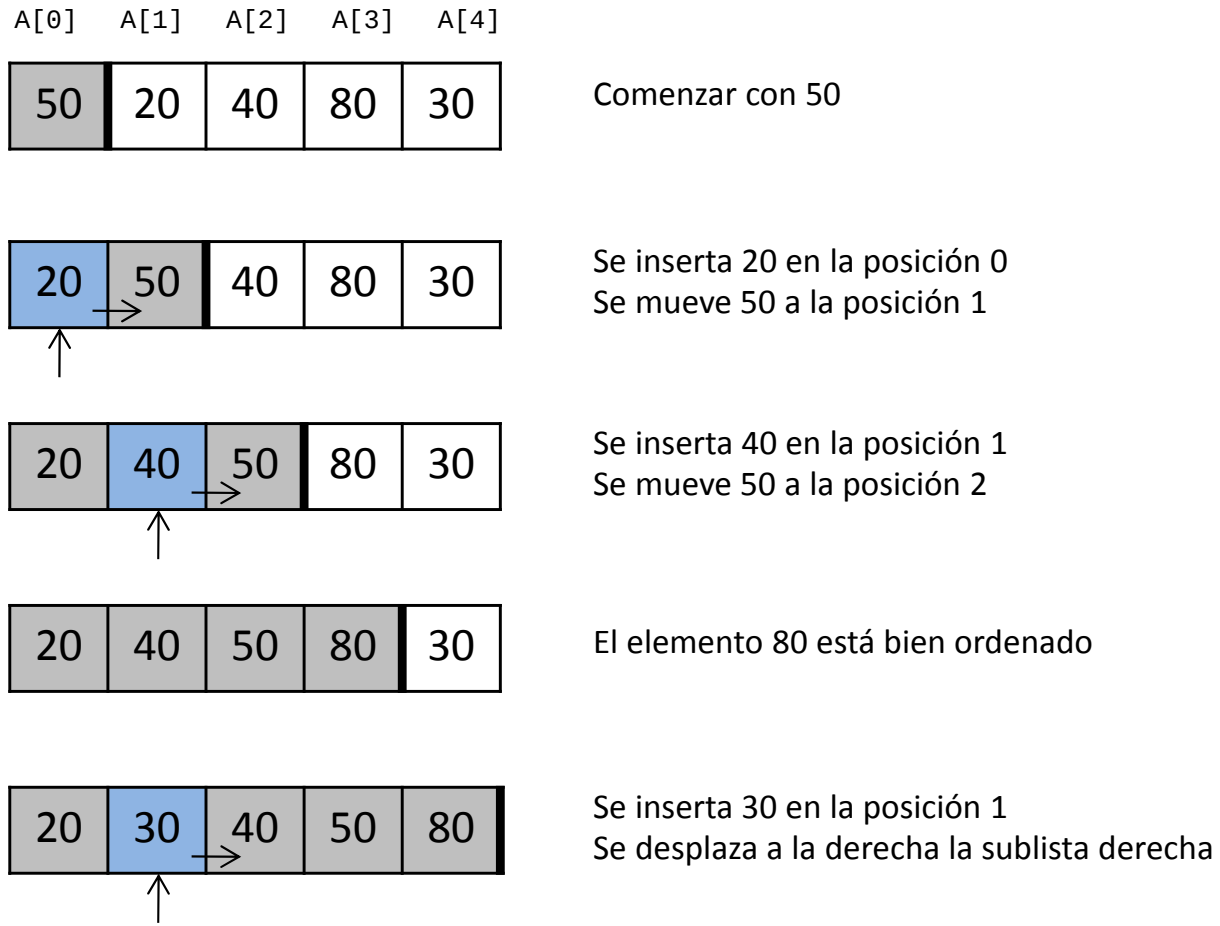
- El método de ordenación por inserción es similar al proceso típico de ordenar tarjetas de nombres (cartas de una baraja) por orden alfabético, que consiste en insertar un nombre en su posición correcta dentro de una lista o archivo que ya está ordenado.

- Los pasos del algoritmo son:
  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 que sea 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; a la vez se mueve 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 en la posición correcta.

6 5 3 1 8 7 2 4

2.4 Ordenación por inserción

- *Ejemplo*:



2.4 Ordenación por inserción

- Codificación del algoritmo de inserción:

$O(n^2)$

```
proc ordInsercion (a[]: ent, n: ent)
    i, j, aux: ent;
    para i: = 1 hasta n paso 1 hacer
        // índice j explora la sublista a[i-1]..a[1] buscando la
        // posición correcta del elemento destino, lo asigna a a[j]
        j := i;
        aux := a[i];
        // se localiza el punto de inserción explorando hacia abajo
        mientras (j > 0 and aux < a[j-1]) hacer
            // desplazar elementos hacia arriba para hacer espacio
            a[j] := a[j-1];
            j := j-1;
        fmientras
        a[j] := aux;
    fpara
fproc
```

3. Algoritmos de búsqueda

- Con mucha frecuencia los programadores trabajan con grandes cantidades de datos almacenados en arrays y registros, y por ello se hace necesario determinar si un array contiene un valor que coincida con un cierto valor clave.

- El proceso de encontrar un elemento específico de un array se denomina *búsqueda*. En esta sección se examinarán dos técnicas de búsqueda:
  - **Búsqueda secuencial** (o lineal), la técnica más sencilla.
  - **Búsqueda binaria** (o dicotómica), la técnica más eficiente.

3.1 Búsqueda secuencial

- En una búsqueda secuencial, los elementos de una lista o vector se exploran (se examinan) en secuencia, es decir, uno después de otro.

- La búsqueda secuencial busca un elemento de una lista utilizando un valor destino (***clave***). El algoritmo compara cada elemento del array con la clave.

- Dado que el array no está en un orden prefijado, es probable que el elemento a buscar pueda ser el primer elemento, el último elemento o cualquier otro.

- De promedio, al menos el programa tendrá que comparar la clave de búsqueda con la mitad de los elementos del array.

- El método de búsqueda lineal funciona bien con arrays pequeños o **no ordenados**.

3.1 Búsqueda secuencial

- Codificación del algoritmo de búsqueda secuencial:

$O(n)$

```
fun BusquedaSec(lista[]: ent, n: ent, elemento: ent) dev i: ent
    i: ent; encontrado: bool;
    i := 0; entontrado := falso;
    // Búsqueda en la lista hasta que se encuentre el elemento
    // o se alcance el final de la lista
    mientras ((!encontrado) and (i <= n-1)) hacer
        si (lista[i] = elemento) entonces
            encontrado := cierto;
        sino
            i := i + 1;
        fsi
    fmientras
    // Si se encuentra el elemento devuelve la posición en la lista
    si (encontrado) entonces
        retorno i;
    sino
        retorno (-1);
    fsi
ffun
```

3.2 Búsqueda binaria

- La búsqueda secuencial se aplica a cualquier lista. Si la lista está **ordenada**, la búsqueda binaria proporciona una técnica de búsqueda más eficiente.

- Una búsqueda binaria típica es la búsqueda de una palabra en un diccionario.

- Se sitúa la lectura en el centro de la lista y se comprueba si la clave coincide con el valor del elemento central. Si no se encuentra el valor de la clave, se sigue la búsqueda en la mitad inferior o superior del elemento central de la lista.

- En general, si los datos de la lista están ordenados se puede utilizar esa información para acortar el tiempo de búsqueda.

3.2 Búsqueda binaria

- Suponiendo que la lista está almacenada como un array, los índices de la lista son: `bajo = 0` y `alto = n-1` y `n` es el número de elementos del array, los pasos del algoritmo son:

  1. Calcular el índice del punto central del array

     `central = (bajo + alto)/2` (división entera)

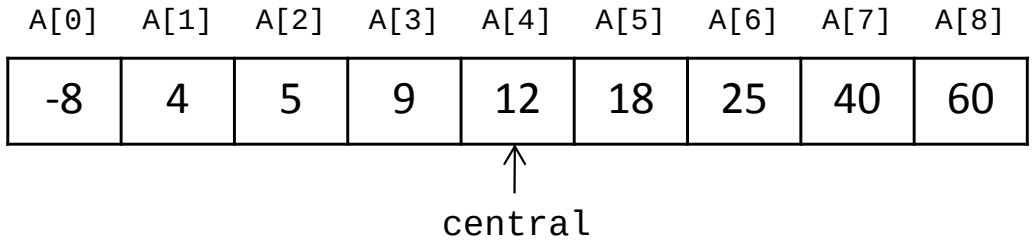
  2. Comparar el valor de este elemento central con la clave:
     - Si `a[central] < clave`, la nueva sublista de búsqueda tiene por valores extremos de su rango [`central+1,alto`].
     - Si `clave < a[central]`, la nueva sublista de búsqueda tiene por valores extremos de su rango [`bajo,central-1`].

  3. El algoritmo termina, bien porque se ha encontrado la clave o porque el valor de `bajo` excede a `alto` (*búsqueda no encontrada*).

3.2 Búsqueda binaria

- Ejemplo: Sea el array de enteros A (-8, 4, 5, 9, 12, 18, 25, 40, 60), buscar la clave, clave = 40.

  Paso 1

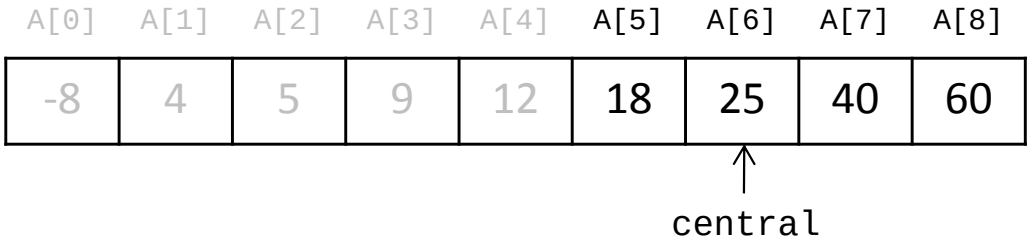


$$central = \frac{bajo + alto}{2} = \frac{0+8}{2} = 4$$

clave (**40**) > a[4] (**12**)

3.2 Búsqueda binaria

Paso 2: Buscar en sublista derecha



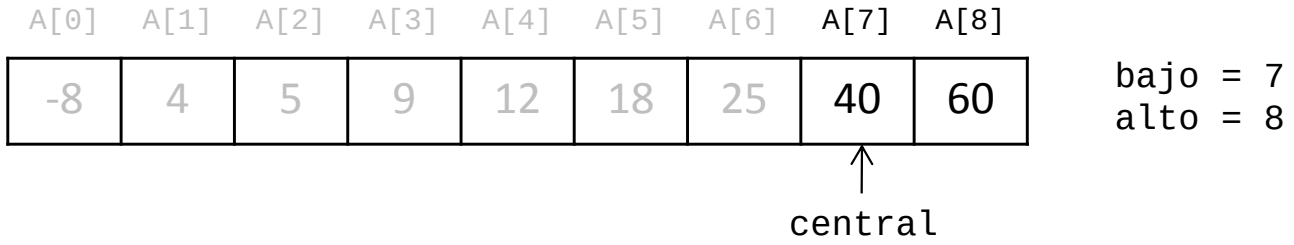

```
bajo = 5
alto = 8
```

$$central = \frac{bajo + alto}{2} = \frac{5+8}{2} = 6 \qquad (división\ entera)$$

```
clave (40) > a[6] (25)
```

3.2 Búsqueda binaria

Paso 3: Buscar en sublista derecha



$$central = \frac{bajo + alto}{2} = \frac{7+8}{2} = 7 \qquad (división\ entera)$$

`clave (40) = a[7] (40)` *(búsqueda con éxito)*

- El algoritmo ha requerido 3 comparaciones frente a 8 comparaciones ($n - 1 = 9 - 1 = 8$) que se hubieran realizado con la búsqueda secuencial.

3.2 Búsqueda binaria

- Codificación del algoritmo de búsqueda binaria:

$O(\log n)$

```
fun busquedaBin(lista[]: ent, n: ent, clave: ent) dev r : ent
    central, bajo, alto, valorCentral: ent;
    bajo := 1;
    alto := n;
    mientras (bajo <= alto) hacer
        central := (bajo + alto)/2;     // índice de elemento central
        valorCentral := lista[central];          // valor del índice central
        si (clave = valorCentral) entonces
            retorno central;            // encontrado, devuelve posición
        sino si (clave < valorCentral) entonces
            alto := central -1;         // ir a sublista inferior
        sino
            bajo := central + 1;        // ir a sublista superior
        fsi
    fmientras
    retorno -1;                         // elemento no encontrado
ffun
```

Bibliografía

- Martí, N., Ortega, Y., Verdejo, J.A. *Estructuras de datos y métodos algorítmicos*. Ejercicios resueltos. Pearson/Prentice Hall, 2003.

- Peña, R.; *Diseño de programas. Formalismo y abstracción*. Tercera edición. Prentice Hall, 2005.

(Estas transparencias se han realizado a partir de aquéllas desarrolladas por los profesores Clara Segura, Alberto Verdejo y Yolanda García de la UCM, y la bibliografía anterior)