Contenido

[**TEMA 4 ARBOLES** 2](#_Toc73634648)

[**1.** **DATOS CON ESTRUCTURA JERARQUICA. ARBOLES** 2](#_Toc73634649)

[**2. REPRESENTACION DE ARBOLES BINARIOS** 4](#_Toc73634650)

[**3. RECORRIDO DE ARBOLES BINARIOS** 5](#_Toc73634651)

[**4. ESTRUCTURA DE BÚSQUEDA, INSERCIÓN Y ELIMINACIÓN** 6](#_Toc73634652)

[**5. ARBOLES AVL (ARBOLES BALANCEADOS EN ALTURA)** 7](#_Toc73634653)

[**6. HEAP (MONTÍCULO)** 9](#_Toc73634654)

[**TEMA 5 GRAFOS** 10](#_Toc73634655)

[**1. GRAFOS DIRIGIDOS Y NO DIRIGIDOS** 10](#_Toc73634656)

[**2. DEFINICIONES BASICAS** 11](#_Toc73634657)

[**3. TAD GRAFO** 12](#_Toc73634658)

[**4. IMPLEMENTACIONES** 13](#_Toc73634659)

[**5. RECORRIDOS** 16](#_Toc73634660)

[**6. CONCEPTOS BASICOS** 18](#_Toc73634661)

[**7. OPERACIONES** 20](#_Toc73634662)

[**TEMA 6 AVANCE RAPIDO** 22](#_Toc73634663)

# **TEMA 4 ARBOLES**

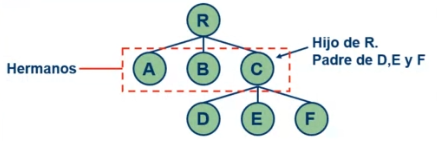
## **DATOS CON ESTRUCTURA JERARQUICA. ARBOLES**

Un árbol es una estructura jerárquica compuesta por una colección finita de objetos (nodo). Cumplen tres condiciones

* Existe un nodo que ocupa la primera posición (raíz)
* Un árbol vacío es un árbol sin nodos
* Un nodo es, por sí mismo, un árbol

Relaciones entre los nodos

* Hijo de: relación de un nodo con otro directamente superior (solo con el de encima, un hijo un padre)
* Padre de: relación de un nodo con otro directamente inferior (un padre muchos hijos)
* Hermano de: relación entre nodos que son hijos de un mismo padre



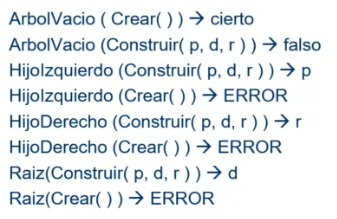
Definiciones importantes:

* Grado de un nodo: número de descendientes directos (nº hijos).
* Grado de un árbol: máximo grado que pueden llegar a tener sus nodos (nº max hijos).
* Nodo hoja: nodo que no tiene descendientes.
* Antecesores de un nodo X: camino desde la raíz hasta llegar a un nodo.
* Nivel de un nodo: lugar que ocupa un nodo dentro de la jerarquía (raíz = nivel 1).
* Profundidad de un árbol: nivel del nodo máximo.
* Bosque: conjunto de árboles que no comparten nodos entre sí.

**Arboles binarios** 🡪 cada padre, tiene como máximo dos hijos.

TAD Árbol Binario:

Defino su signatura (declaración teórica). Cuáles son los posibles, valores, operaciones y axiomas. Funciones posibles: (p y r = arbolbinario; d = elemento)

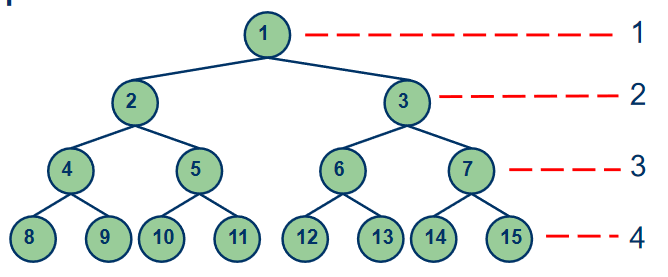


Características más importantes:

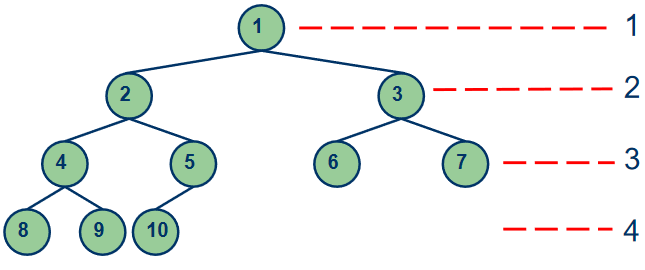
* El número máximo de nodos en el nivel ‘i’ de un árbol binario es 2i-1 para i >= 1.
* El número máximo de nodos de un árbol binario de profundidad k es 2k-1.

Definiciones importantes árbol binario:

* Árbol binario **lleno** de profundidad k, es aquel que tiene 2k-1 nodos.

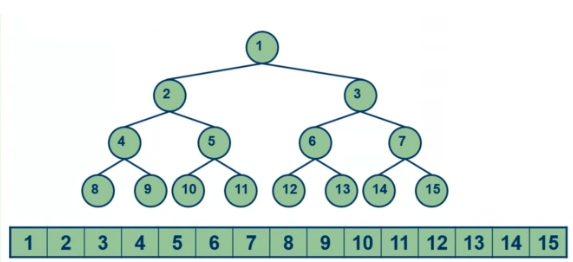


* Árbol binario **completo** de profundidad k, es aquel que está lleno hasta el nivel k-1, es decir, que tiene todos los nodos situados desde la izquierda sin huecos.



* Árbol binario **sesgado** en cada nodo solo existe un hijo

## **2. REPRESENTACION DE ARBOLES BINARIOS**



**REPRESENTACION SECUENCIAL**

Ventajas

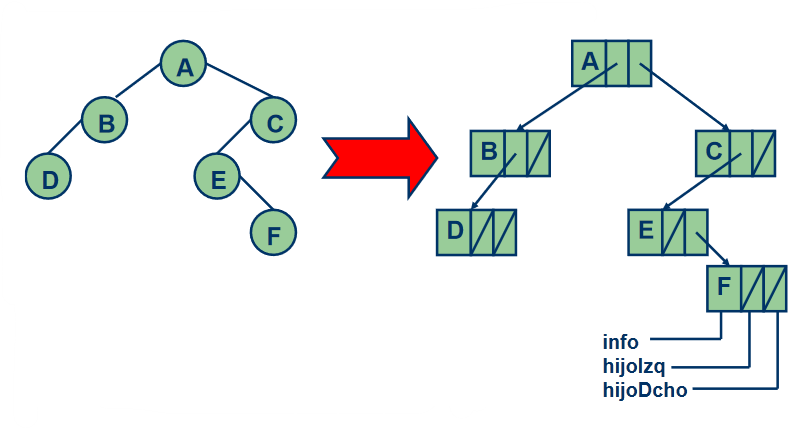
* Localizar padre 🡪 padre de i = (i/2) (salvo para i=1)
* Localizar hijo izquierdo 🡪 hijo izq de i = 2\*i
* Localizar hijo derecho 🡪 hijo der de i = 2\*i + 1

Inconvenientes

* Los elementos nulos se deben indicar como “elemento nulo”
* Como devolver el dato de la posición, como índice o como array

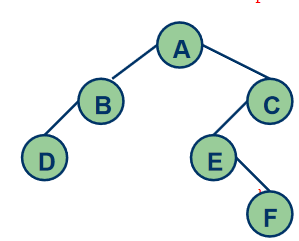
**REPRESENTACION DINAMICA**

Esta representación va a almacenar en cada nodo, el contenido de este elemento y sus dos hijos.

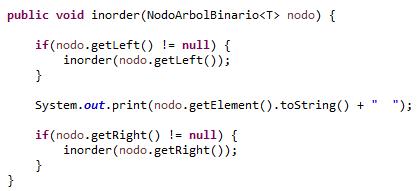


## **3. RECORRIDO DE ARBOLES BINARIOS**

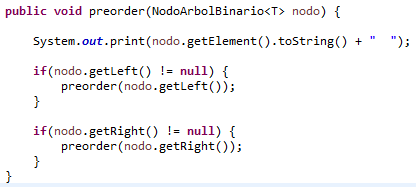
Existen varias maneras posibles de recorrer el nodo de un árbol:



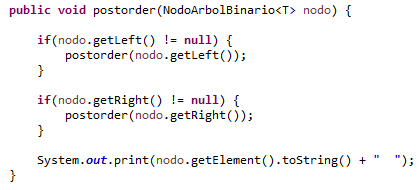
* **Recorrido inorden (IRD)** 🡪 primero recorro el hijo izquierdo, después la raíz y finalmente el hijo derecho. (D,B,A,E,F,C)



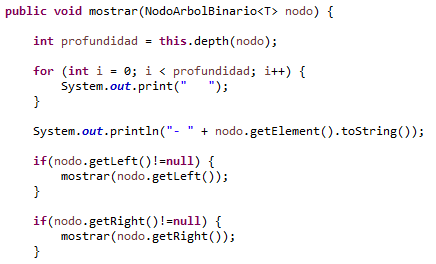
* **Recorrido preorden (RID)** 🡪 primero recorro la raíz, y después el hijo izquierdo y el derecho. (A,B,D,C,E,F)



* **Recorrido postorden (IDR)** 🡪 primero recorro el hijo izquierdo y el derecho y finalmente la raíz. (D,B,F,E,C,A)



* **Recorrido en anchura o forma jerárquica 🡪** recorro de la raíz al final con los nodos del mismo nivel de izquierda a derecha. (A,B,C,D,E,F)



## **4. ESTRUCTURA DE BÚSQUEDA, INSERCIÓN Y ELIMINACIÓN**

**Array**

* Para hacer la búsqueda de un elemento, tendremos que empezar desde la primera posición hasta el final, en el peor de los casos (O(n))
* Para insertar elementos, solo necesitamos saber la última posición para poder colocarlo en la siguiente, siempre va a ser de orden constante (O(1))
* Para eliminar un elemento, tendremos que mover todos los elementos de la derecha a este, una posición a la izquierda. Por lo que en el peor de los casos tendremos que mover todos los elementos (O(n))

**Linked list**

* Para hacer la búsqueda de un elemento, pasaría igual que en el array, por lo que el recorreremos todos los elementos (O(n))
* Para insertar elementos, solo necesitamos la posición, por lo que el orden es constante (O(n))
* Para eliminar un elemento, recorremos la lista entera buscando el elemento para poder eliminarlo (O(n))

Para reducir el tiempo de búsqueda, podemos trabajar con **arrays o listas ordenadas**. De este modo los tiempos serás:

* El tiempo de búsqueda se reducirá a O(n\*log n)
* El tiempo de la inserción aumentara a O(n), ya que tendremos que insertar el nuevo elemento en la posición indicada para que el array siga ordenado.
* El tiempo de eliminación seguirán siendo el mismo (O(n)).

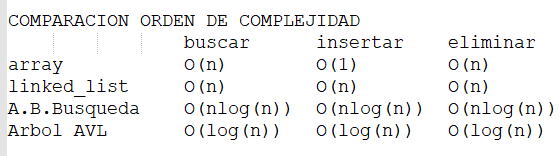
Si queremos reducir los tres tiempos para tener una manera más eficaz, tendremos que utilizar un **árbol ordenado binario** (BST), que los tiempo de búsqueda, inserción y eliminación van a ser de orden O(n\*log n).

Un árbol binario ordenado se caracteriza por tener como subnodo izquierdo un elemento más pequeño, y como subnodo derecho uno mayor.

Para realizar las búsquedas, vamos a comparar con la raíz del árbol, para saber si es mayor o menor, y buscar en la parte correspondiente. Haremos esto sucesivamente con cada subnodo hasta llegar al elemento que queremos buscar.

Para insertar elementos, vamos a comparar el elemento con la raíz, y dependiendo de si es mayor o menos, seguiremos recorriendo el árbol por la rama correspondiente. Hasta llegar al último nodo, donde lo insertaremos a la izquierda o derecha, según nos indique la comparación.

Para hacer la eliminación de un nodo, tenemos varias posibilidades:

* Sustituirlo por un nodo hoja, es decir, uno de sus subnodos que no tenga hijos
* Sustituirlo por el valor mínimo del subárbol derecho
* Sustituirlo por el valor máximo del subárbol izquierdo.

## **5. ARBOLES AVL (ARBOLES BALANCEADOS EN ALTURA)**

Este tipo de árboles, tienen una serie de características que permiten reducir el orden de complejidad de búsqueda, inserción y eliminación a O(log n). Se utiliza para realizar las mismas aplicaciones que un árbol binario de búsqueda, peor con el orden reducido.

El **factor de equilibrio (fe)** es la diferencia entre la altura del hijo izquierdo y del hijo derecho de un nodo. Fe = (Nodos hijo izquierdo – Nodos hijo derecho)

Un **árbol AVL** es un árbol binario de búsqueda en el que ambos hijos son también AVL y la diferencia entre sus alturas es menos o igual que 1 (fe <= 1)

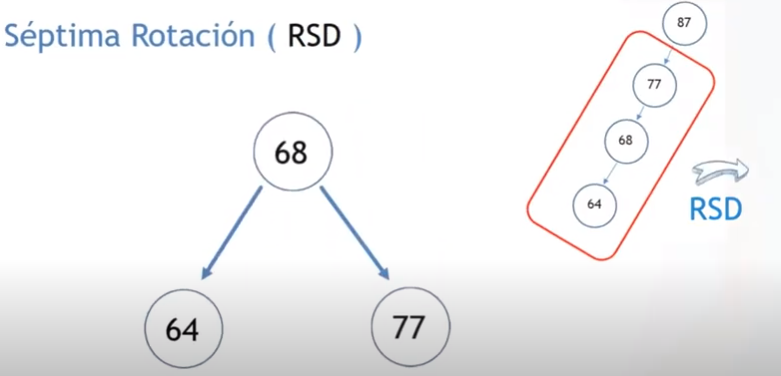
En cuanto a las operaciones que se pueden realizar con este tipo de árboles, tenemos la búsqueda, que similar a la búsqueda en arboles binarios. Sin embargo, si queremos insertar o eliminar un elemento el árbol AVL puede perder su equilibrio, por lo que habrá que recuperar este mediante las rotaciones. Estas rotaciones deben cumplir las condiciones de un árbol binario de búsqueda (menores que un nodo a la izquierda, mayores a la derecha).

* Rotación Izquierda – Izquierda simple 🡪 nodo insertado/eliminado es hijo izquierdo subárbol izquierdo.
* Rotación Derecha – Derecha simple 🡪 nodo insertado/eliminado es hijo derecho subárbol derecho.
* Rotación Izquierda – Derecha doble 🡪 nodo insertado/eliminado es hijo derecho subárbol izquierdo.
* Rotación Derecha – Izquierda doble 🡪 nodo insertado/eliminado es hijo izquierdo subárbol derecho.
* Rotación X – X simple 🡪 nodo insertado/eliminado es hijo X subárbol X.
* Rotación X – Y doble 🡪 nodo insertado/eliminado es hijo Y subárbol X.

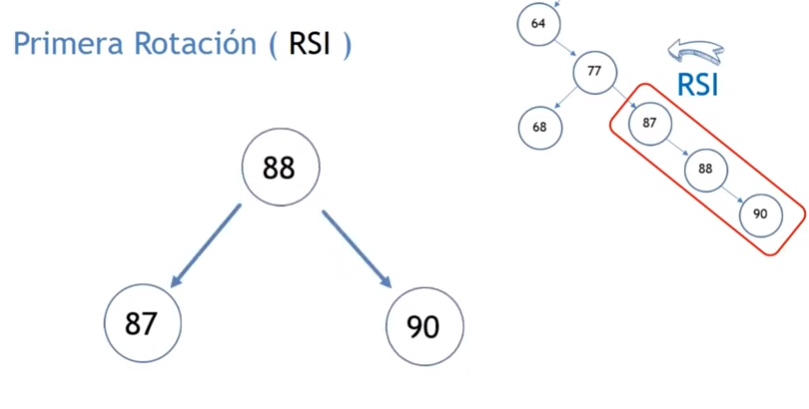
Desde el nodo donde se desequilibra el factor de carga, seguir primeros dos caminos para llegar al nodo insertado. Si hay mas nodos izquierdos, que derechos, coger hijo izquierdo para rotar, sino, coger hijo derecho.

**Rotaciones simples y compuestas**

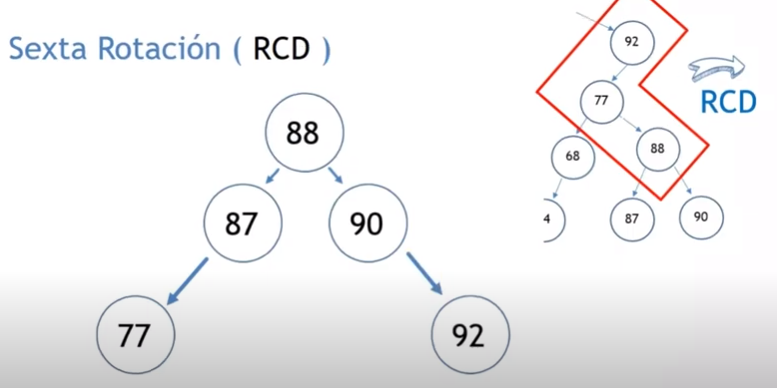
* Rotación simple a la derecha



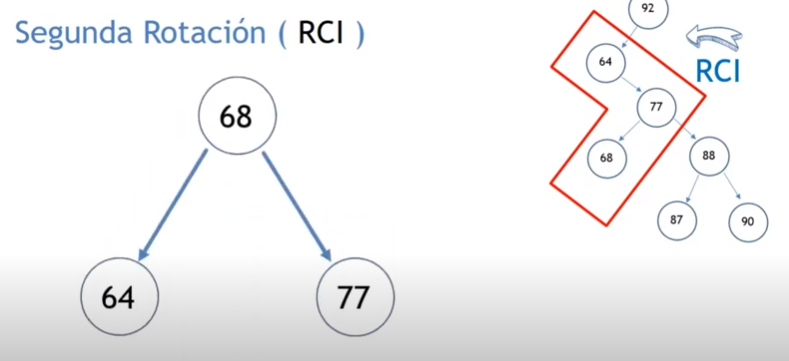
* Rotación simple a la izquierda



* Rotación compuesta a la derecha



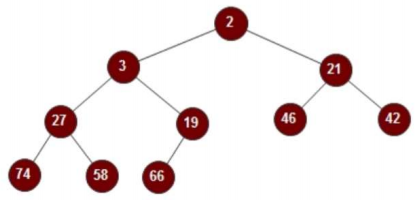
* Rotación compuesta a la izquierda



## **6. HEAP (MONTÍCULO)**

Un montículo es un árbol binario completo (Esta lleno hasta el penúltimo nivel, y en el último tiene al menos el nodo izquierdo) que es un heap o tiene la propiedad del heap si el valor de la raíz es menor (tiene mayor prioridad) que sus hijos, y estos a su vez son montículos también.

Explicación 🡪 árbol binario ordenado de tal forma que la raíz sea el número más pequeño y cuanto más nos alejamos de esta, mayores son los valores de los nodos.



Los beneficios que obtenemos al utilizar los montículos (heap) son la rapidez para extraer o insertar elementos en el árbol, con un orden de complejidad O(log n). El problema que obtenemos es que no garantizamos que después de extraer o insertar el elemento, el árbol cumpla las condiciones del montículo. Estas dos operaciones, van a suponer un máximo de n cambios (n = profundidad del árbol), por lo que su orden de complejidad será O(log n).

**Extracción de elementos**

Siempre se extrae la raíz del árbol y se sustituye por el elemento que este en el último nodo más a la derecha. Después realizamos los cambios necesarios para poder cumplir las condiciones de los montículos.

**Inserción de elementos**

El elemento se inserta en el nodo que este libre más a la derecha. Después, para cumplir las condiciones, subiremos el elemento intercambiándolo por los elementos más grandes de tal manera que este elemento se quede en la posición correspondiente.

**HEAPSORT**

Heapsort es un algoritmo de ordenación de listas. Este método tiene dos posibles soluciones:

* Extraer todos los elementos de la lista y añadirlos al heap (conservando las prioridades heap)
* Extraer todos los elementos del heap de forma ordenada e introducirlos en la lista, para después poder ordenarlos.

Este algoritmo tiene un orden de complejidad O(n log n), ya que voy a realizar n inserciones o eliminaciones, con complejidad O(log n).

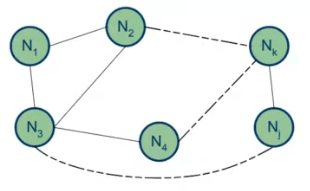
# **TEMA 5 GRAFOS**

## **1. GRAFOS DIRIGIDOS Y NO DIRIGIDOS**

Hemos visto hasta el momento:

* Tipos de datos lineales: un elemento siguiente y un elemento anterior (lista)
* Tipos de datos no lineales: más de un elemento siguiente, pero solo un elemento anterior (arboles)

Los grafos son estructuras que permiten que un elemento tenga uno o varios elementos anteriores y siguientes. Además, nos permiten representar relaciones binarias entre elementos de un conjunto.



**Grafos dirigidos**

Un grafo dirigido (GD) se caracteriza por la dirección de las flechas que unen los nodos, y se define como un par (V, A):

* V es un conjunto de elementos (nodos o vértices)
* A es una relación binaria entre elementos (arcos o aristas)
* Cada par (x, y) de A es un par ordenado con x, y pertenecientes a V (X == Y && X =! Y)

**Grafos no dirigidos**

Un grafo no dirigido (GND) se caracteriza por no tener una dirección definida entre los nodos, y se define como un par (V, A):

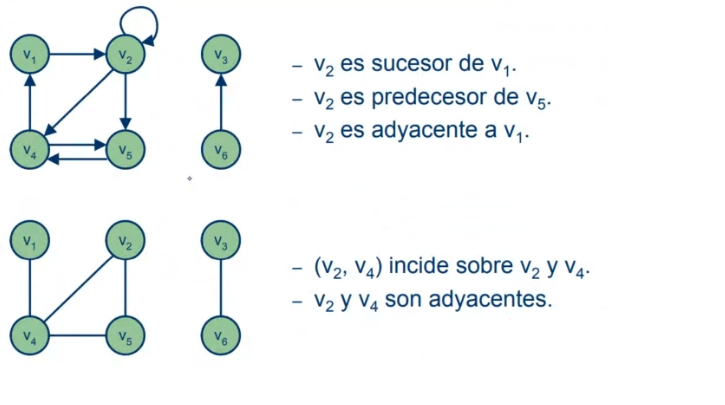
* V es un conjunto de elementos (nodos o vértices)
* A es una relación binaria entre elementos (arcos o aristas)
* Cada par (x, y) de A es un par no ordenado con x, y pertenecientes a V (X =! Y)

**Grafos etiquetados**

Son grafos que tienen información en las aristas

## **2. DEFINICIONES BASICAS**

**Predecesor y sucesor**: se da en grafos dirigidos. En una relación entre nodos (a, b) decimos que sale de ‘a’ y llega a ‘b’. (‘b’ es sucesor de ‘a’; ‘a’ es predecesor de ‘b’) Estas relaciones también se puede conocer como adyacentes (‘b’ es adyacente de ‘a’). En el caso de que trabajemos sobre grafos no dirigidos, decimos que el grafo incide sobre ‘a’ y ‘b’. Estas relaciones también se denominan adyacentes (‘b’ es adyacente de ‘a’; ‘a’ es adyacente de ‘b’).



**Grado de un vértice**:

* Para GND: número de arcos que inciden sobre él.
* Para GD: número de arcos que salen de él, más el número de arcos que llegan a él.

El **grado de un grafo** es el grado del vértice con mayor grado

**Camino**: recorrido que se puede seguir para llegar de un vértice a otro vértice de un grafo, siguiendo los arcos según su sentido. La longitud del camino es el número de arcos entre los vértices. Si existe un camino ‘p’ entre dos vértices (a, b), decimos que ‘b’ es **alcanzable** desde ‘a’ vía ‘p’.

Un camino puede ser **simple** si no se pasa dos veces por un mismo vértice.

Un camino (GD) forma un **ciclo** si todos los vértices son distintos, y el primero y ultimo son el mismo. Un **bucle** se forma cuando un vértice se une consigo mismo.

**Cadena**: es igual que un camino pero sin tener en cuenta la dirección de los arcos (GD). Esta cadena podrá formar un **circuito** cuando el primer y último vértice sean el mismo y contenga al menos un arco.

## **3. TAD GRAFO**

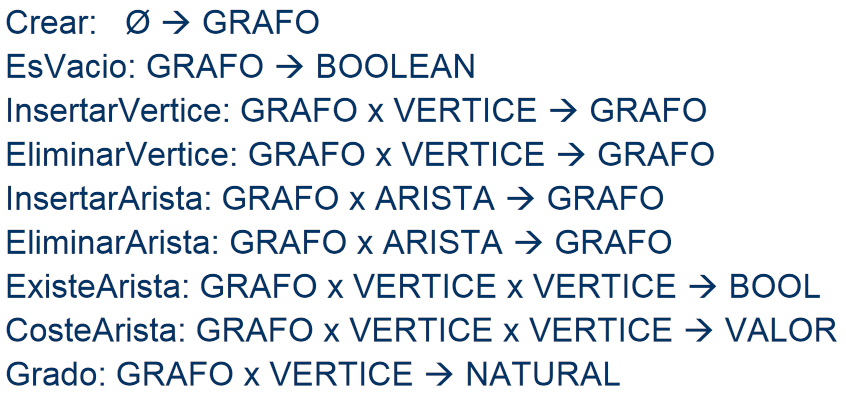
Un grafo consta de un conjunto de vértices, un conjunto de arcos y las operaciones que se puedan realizar sobre el conjunto. Operaciones posibles:

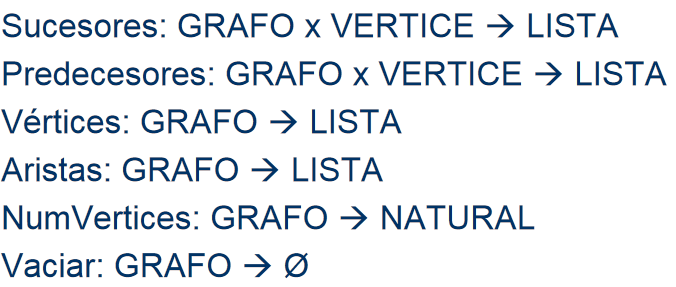
* Añadir o eliminar vértices y aristas
* Comprobar si está vacío el grafo
* Comprobar si hay un elemento en el grafo

Usaremos dos tipos de elementos nuevos:

* Un vértice: objeto compuesto de un identificador y opcionalmente un contenido
* Un arco: objeto compuesto de dos vértices y opcionalmente un peso (relación)

Funciones posibles:

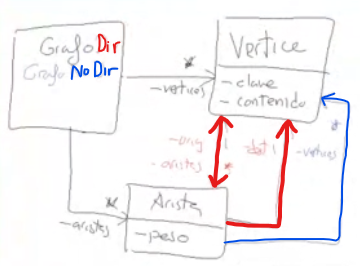




## **4. IMPLEMENTACIONES**

**Con listas de adyacencia**

En este caso tendremos dos listas, una de vértices y otra de aristas. Estas apuntaran al vértice de origen y al de destino (GD) y a una lista de vértices (GND)



* Clase Grafo

Un grafo está compuesto por ‘n’ vértices y ‘n’ aristas.

* Clase Vértice y Clase Arista
  + GD (rojo)

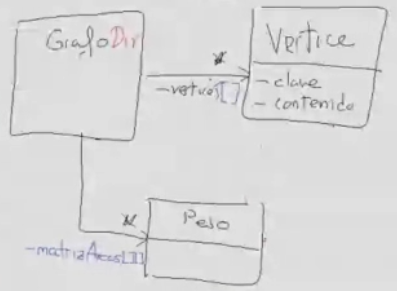
Un vértice tiene una lista de aristas, ya que de un vértice pueden salir ‘n’ aristas. Además, una arista, tiene un vértice origen y un vértice destino.

* + GND (azul)

Una arista tiene una lista de vértices.

**Con matriz de adyacencia**

En este caso tenemos una lista de vértices y una matriz de aristas. Esta matriz de aristas va a almacenar el peso asociado a la relación entre vértices (si existe, sino NULL).



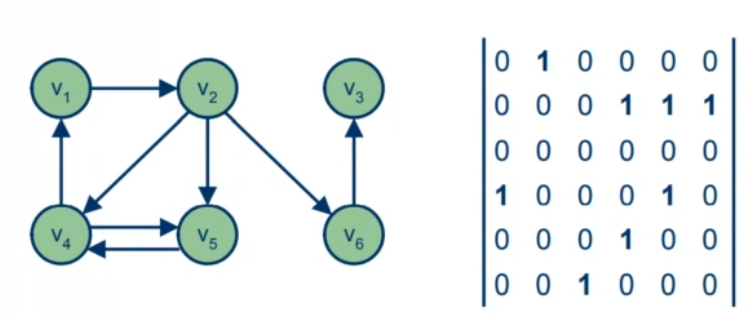
Clase grafo

Esta clase contendrá un array de vértices y una matriz con el peso de cada relación entre vértices. (matriz de adyacencia)

**MATRIZ DE ADYACENCIA**

VÉRTICES DESTINO

V1 V2 V3 V4 V5 V6



VÉRTICES ORIGEN

V1 V2 V3 V4 V5 V6

**Obtener sucesores y predecesores de un vértice (pseudocódigo)**

**obtenerSucesores**( Grafo g, Vértice v) : ListaVertices <vértices>

i = **buscarPosicion**( g, v) 🡪 en la matriz de adyacencia

para j entre 1 y **numeroVertices**(g) repetir

si g.matrizAdyacencia(i, j) > 0 🡪 recorro la fila del vértice

**insertar**(ListaVertices, j) 🡪 (si hay arista entre ‘i’ y ‘j’)

devolver ListaVertices

**obtenerPredecesores**( Grafo g, Vértice v) : ListaVertices <vértices>

i = **buscarPosicion**( g, v) 🡪 en la matriz de adyacencia

para j entre 1 y **numeroVertices**(g) repetir

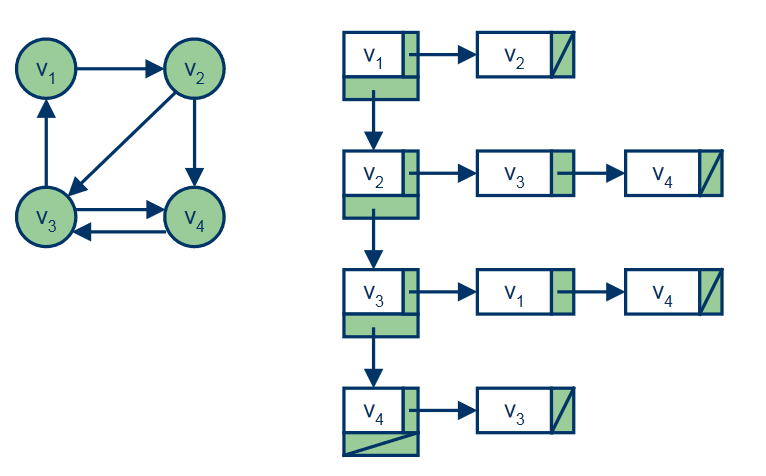
si g.matrizAdyacencia(j, i) > 0 🡪 recorro la columna del vértice

**insertar**(ListaVertices, j) 🡪 (si hay arista entre ‘j’ y ‘i’)

devolver ListaVertices

**REPRESETNACION DE LISTAS DE ADYACENCIA**

La lista de adyacencia está formada por todos los vértices del grafo. En cada nodo de la representación almacenaremos el vértice, la lista de vértices adyacentes y el valor que tenga la arista que une los dos vértices.



Funciones posibles con grafos:

Grado del grafo 🡪 el máximo de los grados de los vértices (entrada + salida)

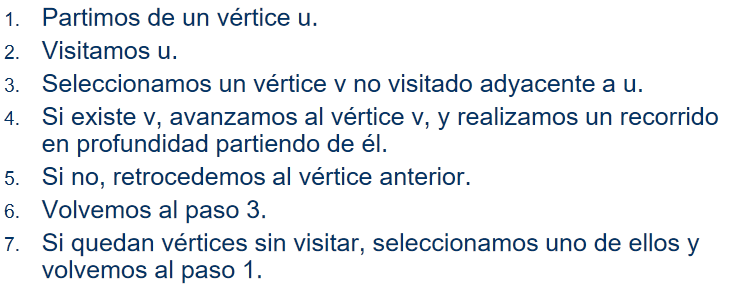
esArbol() 🡪 cada vértice tenga solo un predecesor y varios sucesores. Además, mi vértice no puede estar en la lista de sucesores de otros vértices

esFuente() 🡪 un nodo del que solo salen aristas

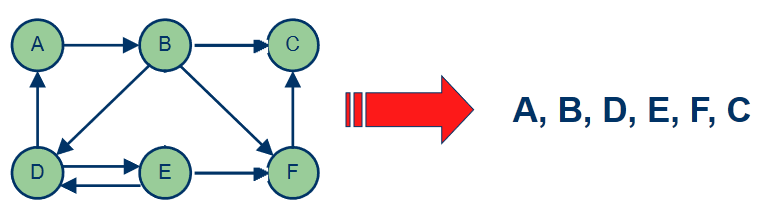
esSumidero() 🡪 un nodo del que solo entran aristas

## **5. RECORRIDOS**

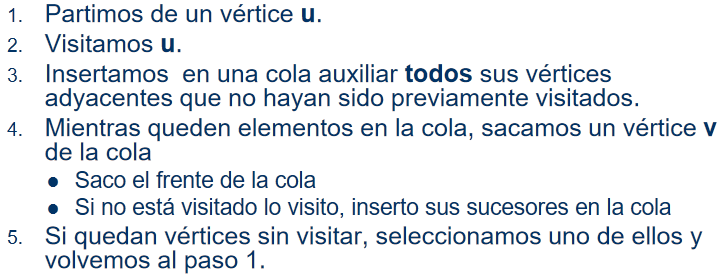
**Recorrido en profundidad**: desde un vértice, voy recorriendo todos los demás vértices sin repetirse.



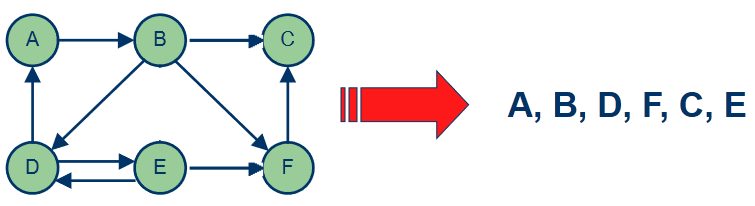
Los vértices visitados se deben de añadir a un conjunto de vértices visitados para recorrer el grafo una sola vez.



**Recorrido en anchura**: selecciono un vértice y visito todos sus adyacentes, y después los adyacentes de los adyacentes. Es necesario hacer uso de una cola, donde se registran los vértices adyacentes no visitados que debo visitar.

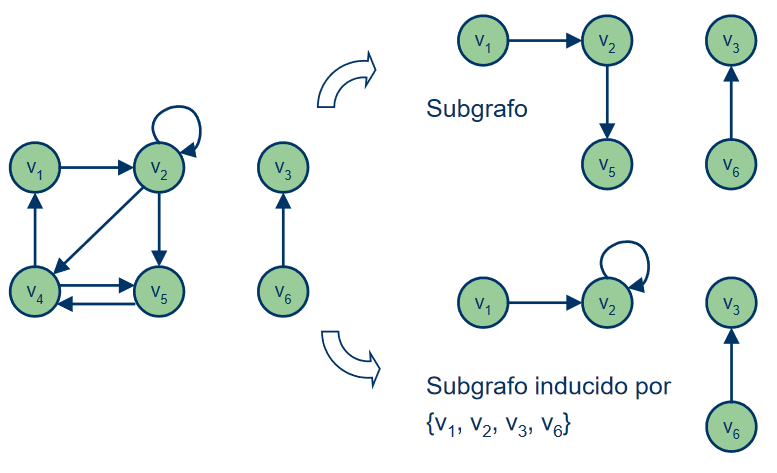


Los vértices visitados se deben de añadir a un conjunto de vértices visitados para recorrer el grafo una sola vez.



## **6. CONCEPTOS BASICOS**

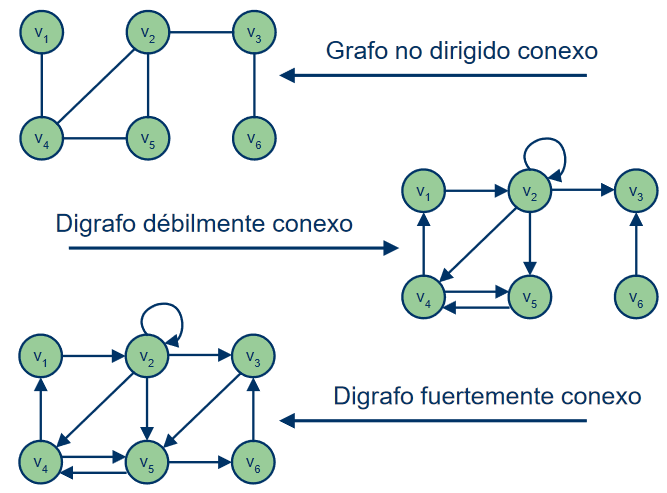
Un **subgrafo** es un subconjunto de un grafo, en el que los vértices y las aristas del subgrafo perteneces al grafo. El **subgrafo inducido** es el que conserva todas las conexiones entre el conjunto de vértices correspondiente.

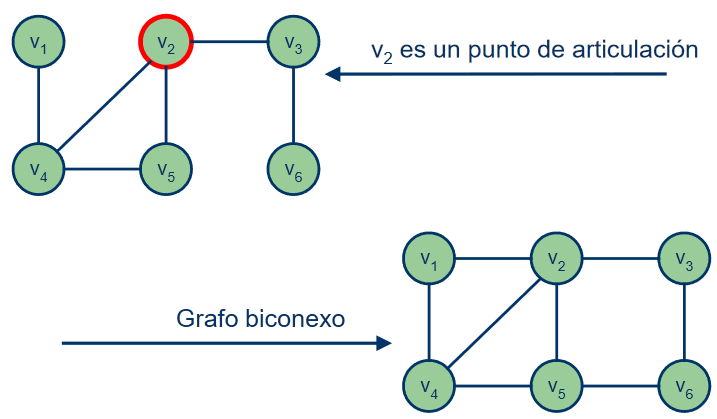


**Conectividad**: un g.n.d. es conexo si existe un camino entre cualquier par de vértices del grafo (desde un vértice puedo llegar a cualquier otro mediante aristas).

* **Fuertemente conexo**: existe camino entre cualquier par de vértices
* **Débilmente conexo**: existe una cadena entre cualquier par de vértices.
* **Vértice aislado**: no tiene conexión con ninguna conexión con otro vértice.
* **Grafo completo**: hay una arista entre cualquier par de vértices.

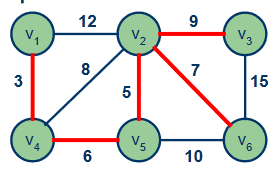
Los **puntos de articulación** que se utilizan para convertir un grafo no conexo a un grafo conexo, es decir, si quito un vértice, el grafo deja de ser conexo.

Un **grafo biconexo** es un grafo que no tiene puntos de articulación.

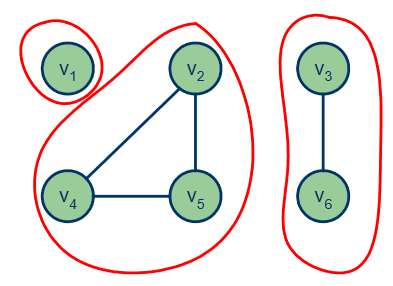


Un **árbol de recubrimiento**: es una estructura que me permite llegar a todos los vértices a través de aristas, y debe ser un árbol, es decir que no llegue a un mismo vértice desde dos vértices diferentes (sin ciclos).

Un **árbol de recubrimiento mínimo**: es un árbol de recubrimiento donde el camino que sigo tiene el peso mínimo en la suma de los pesos de las aristas. (Digital Block 6)



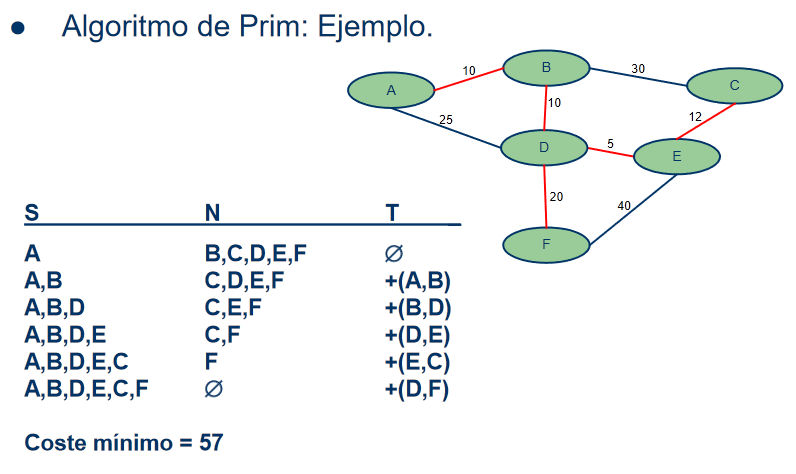
Las **componentes conexas** son los subgrafos inducidos por un subconjunto de vértices que son conexos también. Este grafo no es conexo, pero tiene tres componentes conexas:



## **7. OPERACIONES**

**ALGORITMO DE PRIM**

Elegimos un vértice de salida, voy recorriendo el camino siguiendo las aristas cuyo peso sea menor. Cada vértice solo se puede visitar una vez, y el número de aristas totales serán el número de vértices – 1, cuando esto ocurra (vértices – 1 = arista) se acabará el algoritmo.



# 

# **TEMA 6 AVANCE RAPIDO**

**1. PLANTEAMIENTO GENERAL**

Se utiliza para resolver problemas de una forma más optimizada. Los algoritmos de avance rápido seleccionan en cada paso, la opción más beneficiosa, hasta alcanzar la solución del problema.

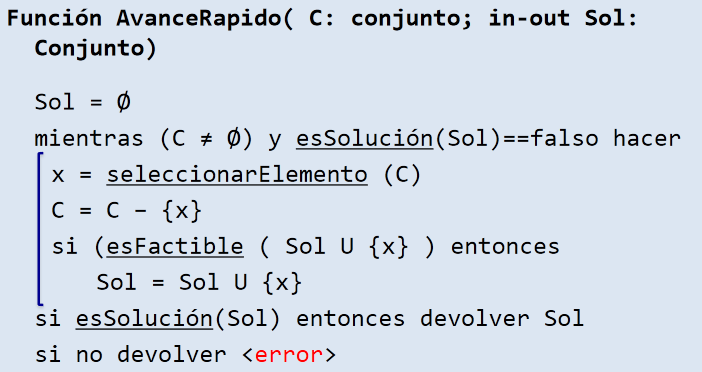
Supongamos que tenemos un conjunto de datos de entrada y una función objetivo (a maximizar o minimizar). El objetivo es encontrar un subconjunto de datos que optimice la función objetivo, satisfaciendo una serie de condiciones adicionales.

Debemos distinguir entre dos posibles soluciones:

* Solución factible: soluciones que son posibles para nuestro problema. El subconjunto de datos satisface las restricciones del problema
* Solución óptima: son soluciones que realmente optimizan la función objetivo. Es la solución factible para la cual la función objetivo devuelve un valor óptimo.

Estrategia a seguir en cualquier algoritmo de avance rápido:

Se seleccionar el elemento **X** más prometedor de los disponibles. Si la solución con este elemento **X** sigue siendo las más factible, se añade **X** a la solución, sino lo ignoro.



La Solución es el conjunto vacío.

Mientras el conjunto de entrada siga teniendo elementos y no hayamos conseguido aun la solución.

Seccionamos un elemento del conjunto de datos (X) (este elemento es el más prometedor).

Quitamos el elemento X del conjunto de datos

Añadimos el elemento al conjunto solución.

Devolvemos el conjunto solución o un error (cuando llegamos al conjunto vacío sin encontrar una solución).

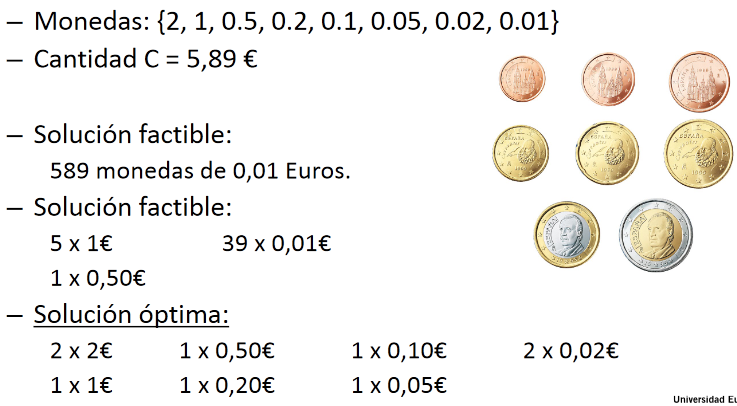
Complejidad de AvanceRapido

* En cada paso, se ejecutan tres funciones (), y la mas compleja de ellas determinara el orden de complejidad final de la funion
* Otro factor que afecta a la complejidad, es el orden inicial de los elementos de entrada.
* El avance rapido es mas rapido que otros enfoques mientras sea aplicable, excepto, en algunos casos, que el DyV es mas eficaz.

**ALGORITMO DEL CAMBIO MÍNIMO**

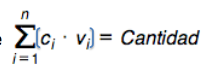
**Ejemplo 1**

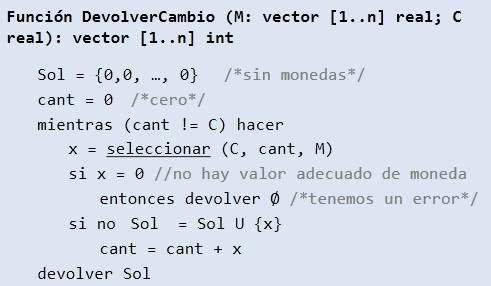
Tenemos un sistema monetario (M) formado por en conjunto de monedas con un valor distinto {V1, V2… Vn}. Además, de la cantidad que se quiere devolver (C). Se busca devolver la cantidad C utilizando el menor numero de monedas posibles (las monedas de todos los tipos son infinitas)



Para represetnar la solucion, crearemos un vector con la cantidad de monedas que selecciono de cada valor. {2, 1, 1, 1, 1, 1, 2, 1} (dos de 2€, una de 1€, una de 0.5€, una de 0.2€, una de 0.1€, una de 0.05€, 2 de 0.02€, una de 0.01€)

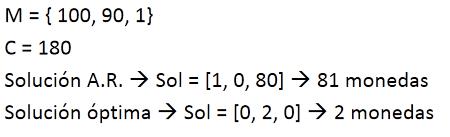
Para representar el objetivo, debemos minimizar el sumatorio de monedas, teniendo en cuenta que el resultado sumatorio debe ser igual a la cantidad C.





El algoritmo se basa en la función seleccionar, que devuelve el valor Vide la moneda mayor que aún no supere la cantidad que falte por devolver: Vi + cant <= C. Si no se encuentra el valor Vi, el sistema da error porque no hay solucion posible.

El algoritmo de cambio mínimo no siempre encuntra una solucion optima, como en el siguiente ejemplo:



Como el primer valor más grande sin pasarse de la cantidad es 100, después me obliga a coger 80 monedas de 1. Sin embargo, la solución óptima seria devolver solo 2 monedas de 90.

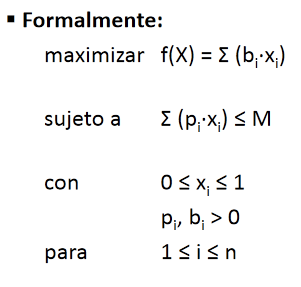
**ALGORITMO DE LA MOCHILA**

Problema: Tenemos n objetos (xi), con un peso (pi) cada uno de ellos, y un beneficio asociado (bi).

Objetivo: llenar la mochila maximizando el beneficio de los objetos seleccionados, respetando su capacidad máxima. Además, los objetos se pueden fraccionar (introducir trozos del objeto).

Solución: vector que en cada posición almacena un numero entre 0 y 1 del objeto que debe meterse (porcentaje del objeto que meto en la mochila).

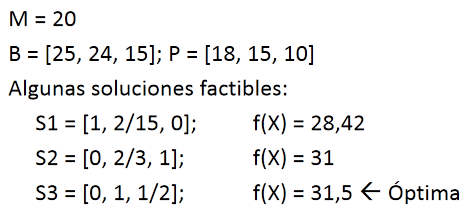
La solución óptima cumplirá que Σ (pi·xi) = M, la mochila se llena por completo.



Ejemplo 1

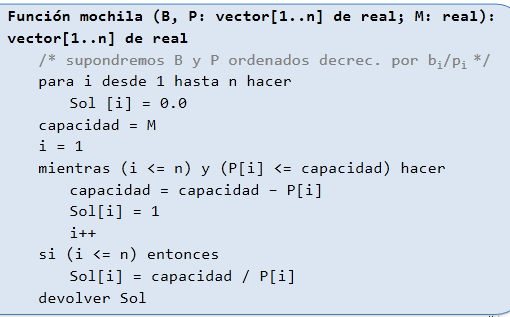
F(x) es el beneficio.

S: solución, fracción que selecciono de cada parte (P)



La función optima es la que mayor beneficio tenga.

* S1 🡪 seleccionar en cada caso el **mayor beneficio**
* S2 🡪 seleccionar en cada caso el **menor peso**
* S3 🡪 seleccionar en cada caso la relación **mayor beneficio/peso**.



**Complejidad:**

* Peor caso: O(n) si tomamos el “mientras” como referencia
* Mejor caso: O(1)
* Considerando la ordenación previa: O(QuickSort) · n

