* Introducción a las estrategias de diseño de algoritmos:

Un algoritmo es una serie de pasos a seguir para resolver un problema.

Los algoritmos siguen siempre una estrategia.

- Tipos de estrategias:

Recursividad: Consiste en solucionar un problema, resolviendo versiones mas pequeñas del mismo.

Es una técnica alternativa a la iteración, la cual:

- Consume más tiempo y memoria.
- Aporta la ventaja de que modela mejor los problemas definidos recursivamente.

Cualquier algoritmo recursivo se puede sobreescribir como un algoritmo iterativo.

Si entendemos como dividir un problema grande en problemas mas pequeños, la recursividad es más natural y facil de entender.

Los algoritmos recursivos siempre estan parametrizados.

Para que un problema se pueda resolver recursivamente deben cumplirse dos condiciones:

- Caso base: Debe existir un valor de los parámetros para el que no se produzca una llamada recursiva, es decir, que se pueda parar la ejecución del algoritmo.
- Monotonamente decreciente: En cada llamada, los parametros se deben acercar al caso base.

Tipos de recursividad:

Simple: Cuando el procedimiento se llama a si mismo.

Indirecta o mutua: Cuando un procedimiento A() llama a otro procedimiento B() y este, a su vez, llama al procedimiento A().

- Divide y conquista:

Esta estrategia se basa en dividir el problema y los recursos en problemas mas pequeños independientes.
Consiste en descomponer el problema en subproblemas mas pequeños, resolver independientemente los subproblemas y por ultimo, combinar las soluciones de los subproblemas para obtener la solución al problema original.

Tres condiciones básicas:

- Subalgoritmo básico: Para problemas pequeños debe existir un subalgoritmo básico que resolva el problema sin descomponerlo mas.
- Descomposición en subproblemas: Debe ser posible descomponer el problema en dos o mas subproblemas.
- Subproblemas similares: Los subproblemas deben ser similares y de tamaño similar.

Otras estrategias:

- Programación dinámica:

Puede ocurrir que en la estrategia divide y conquista, la división del problema nos de subproblemas repetidos (Subproblemas solapados). Si se resuelven los subproblemas sin tener en cuenta estas repeticiones, el algoritmo se vuelve ineficiente.
La idea de la programación dinámica es no resolver dos veces el mismo subproblema, rellenando una tabla con la solución cada vez que resolvemos un subproblema.

- Algoritmos ávidos (greedy algorithms):

Toman decisiones a corto plazo basadas en información inmediatamente disponible y sin importar consecuencias futuras.
Se emplean para encontrar las mejores combinaciones en problemas de optimización.

- Componentes:

Conjunto de candidatos: Se usan para ir creando la solución.
Solución parcial: Candidatos ya elegidos para formar parte de la solución final.

Además incluyen cuatro funciones:
- Función de viabilidad: Indica si un candidato se puede usar para contribuir a la solución.
- Función de selección: Elige al mejor candidato para añadir a la solución de cada paso.
- Función de coste: Asigna un valor a la solución actual.
- Función de solución: Indica cuando se ha alcanzado la solución completa.

Un algoritmo ávido nunca reconsidera un candidato cuando es descartado por la función de viabilidad.

-Método de retroceso (Backtracking):
Solo se usa para resolver problemas combinacionales.
Realiza una búsqueda sistemática de soluciones, generando un árbol de búsqueda en el que cada nivel representa a un candidato que se añade a la solución.
La búsqueda sistemática es muy costosa, con lo que el backtracking añade dos optimizaciones:
- Poda: Cuando la función de viabilidad detecta un camino que no conduce a solución, deja de evaluar esa parte del árbol.
A este abandono y retroceso se le llama backtracking.
- Heurísticas: Son reglas que sirven para decidir cuales son las ramas mas prometedoras y evaluarlas primero.

-Ramificación y poda (branch and bound):
Es similar al backtracking, pero solo se usan para resolver problemas de optimización, en los que al rebasar la función objetivo sabemos que no podemos seguir.
- Ramificación: La búsqueda se realiza en anchura.
- Poda: La función objetivo da un valor máximo y mínimo para un camino. Dados dos caminos A y B, la poda se realiza en el camino B, cuando el valor mínimo de B es mayor al máximo de A.

*Eficiencia de algoritmos:

El objetivo es medir el rendimiento de un algoritmo.
Vamos a trabajar con diferentes métricas para valorar la eficiencia de los algoritmos.

-Medidas de eficiencia:

- Análisis empírico: Consiste en programar varios algoritmos y ejecutarlos para ver lo que tarda cada uno.
- Análisis matemático: Consiste en determinar matemáticamente la cantidad de recursos que necesita el algoritmo en función de los datos de entrada.

En el análisis matemático existen dos formas de medir los recursos consumidos:

- Eficiencia temporal: Indica lo rapido que se ejecuta el algoritmo.
  La eficiencia temporal depende del volumen de datos de entrada, de los cores del sistema, etc.
  Nosotros supondremos que existe un solo core.
- Eficiencia espacial: Indica la cantidad de memoria que consume el algoritmo.

Actualmente, la selección de un algoritmo influye mucho mas la eficiencia temporal.

-Medir el tamaño de entrada:

Se debe elegir la misma variable para comparar algoritmos distintos aplicados a los mismos datos.
La variable que se suele utilizar es el tamaño de los datos de entrada n para medir el tiempo de ejecución.
A veces es difícil definir el tamaño de entrada, así que normalmente, tendremos tres o mas magnitudes.

- Medir el tiempo de ejecución:

En el analisis empirico se utilizan medidas del tiempo de ejecución en segundos, el problema de esto es que los tiempos de ejecución van ligados a la plataforma.

En el analisis matemático se cuenta el numero de veces que se ejecuta una operación.

- La medida del tiempo es independiente de la plataforma.
- Es necesario identificar la operación básica (la operación que contribuye mas al tiempo de ejecución.
- Se cuenta el número de veces que la operacion básica se ejecuta.

Identificar la **operación basica** es facil porque es la que mas tiempo consume.

La **operación básica** debe tener un tiempo de ejecución constante.

La eficiencia $C(n)$ es la que nos indica los recursos (tiempo o espacio), es decir, el numero de veces que se tiene que ejecutar la operación basica con una entrada del tamaño n.

Coste operacional (Cop): tiempo que necesita una operación basica para ser ejecutada.

Se puede estimar el tiempo que lleva ejecutar un algoritmo mediante la formula:

$$T(n) \approx Cop \cdot C(n)$$

Ejemplo: Si el coste computacional de un algoritmo viene dado por la formula $C(n) = \frac{1}{4}n^4$, ¿Cuanto tiempo tardara en ejecutarse el algoritmo si doblamos los elementos de entrada?

$$C(n) = \frac{1}{4}n^4 \rightarrow C(2n) = \frac{1}{4}2n^4$$

$$T(2n) \approx Cop \cdot C(2n) \rightarrow Cop = \frac{T(2n)}{C(2n)} = \frac{T(2n)}{1} : \frac{2n^4}{4} = \frac{4T(2n)}{2n^4}$$

$$T(n) \approx Cop \cdot C(n) \rightarrow Cop = \frac{T(n)}{C(n)} = \frac{T(n)}{1} : \frac{n^4}{4} = \frac{4T(n)}{n^4}$$

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

$$T(2n) = T(n) \cdot n^4$$

-Caso peor, mejor y medio:

En el caso peor llevamos al algoritmo hacia la peor condición que se pueda encontrar.

- Coste computacional peor $C_{worst}(n)$: Número máximo de veces que se ejecuta la operación con n elementos de entrada.

- Coste computacional mejor $C_{best}(n)$: Número mínimo de veces que se ejecuta la operación con n elementos de entrada.

Si no conocemos la distribución de los datos, estudiaremos el caso peor.

Si conocemos la distribución estadística de los datos, podremos estudiar el coste computacional medio $C_{aver}(n)$. Cogemos los datos del caso peor y los datos del caso mejor y haremos la media.

*Análisis de algoritmos:

-Tiempo de ejecución:

El tiempo de ejecución debemos expresarlo mediante una función matemática. Ejemplo: $T(n) = n^2 + 5n + 10$

Esto nos permite conocer la evolución de nuestro tiempo de ejecución dependiendo de nuestros datos de entrada.

Descomposición:

$$T(n) = n^2 + 5n + 10$$

Descomponemos el tiempo en componentes más pequeños:

Esto sirve para ver que componente tiene mas importancia dentro del algoritmo y saber que componente afecta mas al tiempo de ejecución.

Cada componente viene determinado por un trozo del código.

Ordenes que mas aparecen:

- Tratables: Son tiempos de ejecución tratables, es decir, podemos esperar físicamente a que el algoritmo termine.

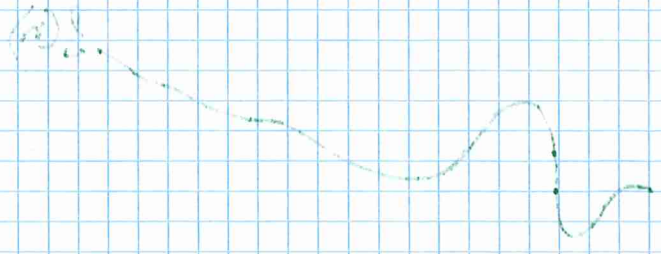
$$1 < \log(n) < n < n \cdot \log(n) < n^2$$

- Intratables: Son tiempos de ejecución intratables.

$$n^2 < n^3 < 2^n < n!$$

n^2 se encuentra en el límite y hay que estudiarlo a parte.

Siempre hay que tener en cuenta el tamaño de la entrada para poder decir si un problema es tratable o intratable.



-Notación asintótica:

La notación asintótica captura el orden de crecimiento de la operación básica según el tamaño de los datos de entrada van aumentando.

Suponemos las funciones $f(n)$ y $g(n)$ como funciones no negativas definidas sobre los números naturales.
$g(n)$ sería la función patrón asintótica con la que comparamos, sale del estudio del peor caso.

$f(n)$ es asintoticamente menor que $g(n)$ cuando:

$$f(n) < g(n)$$

$f(n)$ es asintoticamente mayor que $g(n)$ cuando:

$$f(n) > g(n)$$

$f(n)$ es asintoticamente igual que $g(n)$ cuando:

$$f(n) = g(n)$$

-Para comparar la orden de crecimiento utilizamos tres notaciones:

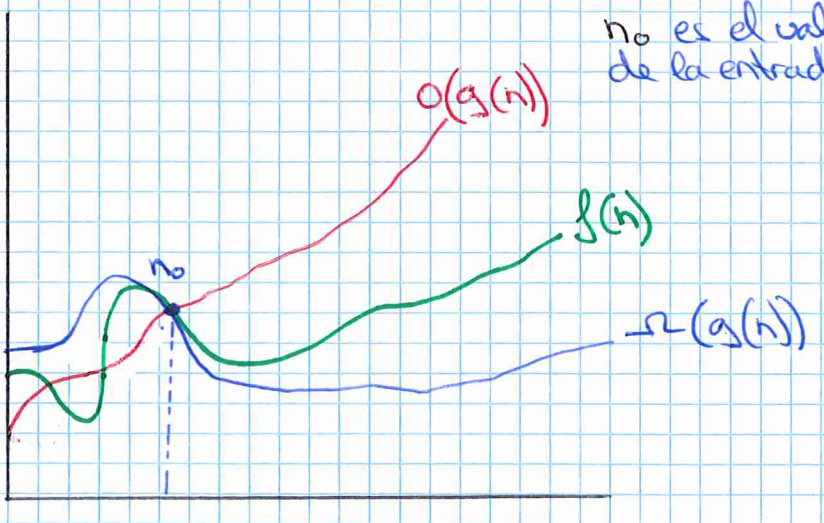
· Cota superior asintótica $O(g(n))$: Son las funciones con menor o igual orden de crecimiento que $g(n)$
De todas las cotas superiores nos interesara la mas baja.

· Cota inferior asintótica $\Omega(g(n))$: Son las funciones con mayor o igual orden de crecimiento que $g(n)$
De todas las cotas inferiores nos quedamos con la mas alta.

· Cota ajustada asintótica $\Theta(g(n))$: Son las funciones con el mismo orden de crecimiento que $g(n)$.

A partir de n_0, $O(g(n))$ y $\Omega(g(n))$, no volveran a crotarse con $f(n)$.

n_0 no tiene porque coincidir en el mismo punto entre $O(g(n))$ y $\Omega(g(n))$



- Orden de complejidad dominante I

Secuencia:
if{ } $C_1(n)$
if{ } $C_2(n)$

• Regla de la suma: Si un algoritmo consta de dos partes en secuencia, sus costes se suman.

$$C(n) = C_1(n) + C_2(n) + \dots + C_n(n)$$

El coste compuesto pertenece al máximo de los subcostes

• Ejemplo: $C(n) = C_1(n) + C_2(n)$

$$\begin{cases} C_1(n) = n \cdot \log(n) \\ C_2(n) = n^2 \end{cases}$$

Costes:	Orden
$C_1(n) = n \cdot \log(n)$	$O_1(n) = n$
$C_2(n) = n^2$	$O_2(n) = n^2$

} Siempre cogemos el valor superior del coste

- Orden de complejidad dominante II

• Regla del producto: Si un algoritmo $C_1(n)$ llama a otro algoritmo $C_2(n)$, sus costes se multiplican.

$$C(n) = C_1(n) \cdot C_2(n)$$
$$O(C(n)) = O(C_1(n)) \cdot O(C_2(n))$$

• Ejemplo:

$f_1(n)$ $\quad C_1(n) = 2n^2 + 5$
↳ $f_2(n)$ $\quad C_2(n) = n^3 + 1$

$C(n) = C_1(n) \cdot C_2(n) = (2n^2 + 5) \cdot (n^3 + 1)$
$O(C_1(n)) = n^2$
$O(C_2(n)) = n^3$
$O(C(n)) = O(C_1(n)) \cdot O(C_2(n)) = n^2 \cdot n^3 = n^5$

• Ejemplo:

```
int longitud(char* str){
  int blancos = 0;                          O(1)
  int i;
  for(i=0; str[i]!='\n'; i++){              O(n)
    if(str[i]==' ' || str[i]=='\t')         O(1)   } O(n)·O(1) = O(n)
      blancos++
  }
  for(i=0; str[i]!='\n'; i++){              O(n)
    return i-blancos                        Total = O(n) + O(n) = 2·O(n)
  }                                         El orden de crecimiento es proporcional a n.
```

- Análisis de algoritmos recursivos:

· Reducción por sustracción:

Ecuación de recurrencia

$$T(n)\begin{cases} c\cdot n^k & \text{Si } 0 \le n < b \\ a\cdot T(n-b)+c\cdot n^k & \text{Si } n \ge b \end{cases}$$

a = Número de llamadas recursivas.

b = Reducción del problema en cada llamada.

$c\cdot n^k$ = Todas las operaciones que hacen falta además de las de recursividad.

Resolución de la ecuación

$$T(n)\begin{cases} \theta(n^k) & \text{si } a<1 \\ \theta(n^{k+1}) & \text{si } a=1 \\ \theta(a^{n/b}) & \text{si } a>1 \end{cases}$$

· Reducción por división: Cuando en la llamada a la función estoy dividiendo el parámetro.

Ecuación de recurrencia

$$T(n)\begin{cases} c\cdot n^k & \text{si } 1 \le n < b \\ a\cdot T(n/b)+c\cdot n^k & \text{si } n \ge b \end{cases}$$

a = Número de llamadas recursivas

b = Reducción del problema en cada llamada.

$c\cdot n^k$ = Todas las operaciones que hacen falta además de las de recursividad.

$$T(n)\begin{cases} \theta(n^k) & n<b^k \\ \theta(n^k\cdot \log(n)) & n=b^k \\ \theta(n^{\log_b a}) & n>b^k \end{cases}$$

Ejemplo:

```
proc P(n){
    var i, j: enteros                    Coste total es A+B+C
    j ← 1
[ si n≤1 entonces terminar ← C
    si no{
        [ para i ← 1 hasta 7 hacer P(n DIV 2) -A → 7·T(n/2)
        [ para i ← 1 hasta 4·n³ hacer J ← J+1. -B → O(n³)
```

*Algoritmos de ordenación:

Los algoritmos de ordenación son algoritmos que reciben una lista de elementos y modifican sus valores para que aparezcan ordenados.

-Ordenación: es el proceso de colocación de los elementos de forma que cada uno y su sucesor satisfagan una relación.

Categorías disjuntas de algoritmos de ordenación:

-Ordenación interna: Cuando el algoritmo requiere una cantidad de memoria lo suficientemente pequeña en relación a lo que tengo, usara la memoria principal (RAM) Por ejemplo, si tengo 16Gb de RAM y mi algoritmo necesita 4Gb de RAM, no sera necesario usar la memoria secundaria (Disco duro).

-Ordenación externa: Cuando hay tantos registros que consume muchos recursos, entonces se recurre a la memoria secundaria (Disco duro).

Algoritmos de ordenación:

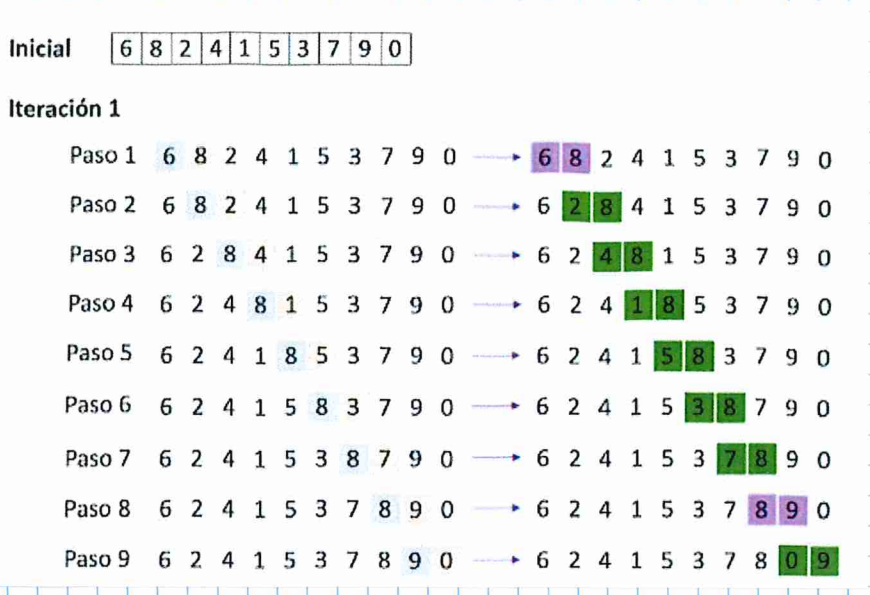
-ORDENACIÓN DE LA BURBUJA:

```
bubble_sort(int A[], int n) {  → Array de entrada.
    // Realiza n-1 iteraciones de burbujeo en la secuencia
    for (i = 1; i < n; i++) {
        // Burbujea el último elemento al final
        for (j = 1; j < n-i+1; j++) {
            if (A[j-1] > A[j])
                swap(A[j-1], A[j]);
        }   ← conmuta entre el elemento [j-1] y [j]
    }         Operación Base.
}
```

-La variable "i" es la que separa los elementos ordenados de los que no lo están.

El algoritmo divide el array en dos partes: Los elementos ordenados y los que no. Los ordenados se mantienen al principio. En cada iteración se coge el elemento no ordenado y el borde interior se usa para encontrar el punto de intersección. Se va desplazando el elemento mayor hacia la derecha.

- El for interno va desplazando números mayores al final y los mas pequeños al principio.
- El for externo hace correr el algoritmo interno hasta que hace (n-1) pasadas y el array queda ordenado.

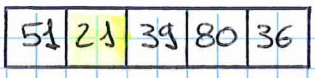


- ORDENACIÓN POR SELECCIÓN:

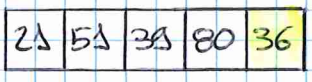
```
selection_sort(int A[], int n) {
for (i=0;i<n-1;i++) {
      // Almacena la posición e índice mínimos
      min_index = i;
      min_value = A[i];
            // Búsca el siguiente más pequeño
            for (j=i+1;j<n;j++) {
                  if (min_value > A[j]) {
                  min_index = j;
                  min_value = A[j];

                        }
                  }
                  if (min_index!=i) {
                        swap(A[i],A[min_index]);
                  }
            }
      }
```

- Busca el menor de los elementos y lo intercambia con el primero. Luego pasa al siguiente valor del array, busca el menor y lo intercambia por esa posición del array.

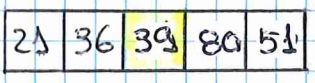
A[0] A[1] A[2] A[3] A[4]



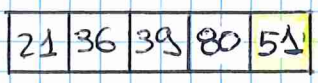
- Pasada 1: Selecciona 21 e intercambia 21 por A[0]



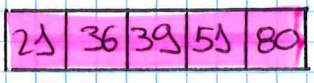
- Pasada 2: Selecciona 36 e intercambia 36 por A[1]



- Pasada 3: Selecciona 39 y como no encuentra uno menor no hace nada



- Pasada 4: Selecciona 51 e intercambia 51 por A[3]



- No hay pasada 5 porque nuestro bucle acaba cuando llegamos a "n-1".

-ORDENACION POR INSERCION:

```
insertion_sort(int A[], int n) {
    // Empieza con el primer elemento como único ordenado
1-  for (i=1;i<n;i++) {
        // value guarda el valor a ser insertado
        value = A[i];
        // Busca pos donde meter el elemento a insertar
        pos = i;
2-      while (pos>0 && value < A[pos-1]) {
            // Desplaza elemento a la derecha
            A[pos] = A[pos-1];
            pos--;
        }
        // Guarda value en el hueco
        A[pos] = value;
    }
}
```

La variable i es la que separa los elementos ordenados de los que no.

Realiza el intercambio desplazando el elemento a la derecha.

El algoritmo divide el array de elementos en dos partes:
Los elementos ordenados y los que no lo estan.
Los ordenados se mantienen al principio y su tamaño va creciendo en cada iteración.

- Es parecido a la ordenación por selección, la diferencia es, que al hacer esa inserción directa, los elementos ordenados quedan siempre al principio del array y en el resto del array los desordenados.

1. En cada iteración se coge el primer elemento no ordenado. y el bucle interior encuentra el punto de inserción.
2. Se realiza una busqueda de la posición de inserción desde atras y se van desplazando los elementos ordenados hacia la derecha, para dejar hueco al elemento a insertar.

A[0]	A[1]	A[2]	A[3]	A[4]
54	26	93	17	77

Se asume que 54 es una lista ordenada de 1 item.

26	54	93	17	77

- Compara 26 y 54 y como es menor, inserta el 26 en A[0]

26	54	93	17	77

- Compara 93 con 54 y como es mayor, no hace nada.

17	26	54	93	77

- Compara 17 con 93 y como es menor lo inserta, luego compara 17 con 54 y hace lo mismo hasta llegar a encontrarse con uno menor y parar.

17	26	54	77	93

- Compara 93 con 77 y al ser menor lo inserta, luego compara 77 con 54 y al ser mayor para.

- Los tres algoritmos anteriores son menos eficientes, porque siguen la estrategia de "comparación por pares" de elementos y los intercambia cuando no mantienen el orden.
- Los siguientes dos algoritmos que vamos a ver, utilizan la estrategia de "divide y conquista" de forma recursiva.

-ORDENACIÓN POR MEZCLA (MERGE_SORT):

Sigue los tres pasos de divide y conquista:
- Divide el array en dos subarrays del mismo tamaño.
- Ordena los elementos de cada subarray.
- Mezcla los elementos de los subarrays ordenados para obtener un array ordenado.

```
merge_sort(int A[], int left, int right)
  if (left<right) {
    center = (left+right) / 2;
    merge_sort(A, left, center);
    merge_sort(A, center+1, right);
    merge(A, left, center, right);
  }
```

La función merge() (intercalar) siempre ordena dos subarrays ordenados adyacentes. Para generar el array ordenado utiliza un array temporal.

-ORDENACIÓN RÁPIDA (QUICK_SORT):

Sigue los tres pasos de divide y conquista pero con alguna diferencia respecto al merge_sort.
- No decide el array por medio, sino que usa un splitter.
- No necesita de un array temporal para realizar las ordenaciones.
- Quick_sort realiza el trabajo principal en la etapa de división (llamada a split()), mientras que merge_sort realiza el trabajo principal durante la etapa de conquista (merge()).

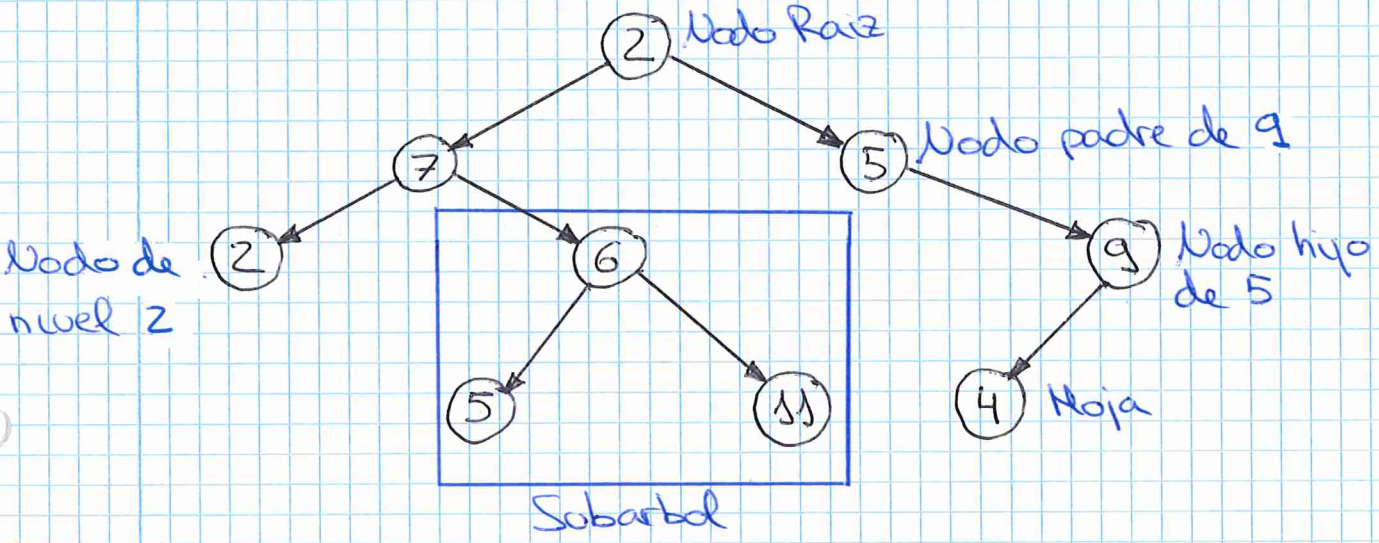
```
void quick_sort(int A[], int left, int right)
  if (left<right) {
    int i = split(A, left, right);
    quick_sort(A, left, i-1);
    quick_sort(A, i+1, right);
}
```

*Algoritmos con árboles:

-Los arboles son una estructura de datos no lineal, porque organizan los datos en nodos relacionados entre si a traves de ramas.

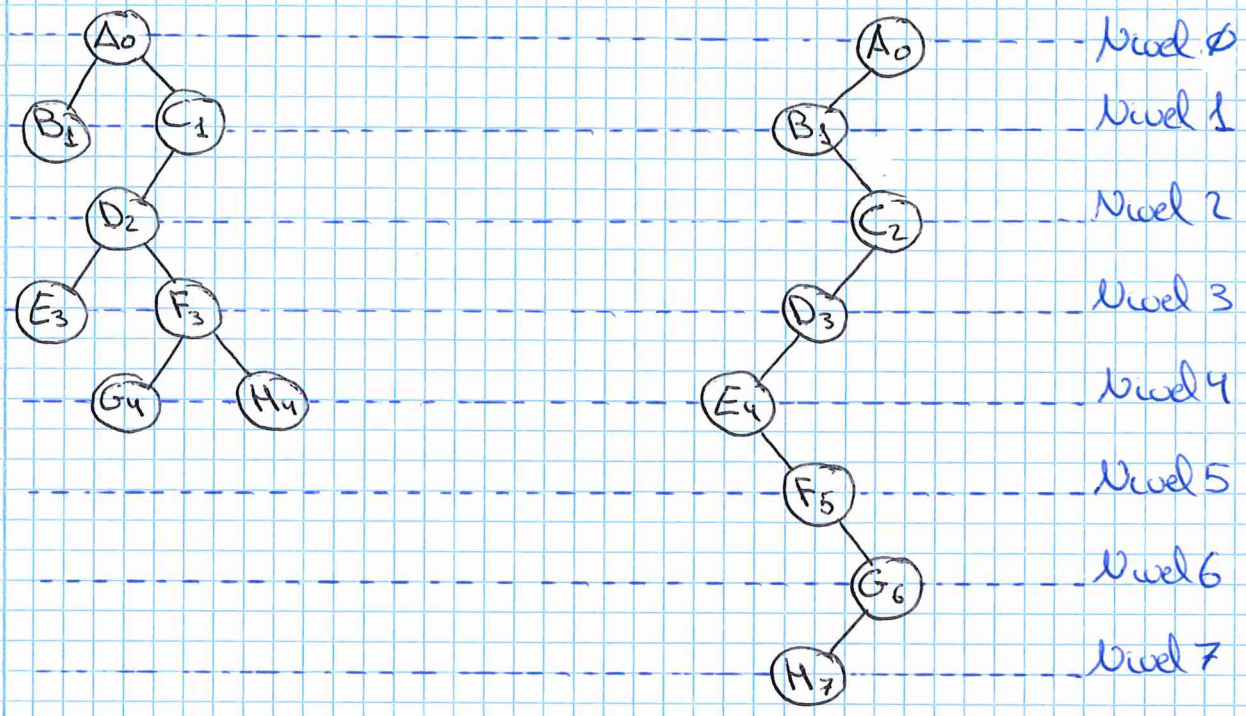
Elementos del arbol:

- Nodo: Almacenan un dato y sus relaciones con los nodos hijo.
- Nodo raiz: Es el nodo del que derivan todos los demas.
- Nodo padre: Todo nodo tiene un solo padre.
- Nodo hijo: Un nodo puede tener cero, uno o mas hijos.
- Subarbol: Cualquier nodo y sus hijos.
- Grado de un nodo: Numero de hijos que salen del nodo
- Nivel de un nodo: Numero de ancestros que tiene desde la raiz.
- Altura del arbol: Numero maximo de niveles de los nodos del arbol.



Este arbol tiene una altura de 4 niveles.

Árbol binario: Es aquel en el que cada nodo tiene como máximo grado 2, con lo cual, tiene como máximo dos hijos. a los que llamaremos hijo izquierdo e hijo derecho.



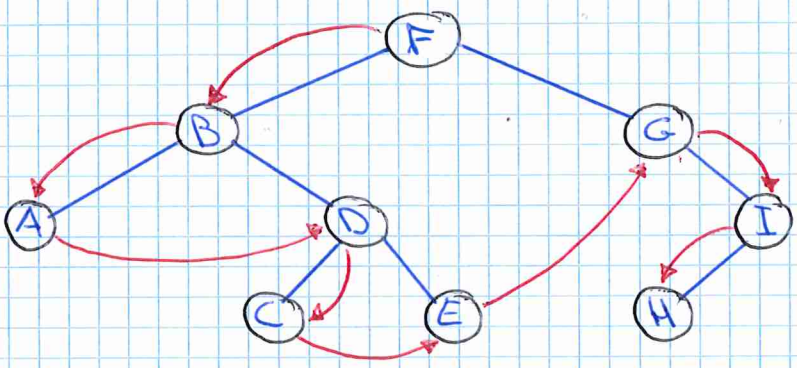
Un árbol de solo 1 hijo cada nodo, no es lo mismo que un array, porque los array no tienen relación gerarquica, pero los arboles si.

Un arbol binario es completo si todos sus nodos internos tienen exactamente dos hijos.
Un arbol binario balanceado es un arbol en el que, para cada subarbol, la altura de las hojas difiere en una unidad como mucho.

Recorridos en los arboles:

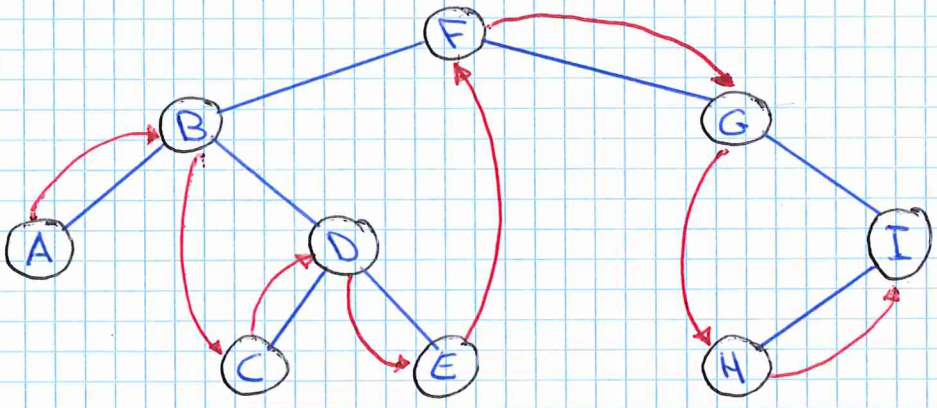
Consiste en recorrer todos sus nodos. Las estructuras lineales solo se pueden recorrer de delante hacia atras.

- Formas de recorrer un arbol:
  - Preorder:
    1. Visitamos la raiz.
    2. Recorremos en preorder el subarbol izquierdo.
    3. Recorremos en preorder el subarbol derecho.



Recorrido: F, B, A, D, C, E, G, I, H.

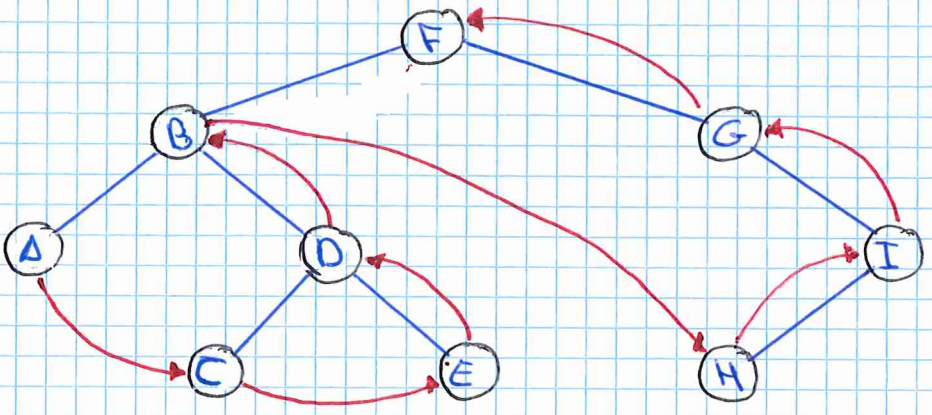
  - Inorder:
    1. Recorremos inorder el subarbol izquierdo.
    2. Visitamos la raiz.
    3. Recorremos inorder el subarbol derecho.



Recorrido: A, B, C, D, E, F, G, H, I

Es recorrerlo de izquierda a derecha, pasando antes del cambio de subarbol por la raiz.

- Postorder:
  1. Recorremos en postorder el subarbol izquierdo.
  2. Recorremos en postorder el subarbol derecho.
  3. Visitamos la raiz.



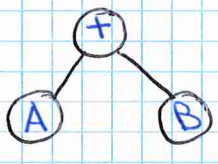
Recorrido: A, C, E, D, B, H, I, G, F

Es recorrerlo de izquierda a derecha, empezando de abajo hacia arriba y acabando en la raiz.

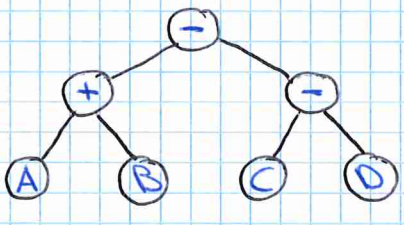
Un uso importante de los arboles es el de representar expresiones:

- Podemos usar un arbol binario para representar expresiones del tipo $(3+x)\cdot 3/(x+y)$.
- Tenemos el objetivo de evaluar esa expresión, el arbol que evalua se llama arbol con la expresión.
- En el arbol con expresión, los operadores se almacenan en los nodos interiores y los operandos se almacenan en los nodos hoja.
- Una vez construido el arbol, se puede usar para evaluar la expresión
- El operador de un nodo padre se ejecuta despues de evaluar a los hijos.

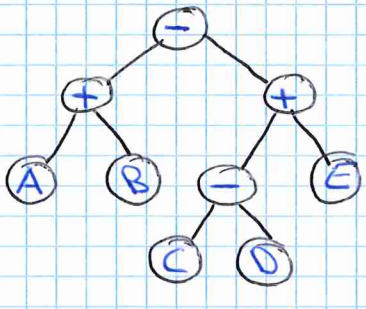
$A+B$



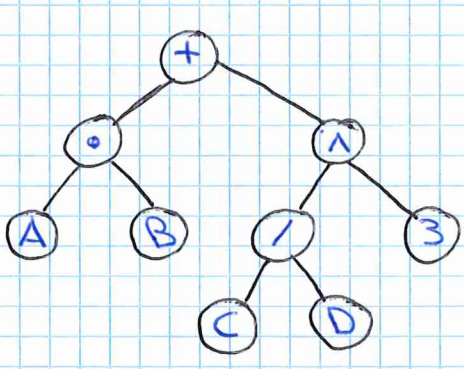
$(A+B)-(C-D)$



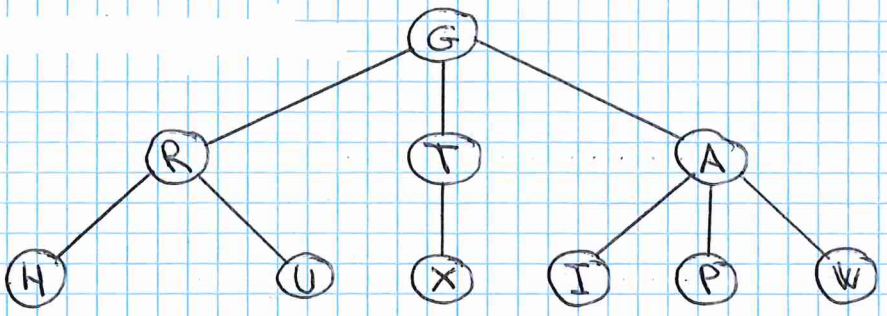
$(A+B)-((C-D)+E)$



Ejercicio: $(A\cdot B)+(C/D)^3$



Ejercicio:



Recorrido preorder: G, R, H, U, T, X, A, I, P, W

Recorrido inorder: H, R, U, G, T, X, I, A, P, W.

Recorrido postorder: H, U, R, X, T, I, P, W, A, G

* Algoritmos con arboles balanceados y ordenados:

- Tener un arbol balanceado y ordenado nos da una mejora en el coste computacional

- Arboles binarios ordenados:

Un arbol binario ordenado, tambien llamado arbol binario de busqueda, es un arbol que comple con la propiedad de ordenación.

• Propiedad de ordenación:

Sea x un nodo en un arbol binario balanceado, entonces:

Si y es un nodo del subarbol izquierdo $y \leq x$

Si y es un nodo del subarbol derecho $y \geq x$

Esto quiere decir que va de menor a mayor de izquierda a derecha.

El recorrido de un arbol binario ordenado es el recorrido inorder y es el que nos devuelve los elementos ordenados.

- Operaciones de los arboles binarios ordenados:

• Tree-Search(x,K): Nos permite encontrar el elemento con clave K en el arbol x.

• Tree-Minimun(x): Devuelve el elemento mas pequeño del arbol x.

• Tree-Maximun(x): Devuelve el elemento mas grande del arbol x.

• Tree-Succesor(x): Devuelve el siguiente elemento de un nodo

• Tree-Predecessor(x): Devuelve el anterior elemento de un nodo.

• Tree-Insert(x,z): Inserta el elemento z en el arbol x, manteniendo la propiedad de ordenación.

• Tree-Delete(x,z): Elimina el elemento z del arbol x, manteniendo la propiedad de ordenación.

- Busqueda en un árbol binario ordenado:

Comenzamos por el nodo raiz, si el valor que buscamos es menor al nodo actual, seguiremos por el nodo de la izquierda. En caso de ser mayor al nodo actual, seguiremos por el nodo de la derecha.

- Borrar nodos de un arbol binario ordenado:

La forma de borrar un nodo depende del tipo de nodo:

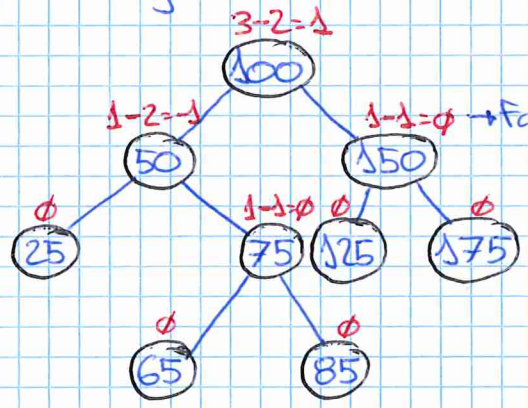
- Si es un nodo hoja basta con borrarlo.
- Si es un nodo con un solo hijo, lo borramos y reenganchamos el nodo hijo al padre del nodo borrado.
- Si es un nodo con dos hijos, ademas de todo lo anterior, tenemos que reordenar el arbol.

- Analisis computacional de los arboles binarios ordenados:

- Las operaciones en un arbol binario ordenado son proporcionales a la altura del arbol.
- Para un arbol binario completo, las operaciones ejecutan en $O(h)$ donde h es la altura del arbol.
- Si el arbol esta perfectamente balanceado $O(h) = O(\log_2 n)$
- Si el arbol fuera una estructura lineal. $O(h) = O(n)$

- Arboles binarios balanceados (AVL):

La eficiencia computacional maxima es cuando el arbol tiene una altura minima. $O(h)$.
El arbol binario balanceado (AVL Tree) es un arbol binario ordenado, donde para todos los nodos, la diferencia en altura entre las ramas izquierda y derecha sea 1, $\emptyset$ o -1.
Factor de balanceo: indica la diferencia entre la rama izquierda y la rama derecha.



Las operaciones de busqueda y recorrido son las mismas, pero las operaciones de insercion y borrado tienen que modificarse para cumplir con la propiedad de balanceo.

Dada la siguiente secuencia de números: 8, 9, 11, 15, 19, 20, 21, 7, 3, 2, 1, 5, 6, 4, 13, 14, 10, 12, 17, 16, 18, generar:

a) Su arbol binario de busqueda:

Sabemos que un arbol binario de busqueda es un arbol binario ordenado, asi que lo primero que haremos sera ordenar los datos.

{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21}

Sabemos que un arbol binario es aquel en el que cada nodo tiene como máximo grado 2.

Tambien sabemos, que para que nuestro coste computacional sea menor, nuestro arbol tiene que tener la menor altura posible.

Para conseguir esto, vamos a contabilizar los datos y encontrar la mediana, pero como variara el resultado dependiendo de si el número de datos es par o impar lo haremos asi:

-Si es impar: $\frac{\text{Numero total de datos}}{2}$, nos quedamos con la parte entera y le sumamos 1.

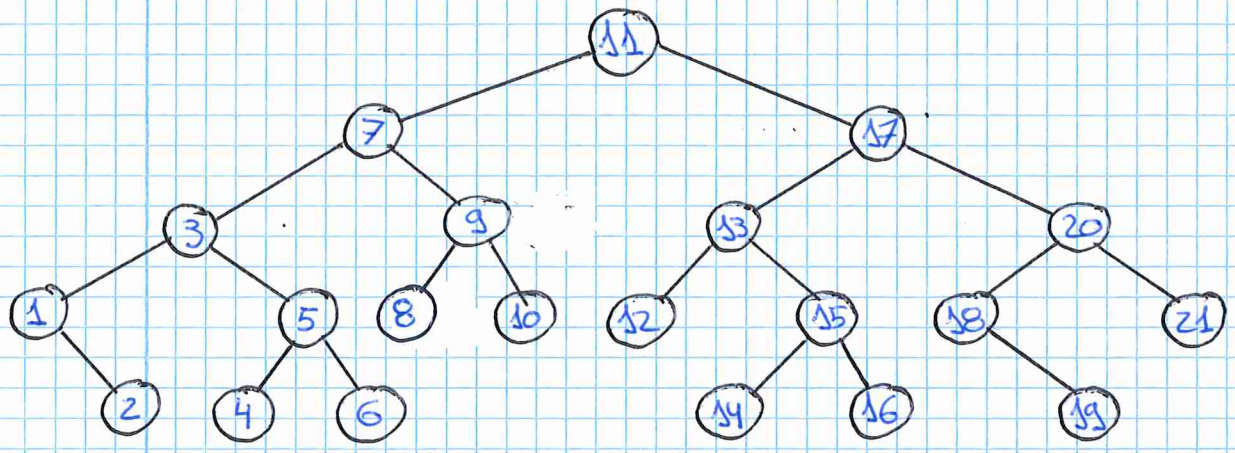
-Si es par: $\left(\frac{\text{Numero total de datos}}{2}\right) + 1$.

En nuestro caso tenemos 21 datos, que son impares:

$\frac{21}{2} = 10,5$ cogemos la parte entera y sumamos 1.

$10 + 1 = 11$ este sera nuestro nodo raiz, que en este caso coincide con el numero 11 porque es una serie numerica de 1 a 21 ordenada.

La ordenación a utilizar es inorder, asi que, como nuestro nodo raiz es el 11, el subarbol izquierdo contendra los datos del [1 al 10] y el subarbol derecho del [12 al 21].

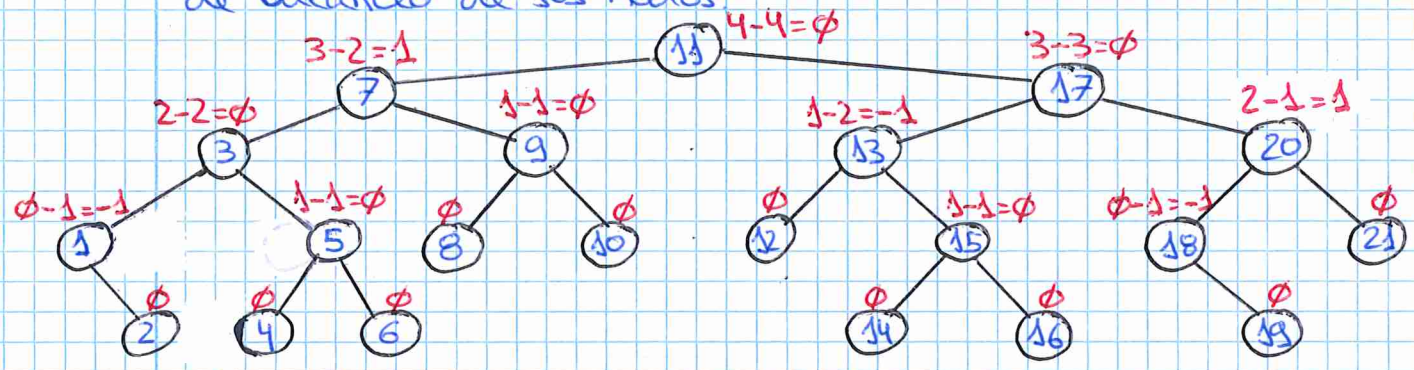


Este sería nuestro árbol binario de búsqueda.

b) Su árbol perfectamente balanceado:

Un árbol balanceado, es un árbol en el que la diferencia en altura entre el subarbol izquierdo y derecho es 1, ∅ o -1.

Vamos a analizar nuestro árbol analizando el factor de balanceo de sus nodos



-Todas las hojas tienen coste ∅.

-Para el resto de nodos: Altura subarbol izquierdo menos altura subarbol derecho.

Podemos decir que nuestro árbol está perfectamente balanceado.

c) Su árbol AVL:

Un árbol AVL es un árbol binario ordenado y balanceado.

En nuestro caso, nuestro árbol es un árbol AVL.

¿Qué conjeturas puedes sacar de estos ejercicios?

La principal conjetura que puedo sacar, es que antes de ponernos a trabajar con un árbol, debemos ordenar los datos.

Al ser árboles binarios, el nodo raíz casi siempre rondará en el valor central de los datos en el recorrido inorder.

¿En que tipo de árbol se genera el árbol de altura mínima?
En los árboles balanceados, ya que para que este balanceado, la diferencia entre la altura de los nodos del subarbol izquierdo y el subarbol derecho debe ser 1, ∅ o -1.

¿Cuáles serian las ventajas y desventajas de uno y otro esquema de representación de árbol?
Un arbol de bosqueda tiene la ventaja de que sus elementos estan ordenados, pero como no necesariamente tiene que estar balanceado, su coste computacional sera mayor que el de un arbol balanceado, el cual, tiene la ventaja de ser el tipo de arbol que genera menos altura y por lo tanto mayor eficiencia.
Pero los arboles AVL, son arboles balanceados y ordenados, con lo que cogen las ventajas de los dos arboles anteriores. Mejor eficiencia por la menor altura del arbol y mejor eficiencia en la bosqueda de datos al encontrarse ordenado. Al estar balanceado y ordenado, cuando realicemos una bosqueda, podemos ir descartando la mitad de los datos en cada paso de nodo.
Sabiendo poco de programación, supongo que una desventaja sera que cada vez que insertamos o eliminemos un dato, su coste crecera al tener que reordenarse y balancearse.

*Algoritmos con heaps:

-Heap (montículo): Es un array ordenado que se puede representar como un arbol.
Un heap nos permite insertar y extraer elementos de forma ordenada.
Cuando queremos realizar estas operaciones, la principal ventaja es que un heap puede realizarlas usando un simple array.
Puede visualizarse como un arbol binario de busqueda, donde cada nodo corresponde a un elemento del array.

-El array del heap tiene dos atributos:
- Capacidad: Indica la longitud del array usado para almacenar los elementos y corresponde a la capacidad máxima del heap.
- Longitud: Indica cuantos elementos tiene actualmente el heap.

-La raiz del arbol corresponde al elemento A[0] y, dado un elemento i, podemos calcular la posición de su padre e hijos con los operadores:
- Parent(i) return $\lfloor (i-1)/2 \rfloor$
- Left(i) return $2i+1$
- Right(i) return $2i+2$

-Los heaps cumplen dos propiedades:
- Propiedad de ordenación: Siempre trabajaremos con un array ordenado y su representación grafica sera en forma de arbol ordenado.

- Propiedad de completitud: El arbol de busqueda estara siempre completamente relleno a todos los niveles excepto, a veces el ultimo, que se rellena de izquierda a derecha.

Heaps inserción:
- Traslado el arbol al array e inserto el item en la ultima posición disponible dentro de la capacidad del array. Despues ordeno el array y vuelvo a representar el arbol.

Heaps de borrado:
- Traslado el arbol al array y borro el item, luego lo ordeno y vuelvo a representarlo en el arbol.

Cada vez que transformamos de arbol a array o viceversa, consumimos recursos de memoria asi que tendriamos que tener el doble de espacio para almacenar arbol y array.
Hay que vigilar el espacio de memoria que consumen.

-Coste computacional:
- Insertar un elemento en un heap tiene un coste $C_{worst} = \log_2 n$.
- Eliminar un elemento en un heap tiene un coste $C_{worst} = \log_2 n$.

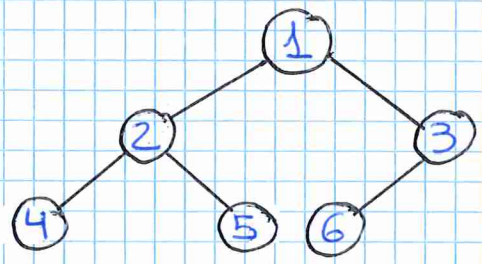
Esto se debe a que el arbol binario de busqueda es completo.

-Heapsort: El algoritmo aprovecha la simplicidad y eficiencia de los heaps para ordenar elementos.
Su coste computacional es de $O(n \log_2 n)$, aunque tiene un coste de memoria al pasar de arbol a array y viceversa.

-Colas de prioridad: son aquellas en las que cada elemento tiene una prioridad asociada, siendo el elemento con mayor prioridad el que sale primero de la cola.

Los heaps me permiten representar los datos en forma de array y en forma de arbol.

1	2	3	4	5	6



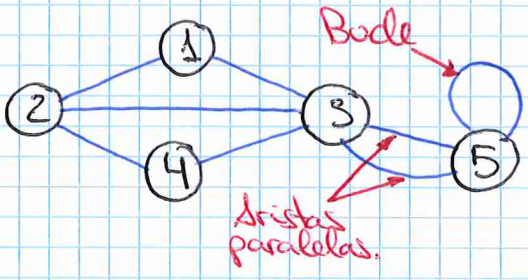
*Algoritmos con grafos:

-Los grafos son un tipo abstracto de datos, altamente flexible para representar y resolver problemas topológicos.
Los grafos estan formados por vertices "V" y aristas "E" que conectan los vertices. $G=(V,E)$
Permiten representar relaciones binarias entre elementos de un conjunto.

Un grafo G es un par ordenado $G=(V,E)$, donde:

-"V" es el conjunto de vertices o nodos.
-"E" es el conjunto de aristas o arcos que relacionan los nodos.
-Llamamos Grado del Grafo a su numero de vertices "V".
-Llamamos Grado del Vertice al numero de aristas incidentes en él.
-Un Bucle a una arista que relaciona el mismo nodo.
-Dos o más aristas son paralelas si relacionan el mismo par de vertices.

$G=(5,8)$

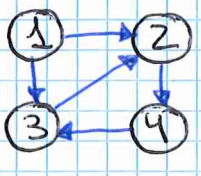


Grado del grafo: 5
Grado de los vertices:
$deg(V_1)=2$
$deg(V_2)=3$
$deg(V_3)=5$
$deg(V_4)=2$
$deg(V_5)=3$

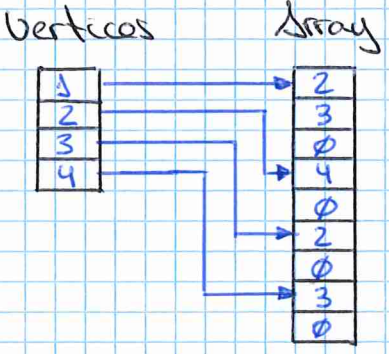
- Formas de representar un grafo:

• Lista de adyacencia: Indica los vertices que son adyacentes a cada vertice.

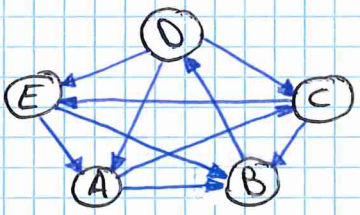
Si el grafo es dirigido, indican los vertices hasta los que llega el arco.



1 → 2, 3
2 → 4
3 → 2
4 → 3

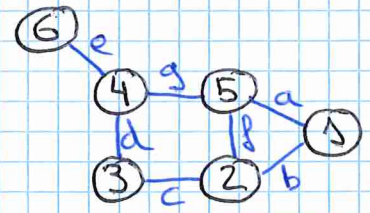


• Matriz de adyacencia: Matriz donde se indica, para cada vertice, si hay un arco hacia el vertice de la otra entrada. Debe marcarse el origen y el destino.



Origen \ Destino	A	B	C	D	E
A	0	1	1	0	0
B	0	0	0	1	0
C	0	1	0	0	1
D	1	0	1	0	1
E	1	1	0	0	0

• Matriz de incidencia: Matriz donde una entrada representa los vertices y otra los arcos. La matriz muestra cuando un vertice incide con un arco.



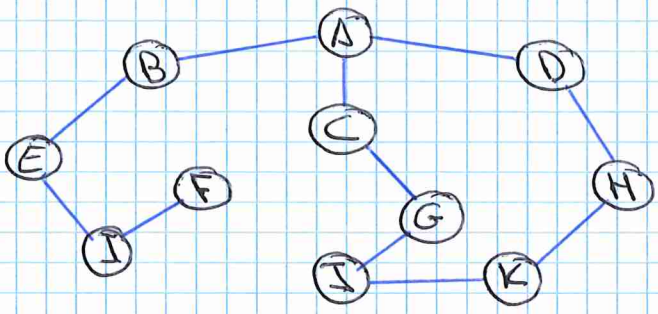
	a	b	c	d	e	f	g
1	1	1	0	0	0	0	0
2	0	1	1	0	0	1	0
3	0	0	1	1	0	0	0
4	0	0	0	1	1	0	1
5	1	0	0	0	0	1	1
6	0	0	0	0	1	0	0

Consumimos menos memoria cuando usamos listas de adyacencia con grafos dispersos. Las matrices de adyacencia son más eficientes para grafos densos.

- Recorrido de grafos:

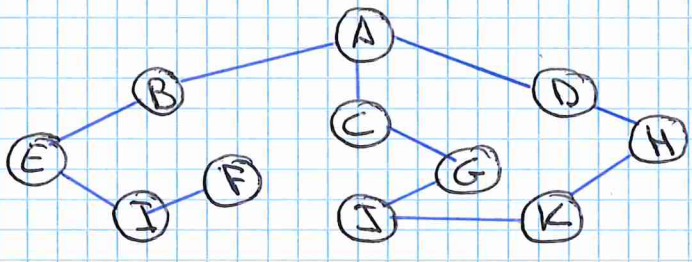
· Recorrido en anchura:

Dado un grafo $G=(V,E)$ y un origen (source "s") $s \in V$, el recorrido en anchura explora todos los arcos "E" para encontrar todos aquellos alcanzables desde "s".



Recorrido en anchura: A, B, C, D, E, G, H, I, J, K, F

· Recorrido en profundidad: El algoritmo de recorrido en profundidad avanza por el primer nodo que encuentra, siempre que sea posible, y no retrocede hasta que no queden más vertices por los que seguir.



Recorrido en profundidad: A, B, E, I, F, C, G, J, K, H, D

Tanto en el recorrido en anchura como el recorrido en profundidad, el coste de recorrer el grafo es $O(V+E)$

Ordenación topológica: Cuando sus elementos cumplen las relaciones:

- Reflexiva
- Transitiva
- Antisimetrica.

Una forma de representarlo es con un grafo dirigido sin ciclos.

*Algoritmos greedy:

Los algoritmos que tratan de resolver problemas de optimización suelen recorrer la misma secuencia de pasos, pero la estrategia de recorrer todas las opciones es demasiado costosa.

Los algoritmos greedy, voraz o ávido siempre realizan la opción que parece mejor en cada momento.
Realiza la elección que le conduzca a la solución más óptima.

-Estrategia greedy:

- Se aplica a problemas de optimización que tienen muchas soluciones, pero se busca una solución que satisfaga una determinada restricción definida.
- A cada uno de los subconjuntos que cumplen las restricciones se les llama soluciones prometedoras.
- Hay que analizar cada paso en la solución que vallamos dando para ver si maximiza o minimiza nuestra función objetivo (solución óptima).

La función objetivo es lo que yo quiero maximizar o minimizar del problema.

-Elementos de la estrategia greedy:

- Método: Encontrar los elementos para que se cumpla nuestro objetivo.
- Conjunto de candidatos: todos aquellos elementos que forman parte de la solución.
- Función de selección: Cada vez que el algoritmo tiene que decidir por donde ir, nos dirá por donde ir. La función de selección activa mientras estamos resolviendo el algoritmo.
- Tener una función que nos diga si el subconjunto de candidatos es prometedor.
- Función objetivo: es la función que queremos maximizar o minimizar. Es la primera que tenemos que definir.
- Función de coste: comprueba si es optima nuestra solución.

A partir de los elementos anteriores:

- Los algoritmos ávidos se construyen de manera que vaya avanzando por etapas, tomando en cada una de ellas la decisión que parece mejor sin considerar las consecuencias futuras.
- Escogerá de entre todos los candidatos el que produce un óptimo local para esa etapa, suponiendo que será, a su vez, el óptimo global del problema.

-Resumen del funcionamiento:

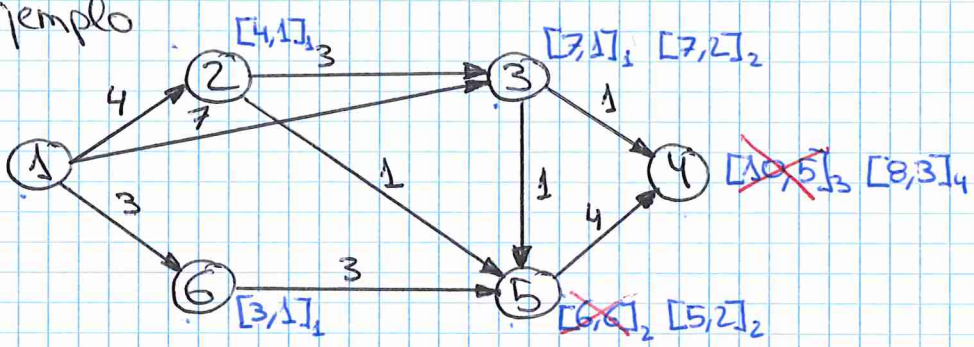
1. Se parte de un conjunto vacío $S = \emptyset$.
2. Se elige al mejor candidato de acuerdo con la función de selección.
3. Comprobamos si se puede llegar a una solución con el candidato seleccionado (función de factibilidad). Si no es así, lo eliminamos de la lista de candidatos para siempre.
4. Si no hemos llegado a una solución, seleccionamos otro candidato y repetimos el proceso hasta llegar a una solución o quedarnos sin candidatos.

*Búsqueda de caminos mínimos:

-Dado un grafo $G(V,E)$ con función de pesos $w: E \to R$, el problema del camino mínimo consiste en encontrar el camino mas corto desde un nodo origen a cualquier otro nodo. (Algoritmo de Dijkstra).

-Se llama peso del camino a la suma de los pesos de los arcos que lo costruyen.

Ejemplo



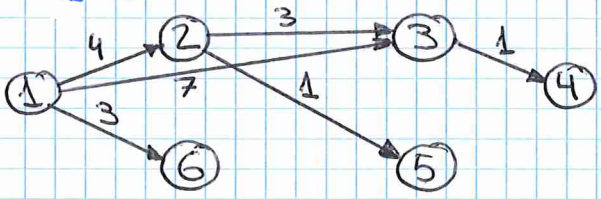
- Empezamos desde un nodo origen, en este caso el 1, y vemos sus posibles caminos.
- Recorremos los caminos y apuntamos los valores con la siguiente notación:

$[D, O]_{(i)}$ ← Iteración.

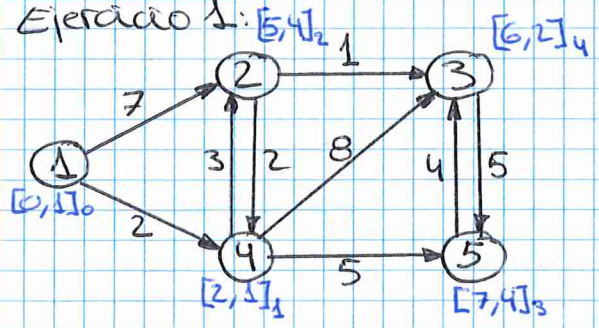
Peso acumulado ↑ ↑ Origen

- De todas las iteraciones seguimos con la del peso mínimo repitiendo el paso anterior y así hasta el final.
- Una vez recorrido, en los nodos donde tengamos varias iteraciones, nos quedamos con la de menor peso. En caso de coincidir los pesos nos quedamos con las dos.

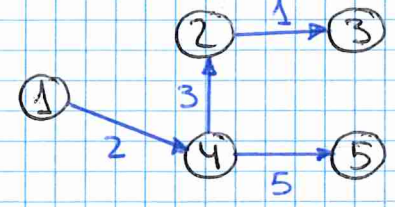
El grafo resultante sería:



Ejercicio 1:

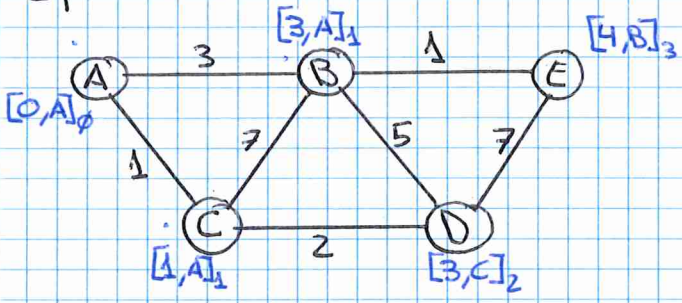


Resultante:

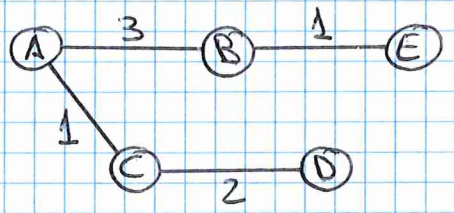


Lista [1,4,2,3,5]

Ejercicio 2:



Grafo resultante:



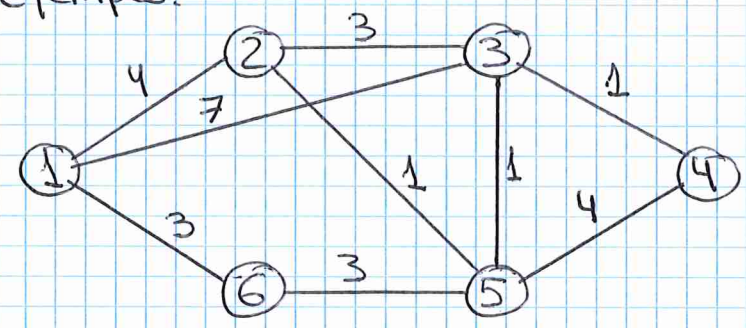
Lista [A,C,B,D,E]

*Árbol de expansión mínima: (Algoritmo de Prim).

Un árbol generador mínimo es un árbol que utiliza todos los nodos del grafo, de tal manera que el costo total de sumar las aristas del grafo es mínimo. Sirve por ejemplo para crear rutas de un origen a un destino con el coste mínimo pasando por todos los nodos.

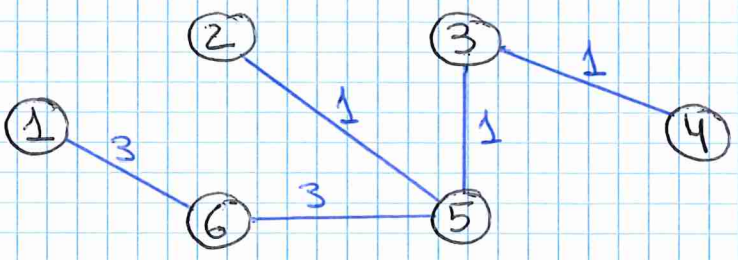
- El algoritmo de Prim mantiene la conexión de todos los nodos.
- En cada paso elige la arista mas barata entre un nodo agregado y otro no agregado todavía.
- En cada paso hay un nodo nuevo sumandose al conjunto de nodos.
- Despues de n-1 pasos, habremos agregado todos los nodos de la forma mas barata posible, obteniendo un árbol generador mínimo.

Ejemplo:

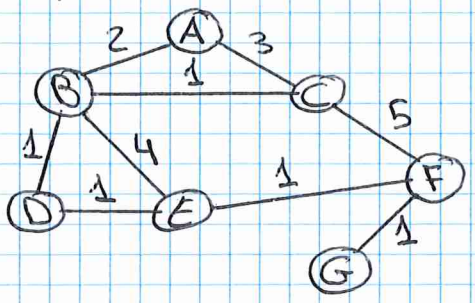


- Elegimos un nodo de inicio, en este caso 1.
- Elegimos la arista mas barata para conectar el nodo uno con otro, en este caso la 1-6.
- Ahora buscamos la arista mas barata para conectar los nodos que ya he agregado con el siguiente nodo. 6-5.
- Seguimos con este mecanismo hasta terminar agregando todos los nodos.

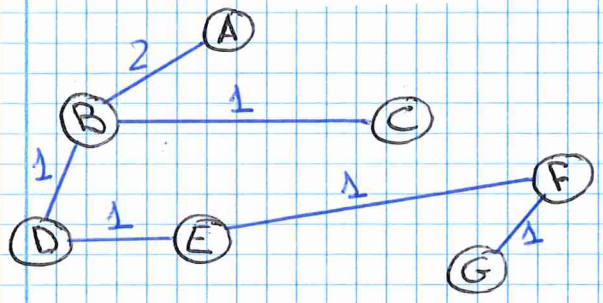
-Grafo resultante:



Ejercicio: Determinar el arbol de expansión mínima.



-Como para usar el algoritmo de Prim a diferencia del algoritmo de Dijkstra no necesitamos nodo especifico de inicio, utilizaremos el que queramos.

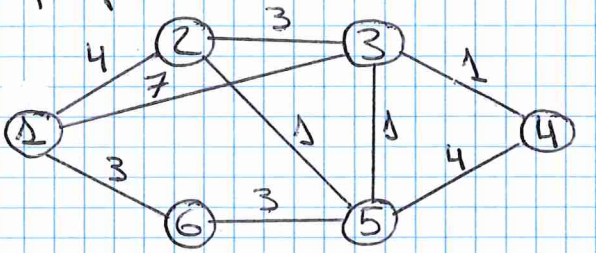


Este es nuestro arbol de expansion mínima resultante.

-Algoritmo de Kruskal:

En este algoritmo se ordenan primero las aristas por orden creciente de pesos y en cada etapa se decide que hacer. Si el arco no forma un ciclo con los ya seleccionados se incluye y si no se descarta.

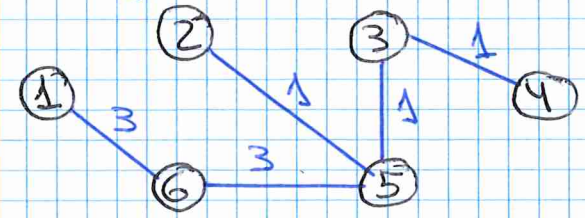
Ejemplo:



·Ordenamos las aristas de menor a mayor y elegimos las mas pequeñas. (1,3,4,7) Nos quedamos con 1.

· Seleccionamos todas ellas siempre que no formen ciclos.
· Pasamos a la siguiente menor, 3, y repetimos el paso anterior.
· Asi hasta dejar nuestro grafo conexo y sin ciclos.

-Grafo resultante:



*Backtracking:

En los problemas con grafos puede haber mas de una solución al problema, pero no todas las soluciones son igual de eficientes.

En este tipo de problemas, el tamaño de la entrada es crucial.

Para estos problemas podemos usar la técnica del Backtracking.

¿Que es el backtracking?

- Tiene un coste exponencial, pero si se diseña correctamente, puede tener un coste muy eficiente.

  En los algoritmos voraces se va construyendo la solución por etapas, siempre avanzando sobre la solución parcial construida. Pero esto no siempre es posible.
- Hay que estudiar el grafo al que me voy a enfrentar para poder diseñar estrategias que sean mas eficientes.
- El Backtracking va a eliminar las soluciones que sabemos que no nos van a llevar a la solución óptima.
- Espacio de soluciones.
  - Los algoritmos de backtracking determinan las soluciones del problema buscando en el espacio de soluciones.
  - Esta busqueda puede ser representada en un arbol de soluciones.
  - Conociendo las ramas del grafo que no me van a llevar a nada, podemos descartarlas y mejorar nuestro algoritmo.

Examen 1.

Ejercicio 1.

¿Cual es el coste computacional (O) de los siguientes algoritmos?

a)

```
int f(int n)
{
    int sum=0;
    for (int i=1; i<=n; ++i)        → O(n)
        sum += i;                   → operación básica
    return sum;
}
```

O(n) es el coste computacional ∈ O(n)

b)

```
int g(int n)
{
    int sum=0;
    for(int i=1; i<n; ++i)          → O(n)
        sum+= i + f(n);             → Operación básica  f(n) = O(n)
    return sum;
}
```

$O(n) \cdot O(n) = O(n^2)$

c)

```
int h(int n)
{
    return f(n) + g(n);   f(n) + g(n).
}
```

$f(n) = O(n) \qquad g(n) = O(n^2)$

$f(n) + g(n) = O(n) + O(n^2)$

Ejercicio 2:

Encontrar el coste de los siguientes algoritmos siendo $h(n,r,k) \in O(n)$

a) procedimiento uno (n,K: entero)

```
VAR i,r: entero;
    Comienzo
      Si n<2 Entonces devolver 1;
    SI NO
    Comienzo
    r← uno(n DIV 2, K-1);
    r←r+uno(n DIV 2, K+1);
    r←r·uno(n DIV 2, K+2);
      Devolver r;
    FIN
FIN
```

a = Número de llamadas recursivas = 3

b = reducción del problema en cada llamada = 2

Según las fórmulas el coste de cada llamada sería:

$O(n^{\log_2 3})$

Como son 3 secuencias independientes, sus costes se suman, por lo tanto:

$$O(n^{\log_2 3}) + O(n^{\log_2 3}) + O(n^{\log_2 3}) = (n^{\log_2 3})$$

El coste será $O(n)$ si $n<2$ y $(n^{\log_2 3})$ si $n \geq 2$.

b) procedimiento dos (n, K: enteros)
VAR i, r: entero;
Comienzo
r ← dos (n DIV 2, K-1);
r ← r + dos (n DIV 2, K+1);
} Parte 1
Para i ← 1 hasta n/2 Hacer
Comienzo
r ← h(n, r, i);
r ← r + h(n, r-1, i)
} Parte 2
FW
r ← r + dos (n DIV 2, K+2);
} Parte 1
Devolver r;
FW
FW

Este algoritmo lo estudiaremos por partes

Parte 1:

a = número de llamadas recursivas = 3

b = reducción del problema en cada llamada = 2

$O(n^{\log_2 3})$

La parte 1 realiza 3 llamadas recursivas, pero en la segunda llamada tiene anidada otra llamada diferente con dos llamadas anidadas dentro, por lo tanto:

Parte 2:

$a = 1 \quad b = 2 \rightarrow O(n)$

En el algoritmo anidado (reducción por sustracción)

· la primera llamada no reduce, pero la segunda llamada reduce 1. $a = 2$ y $b = 1$

So coste sera $O(n)$

Ahora realizamos las cuentas empezando por la parte 2, Como estan anidados: $O(n) \cdot O(a^n) = O(n) \cdot O(2^n)$

Como la parte 1 no estan anidados, realizamos la suma:

$O(n^{\log_2 3}) + O(n^{\log_2 3}) + O(n^2) + O(n^{\log_2 3}) = O(n^2)$

Ejercicio 3:
¿En que orden esta el tiempo de ejecución del siguiente algoritmo? Justifica tu respuesta.

```
procedimiento h(n, i, j)
  si n>0 entonces
    h(n-1, i, 6-i-j);
    escribir i"→"j;
    h(n-1, 6-i-j, j);
  fsi.
```

$a=2 \quad b=1$ coma $a>1 \rightarrow a^{n/b}$

$O(2^n)$ al tener este coste su tiempo sera exponencial dependiendo de n.

Ejercicio 4:
Explica de forma detallada y razonada las diferencias entre los algoritmos de ordenación basados en selección, inserción y borbuja.
-En el algoritmo de la burbuja vamos recorriendo el array desde el principio al final y vamos comparando los elementos, llevando el mayor hacia la derecha intercambiando los elementos hasta que quedase ordenado.
-En el algoritmo por inserción comienza a recorrer el array desde el principio y va colocando los elementos en la posición que nos parecia adecuada en ese momento, al hacer la inserción directa, los elementos ordenados quedan siempre al principio del array y en el resto del array los desordenados.
-En el algoritmo por selección se selecciona el elemento que queremos extraer y lo colocamos en la posición correcta. Busca el menor del array y lo coloca en la posicion 1, luego busca otra vez el menor y lo coloca en la posición 2, asi hasta que quede ordenado.
-La diferencia entre los tres algoritmos es la forma de permutar el orden del array.

¿Cual es la principal diferencia entre el algoritmo Mergesort y QuickSort?

La principal diferencia es la forma en la que se divide el array en el corte inicial.

Mergesort lo divide por la mitad y quicksort lo divide con un split.

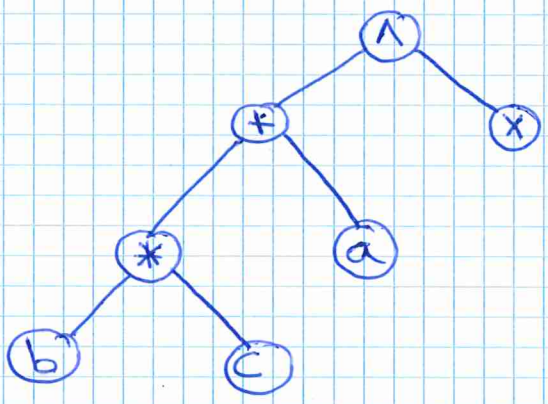
¿De que depende la decisión de aplicar un algoritmo u otro?

- Del numero de elementos a ordenar.
- Del hardware del que disponemos.
- De los datos que vamos a ordenar, conocer el tipo de elementos y si hay algunos previamente ordenados.

Ejercicio 5:

Dada la siguiente expresión matemática $(a+b \cdot c)^x$, se pide:

- Construir el arbol binario que representa la expresión matemática.



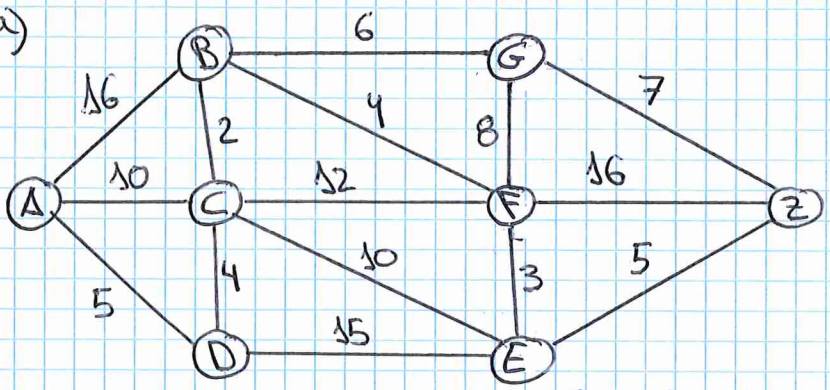
- Recorrer el árbol obtenido en preorder, inorder y postorder:

  - Preorder: ^, +, *, b, c, a, x
  - Inorder: b, *, c, +, a, ^, x
  - Postorder: b, c, *, a, +, x, ^

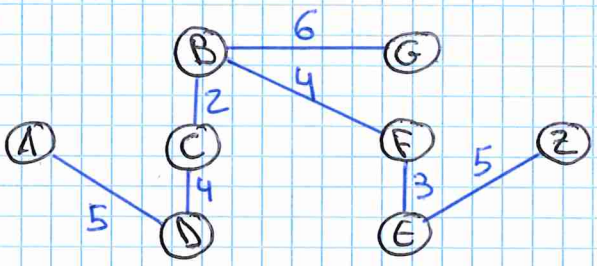
Ejercicio 7:

Generar el arbol de expansión mínima de los siguientes grafos aplicando el algoritmo de Prim.

a)

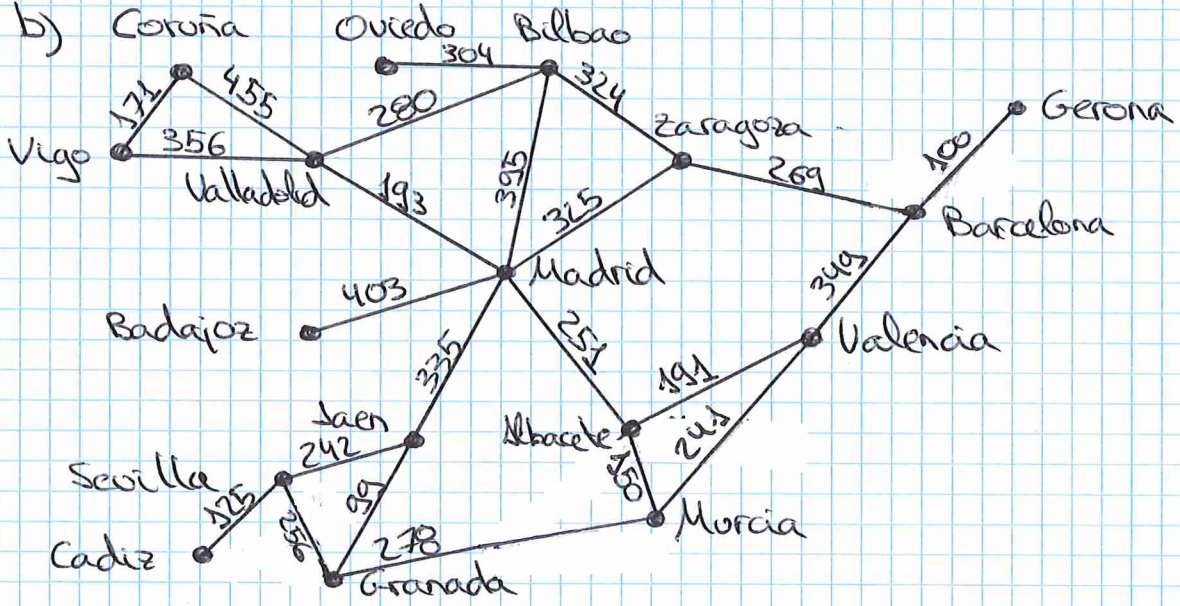


Vamos a comenzar por el nodo A:

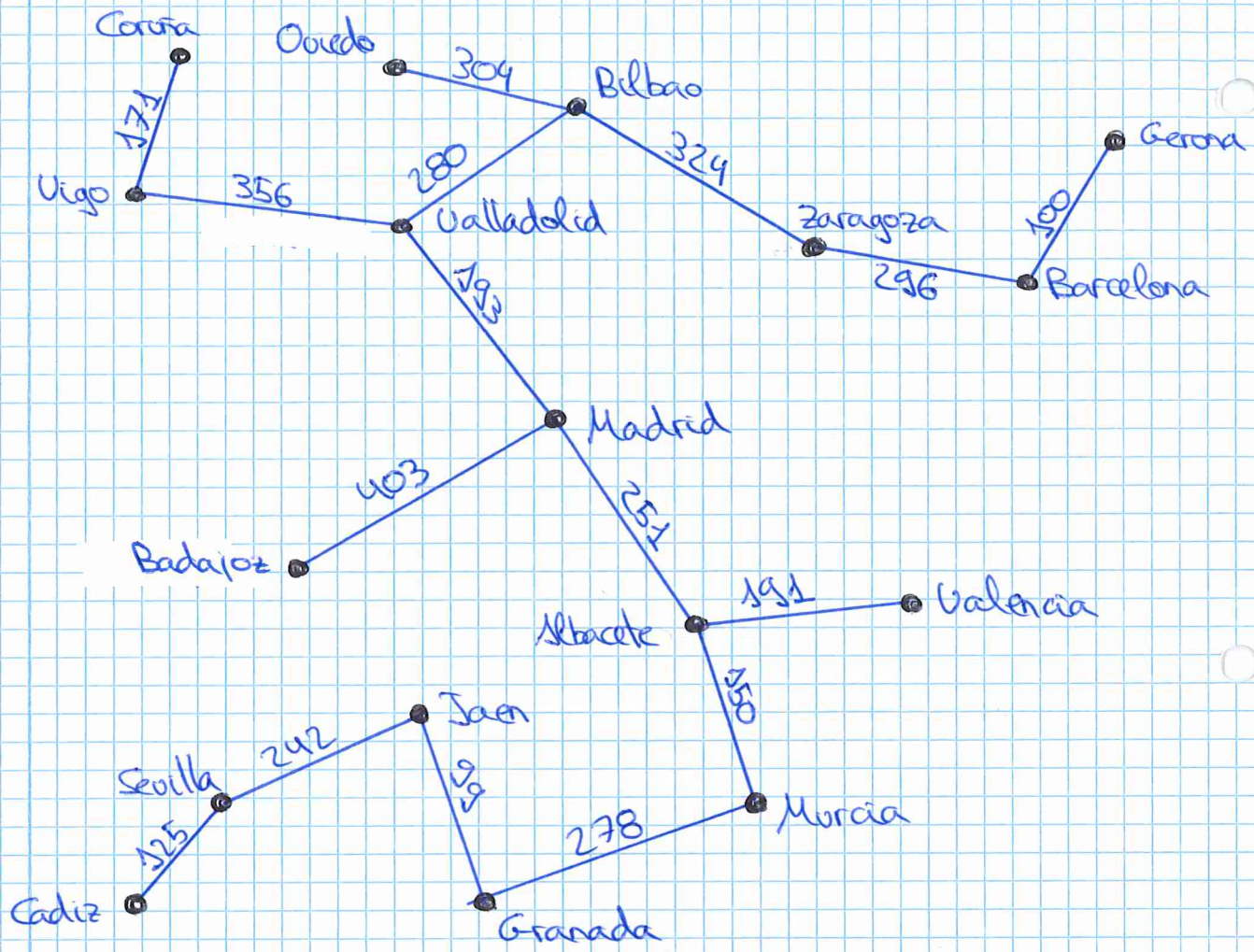


Este es nuestro grafo resultante.
A, D, C, B, F, E, Z, G

b)



Comenzaremos por Vigo:
Vigo, Coruña, Valladolid, Madrid, Bilbao, Albacete, Murcia, Valencia, Granada, Jaen, Sevilla, Cadiz, Oviedo, Zaragoza, Barcelona, Bilbao.


Coruña
Oviedo
304
Bilbao
171
280
324
Gerona
Vigo
356
Valladolid
Zaragoza
100
296
Barcelona
193
Madrid
403
251
Badajoz
191
Valencia
Albacete
150
Jaen
242
Sevilla
129
325
278
Murcia
Cadiz
Granada