# EDA

## “Temari Final”

- Bloc 1: **Análisis de algoritmos**
- Bloc 2: **Divide y vencerás**
- Bloc 3: **Estructuras de Datos**
- Bloc 4: **Grafos**
- Bloc 5: **Búsqueda Exhaustiva**
- Bloc 6: **Clases P y NP**

*A.S.E.S.*

**Miriam**
**Telf: 93.204.62.56**

# Análisis de algoritmos

## Idea

Si tenemos varios algoritmos correctos, analizamos una serie de aspectos para saber que un algoritmo es mejor que otro. Los aspectos a analizar de un algoritmo son:

- Tiempo que tarda (aspecto más importante)
- Espacio que ocupa

Estos dos aspectos pueden estar confrontados. Ejemplo: un algoritmo que ocupe un espacio de n elementos podría ser mejorado en tiempo si ocupase un espacio de $n^2$ elementos. En EDA se tiene en cuenta básicamente el tiempo que tarda. Existen dos formas de calcular el tiempo de ejecución de un algoritmo:

INCORRECTO
- Ejecutar el algoritmo en un computador y calcular el tiempo que tarda con un cronómetro.
Problemas: depende de la cpu, lenguaje, SO, etc.

CORRECTO
- Comparar con métodos analíticos, diremos que operaciones básicas como comparaciones o asignaciones tardan una unidad de tiempo.

## Análisis por casos

El tiempo de ejecución de un algoritmo depende de dos factores:

- El tamaño de la entrada. Ejemplo: un algoritmo de búsqueda de un elemento en una tabla tardará más si la tabla tiene 1000 elementos en vez de 50.
- Organización de la entrada: se trata de cómo está organizada la entrada en un caso particular, ya que si por ejemplo nos aparece en el código "si (...) entonces..." el coste aumenta si la condición del si se cumple. Entonces es cuando el análisis de un algoritmo se realiza mediante un análisis por casos:

# caso peor: tiempo máximo de ejecución. LA +UTILIZADA EN EDA
# caso medio: tiempo medio de ejecución. +EXACTO, MAYOR COMPLEJIDAD
# caso mejor: tiempo mínimo de ejecución. NO ES MUY USADO, PARA CONTRASTAR EFICIENCIA INTERESA SABER EL PEOR COMPORTAMIENTO DEL ALGORITMO

***Ejemplo***: análisis de un algoritmo que busca un elemento x en un vector de enteros

```
bool buscarEntero(const vector<int>& v, int x) {
   int i = 0;
   while (i < v.size() and v[i] != x)  i++;
   return (i < v.size());
}
```

**caso peor:** el vector v no contiene el elemento x, por lo tanto, se recorrerá todas las posiciones del vector.
**caso mejor:** la primera posición del vector contiene el elemento x, por lo tanto, el bucle sólo se ejecuta una vez.

## Notación asintótica

Objetivo: asociar una función (o varias) a un algoritmo para que describa su coste. Definiciones:

- **cota superior O(f)**: indica todas las funciones g que su tasa de crecimiento es igual o inferior a f.

  Definición formal:

  $$O(f) = \{g : \mathbb{N} \to \mathbb{R} \mid [\exists c > 0, \exists n_0 \in \mathbb{N} \mid \forall n \geq n_0 : |g(n)| \leq c \cdot |f(n)|]\}$$

  Dadas dos funciones $f, g : \mathbb{N} \to \mathbb{R}$, podemos decir que:

  $$g \in O(f) \text{ sii } 0 \leq \lim_{n\to\infty} \left(\left|\frac{g(n)}{f(n)}\right|\right) < \infty$$

- **cota inferior Ω(f):** indica todas las funciones g que su tasa de crecimiento es igual o superior a f.

  Definición formal:

  $$\Omega(f) = \{g : \mathbb{N} \to \mathbb{R} \mid [\exists c > 0, \exists n_0 \in \mathbb{N} \mid \forall n \geq n_0 : |g(n)| \geq c \cdot |f(n)|]\}$$

  Dadas dos funciones $f, g : \mathbb{N} \to \mathbb{R}$, podemos decir que:

  $$g \in \Omega(f) \text{ sii } 0 < \lim_{n\to\infty} \left(\left|\frac{g(n)}{f(n)}\right|\right) \leq \infty$$

- **cota igual θ(f):** indica todas las funciones g que su tasa de crecimiento es igual a f. => O(f) $\wedge$ Ω(f).

  Definición formal:

  $$\Theta(f) = \{g : \mathbb{N} \to \mathbb{R} \mid [g \in O(f) \wedge f \in O(g)]\}$$

  Dadas dos funciones $f, g : \mathbb{N} \to \mathbb{R}$, podemos decir que:

  $$g \in \Theta(f) \text{ sii } 0 < \lim_{n\to\infty} \left(\left|\frac{g(n)}{f(n)}\right|\right) < \infty$$

**Propiedades**:

- Los operadores $\Theta()\ y\ O()\ \ y\ \Omega()$ son reflexivos. $f \in \Theta(f)\ \ y\ \ f \in O(f)\ y\ \ f\ \Omega(f)$.
- EL operador $\Theta()$ es simétrico: $f \in \Theta(g)\ sii\ \ g\ \in \Theta(f)$
- Si $f \in \Theta(g)\ \ llavors\ \ a \cdot f \in \Theta(b \cdot g)$
- Si $f \in O(g)\ \ llavors\ \ a \cdot f \in O(b \cdot g)$
- El operador $\Theta()$ es transitivo: $f \in \Theta(g)\ y\ g \in \Theta(h)\ \ \ llavors\ \ f \in \Theta(h)$
- Si $f \in \Theta(g)\ y\ g \in O(h)\ \ llavors\ \ f \in O(h)$
- Si $f \in O(g)\ y\ g \in \Theta(h)\ \ llavors\ \ f \in O(h)$
- El operador $O()$ es transitivo: $f \in O(g)\ y\ g \in O(h)\ \ \ llavors\ \ f \in O(h)$
- Si $f \in \Theta(g)$ llavors $\Theta(f) = \Theta(g),\ \Theta(f) \subset O(g),\ O(f) = O(g)$
- Si $f \in O(g)$ llavors $\Theta(f) \subset O(g),\ O(f) \subseteq O(g)$

- $\Theta(f) + \Theta(g) = \Theta(f+g) = \Theta(\max\{f,g\})$
- $O(f) + O(g) = O(f+g) = O(\max\{f,g\})$
- $\Omega(f) + \Omega(g) = \Omega(f+g) = \Omega(\max\{f,g\})$
- $\Theta(f) \cdot \Theta(g) = \Theta(f \cdot g)$
- $\Theta(f) \cdot O(g) = O(f) \cdot O(g) = O(f \cdot g)$

En el algoritmo anterior de buscar un entero x en un vector de enteros los análisis por casos se denotarían matemáticamente de la siguiente manera:

*caso peor:* θ(n) => se mira los n elementos de la tabla
*caso mejor:* θ(1) => se mira un único elemento, el primero de la tabla

también se puede hacer un caso general donde se dice el coste del algoritmo sin tener en cuenta como está organizada la entrada (cálculo más inexacto), en este algoritmo el coste del algoritmo en caso general sería O(n) => como mucho se mira todos los elementos de la tabla

## Órdenes de magnitud más habituales

Los órdenes de magnitud más utilizados de menor a mayor son:

$$\mathbf{1 < \log n < n < n\log n < n^2 < n^3 < \ldots < n^k < 2^n < 3^n < \ldots < k^n < n! < n^n}$$

- **θ(1):** tiempo constante. Coste de realizar instrucciones básicas como asignaciones, comparaciones ,etc.
- **θ(log n):** logarítmico. Aparece en algoritmos que descartan muchos valores en cada paso, como por ejemplo la cerca dicotómica, que va descartando en cada iteración la mitad de los elementos del subvector a procesar.
- **θ(n):** lineal. cuando nos tenemos que recorrer los n elementos, como por ejemplo la búsqueda del elemento máximo de un vector.
- **θ($n\log n$):** casilineal
- **θ($n^k$):** polinómico. Cuando nos recorremos n veces los n elementos.
- **θ($k^n$):** exponencial. Casos de búsqueda exhaustiva.
- **θ(n!):** factorial. Casos de búsqueda exhaustiva
- **θ($n^n$):** Casos de búsqueda exhaustiva

## Costes de algoritmos iterativos

fórmulas:
θ(a) + θ(b) = θ(max{a,b})
θ(a) * θ(b) = θ(a*b)

- Comparaciones, operaciones aritméticas, lectura/escritura tipo elementales, accesos a un vector => tiene **coste θ(1)**

- Coste de una **asignación**: v := E => coste de evaluar E + coste de copiar sobre v.

*Ejemplo:*
v := x + y => coste de evaluar E: θ(1) + θ(1) = θ(1)
coste de copiar sobre v: θ(1)
coste de esta asignación: θ(1) + θ(1) = θ(1)

- **Condicionales**:
  **Si** E **entonces** A **fsi**
    - caso general:
      coste del **if** = θ(coste E) + O(coste A) si caso peor ≠ caso mejor
      coste del **if** = θ(coste E) + θ(coste A) si caso peor = caso mejor
    - análisis por casos:
      - peor (E es cierto): coste del **if** = θ(coste E) + θ(coste A)
      - mejor (E es falso): coste del **if** = θ(coste E) => no entra en el IF

  **Si** E **entonces** A **sino** B **fsi**
    - caso general:
      coste del **if** = θ(coste E) + O(max{coste A, coste B}) si caso peor ≠ caso mejor
      coste del **if** = θ(coste E) + θ(coste A o coste B) si caso peor = caso mejor
    - análisis por casos:
      - peor: coste del **if** = θ(coste E) + θ(max{coste A, coste B})
      - mejor: coste del **if**= θ(coste E) + θ(min{coste A, coste B})
- **Bucles**:
  **mientras** E **hacer** A **fmientras**
    - caso general:
      coste del bucle = O(nºiteraciones * (coste E + coste A)) si caso peor ≠ caso mejor
      coste del bucle = θ(nºiteraciones * (coste E + coste A)) si caso peor = caso mejor
    - análisis por casos:
      - peor : coste del bucle = θ(nºiteraciones * (coste E + coste A))
      - mejor: coste del bucle = θ(coste E) =>no entra dentro del bucle

  Dicho de otra manera:
  Coste del bucle = coste($E_0$) + $\sum_{i=1}^{n}(\text{coste}(E_i) + \text{coste}(A_i))$
  Siendo:
  $E_0$ el coste de la evaluación de E antes de la 1º iteración
  $E_i$ es el coste de la evaluación de E después de la i-ésima iteración
  $A_i$ es el coste de la evaluación de A en la i-ésima iteración
  N es el número de iteraciones

**NOTA!!**: por lo general, el coste de E será θ(1), ya que se tratarán de expresiones simples.

*Ejemplo:*

```
int pos(const vector<int>& T, int n, int x) {
/*Pre: T no contiene repetidos i 0 <= n <= T.size()*/
   int i = 0;
   bool entontrado = false;
   while (i < n and not encontrado) {
      if(T[i] == x) encontrado = true;
      else i++;
   }
   if (not encontrado) i = -1;
   return i;
}
/*Post: si x no está en T devuelve -1, sino devuelve la posición donde se
encuentra*/
```

**inicialización =>**
i = 1; encontrado = False;
Son asignaciones básicas, θ(1) + θ(1) = θ(1)

**bucle =>**
**condicional interior:**
T[i] == x: comparación básica, θ(1)
encontrado = true: asignación básica, θ(1)
i++, asignación básica, θ(1)

por lo tanto, tanto en caso general, peor o mejor será θ(1) + θ(1) = θ(1)

el bucle se realiza n veces en caso peor: θ(n) * θ(1) = θ(n)
el bucle se realiza 1 vez en caso mejor: θ(1) * θ(1) = θ(1)
el bucle se realiza n veces como mucho en caso general: O(n) * θ(1) = O(n)

**condicional final =>**
not encontrado: comparación básica, θ(1)
i = -1: asignación básica, θ(1)

en caso mejor o peor es θ(1)

**Coste del algoritmo:**

en caso peor = θ(1) + θ(n) +θ(1) = θ(n)
en caso mejor = θ(1) + θ(1) + θ(1) = θ(1)
en caso general= θ(1) + O(n) + θ(1) = O(n)

## Costes de algoritmos recursivos

Para el cálculo del coste de algoritmos recursivos deberemos plantear una recurrencia que resolveremos mediante un teorema maestro.

**Recurrencia para las sustractoras:** en cada llamada recursiva se reduce el tamaño de la entrada.

$$T(n) = \Theta(n^k) + a \cdot T(n-b)$$

Siendo :

- $\Theta(n^k)$ el coste del caso base.
- a: número de llamadas recursivas
- b: lo que se reduce el tamaño de la entrada entre dos llamadas recursivas (factor de decrecimiento)

Usaremos el teorema maestro 1 para calcular el coste de las recurrencias sustractoras:

$$T1(n) = \begin{cases} \theta(n^k) & si\ a < 1 \\ \theta(n^{k+1}) & si\ a = 1 \\ \theta(a^{\frac{n}{b}}) & si\ a > 1 \end{cases}$$

**Recurrencia para las divisoras:** en cada llamada recursiva se divide el tamaño de la entrada.

$$T(n) = \Theta(n^k) + a \cdot T(n/b)$$

Siendo :

- $\Theta(n^k)$ el coste del caso base.
- a: número de llamadas recursivas
- b: lo que se reduce el tamaño de la entrada entre dos llamadas recursivas (factor de división)

Usaremos el teorema maestro 2 para calcular el coste de las recurrencias divisoras:

$$T2(n) = \begin{cases} \theta(n^k) & si\ \alpha < k \\ \theta(n^k \log n) & si\ \alpha = k \\ \theta(n^\alpha) & si\ \alpha > k \end{cases}$$

k = coste de lo que esta elevado g(n)
$\alpha = \log_b a$

*Ejemplo:*

Tenemos el siguiente algoritmo que nos calcula el factorial:

```
int factorial (int x)
{
   //Caso base: Cuando x < 2 devolvemos 1 puesto que 1!=1
   if (x < 2) return 1;
   return x * factorial(x - 1); // Si X >= 2 devolvemos el producto de
                                // 'X' por el factorial de 'X'-1
}
```

Observando el algoritmo podemos sacar las siguiente recurrencia:

$$Tfact(n) = \begin{cases} \theta(1) & si\ n = 0 \vee n = 1\ (caso\ base) \\ Tfact(n-1) + \theta(1) & si\ n > 1\ (caso\ recursivo) \end{cases}$$

a = 1, c = 1, g(n) = θ(1), f(n) = θ(1)

Por lo tanto, viendo que se trata de una recurrencia sustractora y como a = 1 el coste del algoritmo es **$\theta(n^{(k+1)}) = \theta(n)$**

# Algoritmos de divide & vencerás

## Idea

El esquema de divide y vencerás es una técnica para resolver problemas que consiste en dividir el problema original en subproblemas (de menor tamaño que el original), resolver los subproblemas y finalmente combinar las soluciones de los subproblemas para dar una solución al problema original. Si los subproblemas son similares al problema original entonces puede utilizarse el mismo procedimiento para resolver los subproblemas recursivamente.
En cada nivel del procedimiento consta de los siguientes tres pasos:

**1. Dividir el problema en subproblemas.**
**2. Resolver cada uno de los subproblemas.**
**3. Combinar las soluciones de los subproblemas para obtener la solución del problema original.**

## Algoritmo de búsqueda dicotómica

También llamado binary-searh, se trata de un algoritmo típico que utiliza el esquema divide y vencerás. El problema se trata de buscar la posición donde iría un elemento x en un vector T ordenado crecientemente. El algoritmo nos retornará:

- T[0]<=x<=T[n-1] (retorna la posición donde iría el elemento x en el vector T, donde esa posición indica que T[i]<=x<T[i+1])
- x<T[0] (retorna –1)
- x>T[n-1] (retorna -1)

```
int bsearch(const vector<int>& T, int x, int i, int j) {
   if (i > j) return -1;
   int m = (i+j)/2;
   if (T[m] == x) return m;
   else if (x < T[m]) return bsearch(T,x,i,m-1);
   else return bsearch(T,x,m+1,j);
}
```

El coste del binary-search sería, sacando su recurrencia:

$$Tbs(n) = \begin{cases} \theta(1) & si\ n = 2 \\ Tbs\left(\frac{n}{2}\right) + \theta(1) & si\ n > 2 \end{cases}$$

a = 1, b = 2, g(n) = θ(1), f(n) = θ(1)

Por lo tanto, viendo que se trata de una recurrencia divisora, k = 0 y $\alpha = \log_2 1 = 0$, $(\alpha = k)$, el coste del algoritmo es **$\theta(n^k \log n) = \theta(\log n)$**

# Algoritmos básicos de ordenación

## Selección

En cada momento "seleccionamos" el elemento más pequeño de la parte no ordenada y lo insertamos ordenadamente en la parte ya ordenada. EL COSTE ESTA EN SELECCIONAR

```
void selección (vector<int>& T) {
   int n = T.size();
   for (int i = 0; i < n-1; ++i) {
      int p = pos_min(T,i,n-1);
      swap(T[i],T[p]);
   }
}

int pos_min (vector<int>& T, int i, int n) {
   int p = i;
   for (int j = i+1; j <= n; ++j)
      if (T[j] < T[p]) p = j;
   return p;
}
```

Coste de la función pos_min:

$$\sum_{j=i}^{n} 1 = n - i$$

Coste del algoritmo selección, para el caso peor, mejor y medio:

$$\sum_{i=0}^{n}(n-i) = \sum_{i=0}^{n} n - \sum_{i=0}^{n} i = n(n+1) - \frac{n(n+1)}{2} = \frac{n^2+n}{2} \in \theta(n^2)$$

## Inserción

En cada iteración se selecciona el primer elemento de la parte no ordenada y se inserta ordenadamente en la parte ordenada. EL COSTE ESTA EN INSERTAR

```
void inserción(vector<elem>& T) {
   int n = T.size();
   for (int i = 1; i < n; ++i) {
      int x = T[i];
      int j;
      for (j = i; j > 0 and T[j-1] > x; --j)
         T[j] = T[j-1];
      T[j] = x;
   }
}
```

Coste del algoritmo de inserción, en caso peor y medio:

$$\sum_{i=1}^{n}\sum_{j=0}^{i} 1 = \sum_{i=1}^{n} i = \frac{n(n+1)}{2} = \frac{n^2 + n}{2} \in \theta(n^2)$$

En caso mejor (cuando el vector esta ordenado el bucle interior no realiza ninguna iteración):

$$\sum_{i=1}^{n} 1 = n \in \theta(n)$$

## Burbuja

Para todas las posiciones del vector, coge el último elemento de éste y hace que "flote" de manera ordenada hasta su posición adecuada. Con esto conseguiremos en cada iteración decir cuál es el mínimo. PEOR ALGORITMO DE ORDENACIÓN, REALIZA MUCHOS SWAPS

```
void burbuja (vector<int>& T) {
   int n = T.size();
   bool b = true;
   for (int i = 0; i < n-1 and b; ++i) {
      b = false;
      for (int j = n-1; j > i; --j) {
         if (T[j-1] > T[j]) {
            swap(T[j-1],T[j]);
            b = true;
         }
      }
   }
}
```

Coste del algoritmo de la burbuja, en caso peor y medio:

$$\sum_{i=0}^{n}\sum_{j=i}^{n} 1 = \sum_{i=0}^{n}(n-i) = \frac{n(n+1)}{2} = \frac{n^2 + n}{2} \in \theta(n^2)$$

En caso mejor (cuando el vector esta ordenado, en la primera ejecución del bucle interior la variable bool b no llegará a ponerse a true, y por lo tanto, en el bucle exterior no se cumplirá la condición):

$$\sum_{i=0}^{1}\sum_{j=0}^{n-i} 1 = \sum_{i=0}^{1}(n-i) = n + n - 1 = 2n - 1 \in \theta(n)$$

## MergeSort (ordenación por fusión)

El MergeSort utiliza el esquema de divide & vencerás:
- dividir: divides en dos el vector (tendremos dos subproblemas).
- resuelve el subproblema: ordena los dos subvectores recursivamente con MergeSort.
- combinar: fusiona los dos subvectores, sabiendo que cada uno de ellos ya esta ordenado para obtener un vector ordenado.

Algoritmo MergeSort para vectores:

```
void mergeSort (vector<int>& T, int i, int j) {
   if(i < j) {
      int m = (i+j)/2;
      mergeSort(T,i,m);
      mergeSort(T,m+1,j);
      fusion(T,i,m,j);
   }
}

=>hacer la fusión tiene coste θ(n)
void fusion (vector<int>& T, int i, int m, int j) {
//aquí se declara el vector auxiliar => no in-situ. Si en vez de ordenar
//un vector fuera una lista sería in-situ, se podría realizar la fusión
//en la misma lista
   vector<int> B(j-i+1);
   int i' = i, j' = m+1, k = 0;
   while (i' <= m and j' <= j) {
      if (T[i'] <= T[j']) B[k++] = T[i'++];
      else B[k++] = T[j'++];
   }
   while (i' <= m) B[k++] = T[i'++];
   while (j' <= d) B[k++] = T[j'++];
   for (k = 0; k <= j-i; ++k) T[i+k] = B[k];
}
```

Si miramos el coste a simple vista, ir dividiendo el vector en dos tendría un $\Theta(\log_2 n) = \Theta(\log n)$ y como en cada subdivisión realizamos la fusión que tiene un coste θ(n), por lo tanto el algoritmo tiene un coste de θ(nlogn).

El coste del MergeSort sería, sacando su recurrencia: en caso peor, medio o mejor ya que no varía el comportamiento del algoritmo dependiendo de la organización de la entrada.

$$Tms(j-i+1) = Tms(n) = \begin{cases} \theta(n^2) & si\ n = 1 \\ 2Tms\left(\frac{n}{2}\right) + \theta(n) & si\ n > 1 \end{cases}$$

a = 2, b = 2, g(n) = θ(n), f(n) = $\theta(n^2)$

Por lo tanto, viendo que se trata de una recurrencia divisora, k = 1 y $\alpha = \log_2 2 = 01, (\alpha = k)$, el coste del algoritmo es $\boldsymbol{\theta(n^k \log n) = \theta(n \log n)}$

## QuickSort (ordenación rápida)

El QuickSort utiliza el esquema de divide & vencerás:

- dividir: se elige un pivote (un elemento del vector elegido al azar o por otros sistemas y colocas los elementos menores que el pivote a su izquierda y los mayores o igual que el pivote a su derecha.
- resuelve el problema: ordena los dos subvectores recursivamente con QuickSort.
- combinar: como cada subvector se ordena in-situ no requiere trabajo.

Algoritmo QuickSort para vectores, para elegir el pivote usaremos el método de Hoare(intenta elegir una buena posición para el pivote):

```
void quickSort (vector<int>& T, int i, int j) {
   if (i < j) {
      int q = partition(T,i,j);
      quickSort(T,i,q);
      quickSort(T,q+1,j);
   }
}

=> ordena a partir del pivote que elige el propio algoritmo, tiene coste
θ(n)
int partition (vector<int>& T, int i, int j) {
   int x = T[i], i' = i-1, j' = j+1;
   for (;;) {
      while (x < T[--j']);
      while (T[++i'] < x);
      if (i' >= j') return j';
      swap(T[i'],T[j']);
   }
}
```

Como en el caso del MergeSort, el vector se va dividiendo en dos por lo que tendría un coste de aproximadamente $\Theta(\log_2 n) = \Theta(\log n)$ (digo aprox. porque el caso perfecto sería que el pivote siempre dividiera el vector en dos, que aprox. sucede en el caso mejor y promedio), y como la ordenación del vector a partir del pivote tiene un coste de θ(n) el algoritmo QuickSort tiene un coste θ(nlogn). Este coste como ya he dicho sería en el caso mejor y promedio. En el caso peor, donde la elección del pivote fuera irregular, que por ejemplo dividiera el vector en dos tamaños, el primero de 1 posición y el segundo de n-1 posiciones, así con todas las llamadas recursivas, entonces entraríamos en el caso peor del algoritmo donde tendría un coste de $\theta(n^2)$. A pesar de tener un coste peor mayor al MergeSort se le considera el algoritmo más rápido de ordenación para vectores, tan solo hay que tener un buen algoritmo para elegir el pivote y no llegar al caso peor. Otra cosa a tener en cuenta es que al ordenar el vector a partir del pivote se incumple la propiedad de estabilidad y por lo tanto se convierte en un algoritmo no estable.

El coste del QuickSort sería, sacando su recurrencia:

En el caso mejor o promedio( teorema maestro 2 para recurrencias divisoras):

$$Tqs(j-i+1) = Tqs(n) = \begin{cases} \theta(1) & si\ n = 1 \\ 2Tqs\left(\frac{n}{2}\right) + \theta(n) & si\ n > 1 \end{cases}$$

a = 2, b = 2, g(n) = θ(n), f(n) = θ(1)

Por lo tanto, viendo que se trata de una recurrencia divisora, k = 1 y $\alpha = \log_2 2 = 1$, $(\alpha = k)$, el coste del algoritmo es **$\theta(n^k \log n) = \theta(n \log n)$**

En el caso peor( teorema maestro 1 para recurrencias sustractoras):

$$Tqs(j-i+1) = Tqs(n) = \begin{cases} \theta(1) & si\ n = 1 \\ Tqs(1) + Tqs(n-1) + \theta(n) & si\ n > 1 \end{cases}$$

a = 1, c = 1, g(n) = θ(n), f(n) = θ(1)
Por lo tanto, viendo que se trata de una recurrencia sustractora, k = 1 y a = 1 (a $= k$), el coste del algoritmo es **$\theta(n^{k+1}) = \theta(n^2)$**

## QuickSelect

El algoritmo de QuickSelect se utiliza para saber qué elemento ocuparía una posición k en un vector T (no ordenado) si llegase a estar ordenado. Se trata de una mezcla de QuickSort y búsqueda dicotómica:

- Primero ordena el vector T usando la técnica del pivote. Colocando los elementos menores que el pivote a su izquierda y los mayores o igual que el pivote a su derecha.

- Luego podemos saber si la posición del elemento k estaría en el subvector uno o en el dos, y por lo tanto, llamamos recursivamente a QuickSelect tan solo con ese subvector.

```
int quickSelect (vector<int>& T, int i, int j, int k) {
   if (i - j > 0) {
      int q = partition(T,i,j);
      if ( q == k) return T[q];
      if (k < q) quickSelect(T,i,q-1,k);
      else quickSelect(T,q+1,j,k);
   }
}
```

Este algoritmo a simple vista tiene un coste de ordenar respecto el pivote θ(n) y un coste θ(logn) para ir dividiendo el vector. Pero como sólo llamamos con una parte del vector el coste del algoritmo sería algo del estilo: max{θ(n),θ(logn)}= θ(n).

A continuación sacamos la recurrencia de este algoritmo:

En el caso mejor o promedio( teorema maestro 2 para recurrencias divisoras:

$$Tsel(n) = \begin{cases} \theta(1) & si\ n = 1 \\ Tsel\left(\frac{n}{2}\right) + \theta(n) & si\ n > 1 \end{cases}$$

a = 1, b = 2, g(n) = θ(n), f(n) = θ(1)

Por lo tanto, viendo que se trata de una recurrencia divisora, k = 1 y $\alpha = \log_2 1 = 0$, $(\alpha < k)$, el coste del algoritmo es **$\theta(n^k) = \theta(n)$**

## Karatsuba

El algoritmo de Karatsuba es un procedimiento para multiplicar números grandes eficientemente, que se basa en divide y vencerás.

Dividir : Los números x, y de *n* dígitos se dividen en 2 mitades.

- $x = a10^{n/2} + b$ $x =$ | a | b |
- $y = c10^{n/2} + d$ $y =$ | c | d |

Resuelve el problema: Hacemos multiplicaciones de números de n/2 dígitos
Combinar: Se añaden ceros y se hacen sumas.

La multiplicación de 2 números de *n* dígitos sería:

$xy = (a10^{n/2} + b)\ (c10^{n/2} + d) = ac10^{n} + (ad + bc)10^{n/2} + bd$

El coste de este algoritmo seria: Hacemos 4 llamadas recursivas (ac, ad, bc, bd) con la mitad de dígitos, y el coste de la parte no recursiva es lineal, añadimos ceros y sumamos.

$$Tmult(n) = \begin{cases} \theta(1) & si\ n \leq 1 \\ 4Tmult\left(\frac{n}{2}\right) + \theta(n) & si\ n > 1 \end{cases}$$

a = 4, b = 2, g(n) = θ(n)

Por lo tanto, viendo que se trata de una recurrencia divisora, k = 1 y $\alpha = \log_2 4 = 2\,, (\alpha > k)$, el coste del algoritmo es **$\theta(n^{\alpha}) = \theta(n^2)$**

Karatsuba resuelve el mismo problema haciendo 3 multiplicaciones:

$xy = ((a+b)\ (c+d) - ac - bd)\ 10^{n/2} + ac\ 10^{n} + bd$

El coste de Karatsuba seria: Hacemos 3 llamadas recursivas (ac, bd, (a+b)(c+d)) con la mitad de dígitos, y el coste de la parte no recursiva es lineal, añadimos ceros y sumamos.

$$Tkarat(n) = \begin{cases} \theta(1) & si\ n \leq 1 \\ 3TKarat\left(\frac{n}{2}\right) + \theta(n) & si\ n > 1 \end{cases}$$

a = 3, b = 2, g(n) = θ(n)

Por lo tanto, viendo que se trata de una recurrencia divisora, k = 1 y $\alpha = \log_2 3 = 1{,}58, (\alpha > k)$, el coste del algoritmo es **$\theta(n^{\alpha}) = \theta(n^{1{,}58})$**

El mismo problema se puede resolver con *n* bits. $xy = ((a+b)\ (c+d) - ac - bd)\ 2^{n/2} + ac\ 2^{n} + bd$

## Strassen

El algoritmo de Strassen es un procedimiento para multiplicar dos matrices A, B de *n x n,* que se basa en divide y vencerás.

Dividir : Las matrices A, B se dividen en 4 cuadrantes

A = $\begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}$ B = $\begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$ C = $\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}$

Resuelve el problema: Hacemos multiplicaciones de números de matrices de *n/2 x n/2*

Combinar: Se hacen sumas.

La multiplicación de 2 matrices A,B de *n x n*:

$C_{11} = A_{11}B_{11} + A_{12}B_{21}$
$C_{12} = A_{11}B_{12} + A_{12}B_{22}$
$C_{21} = A_{21}B_{11} + A_{22}B_{21}$
$C_{22} = A_{21}B_{12} + A_{22}B_{22}$

El coste de este algoritmo seria: Hacemos 8 llamadas recursivas con la mitad del tamaño de la matriz original , y el coste de la parte no recursiva es cuadratico, sumamos.

$$Tmat(n) = \begin{cases} \theta(1) & si\ n \leq 1 \\ 8Tmat\left(\frac{n}{2}\right) + \theta(n^2) & si\ n > 1 \end{cases}$$

a = 8, b = 2, g(n) = $\theta(n^2)$

Por lo tanto, viendo que se trata de una recurrencia divisora, k = 2 y $\alpha = \log_2 8 = 3$ , $(\alpha > k)$, el coste del algoritmo es **$\theta(n^{\alpha}) = \theta(n^3)$**

Strassen resuelve el mismo problema haciendo 7 multiplicaciones:

$M_1 = (A_{11} + A_{22})\ (B_{11} + B_{22})$
$M_2 = (A_{21} + A_{22})\ B_{11}$
$M_3 = A_{11}\ (B_{12} - B_{22})$
$M_4 = A_{22}\ (B_{21} - B_{11})$
$M_5 = (A_{11} + A_{12})\ B_{22}$
$M_6 = (A_{21} - A_{11})(B_{11} + B_{12})$
$M_7 = (A_{12} - A_{22})(B_{21} + B_{22})$

Y volvemos a definir C:

$C_{11} = M_1 + M_4 - M_5 + M_7$
$C_{12} = M_3 + M_5$
$C_{21} = M_2 + M_4$
$C_{22} = M_1 - M_2 + M_3 + M_6$

El coste seria

$$Tstra(n) = \begin{cases} \theta(1) & si\ n \leq 1 \\ 7Tstra\left(\frac{n}{2}\right) + \theta(n^2) & si\ n > 1 \end{cases}$$

a = 7, b = 2, g(n) = $\theta(n^2)$

Por lo tanto, viendo que se trata de una recurrencia divisora, k = 2 y $\alpha = \log_2 7 = 2{,}81$, $(\alpha > k)$, el coste del algoritmo es **$\theta(n^{\alpha}) = \theta(n^{2,81})$**

# Estructuras de Datos

## Idea

Para comenzar este tema definiremos las siguientes estructuras:

- Un Tipo Abstracto de Datos (TAD) es un conjunto de elementos junto con las operaciones definidas en él, independientemente de su implementación.
- El TAD diccionario es un conjunto S de elementos con una clave y una información asociadas. Las claves deben ser comparables entre ellas. Las operaciones definidas para el TAD diccionario son: crear un diccionario vacío, determinar si está o no un elemento, insertar un elemento y borrar un elemento.
- Un TAD cola de prioridad es un conjunto S de elementos en el que cada elemento x tiene una prioridad asociada. Las operaciones asociadas al conjunto de elementos son: crear una cola de prioridad vacía, insertar un elemento, buscar el elemento con prioridad mínima (o máxima, pero no ambas) y borrar el elemento con prioridad mínima (o máxima, pero no ambas).

En este tema veremos los siguientes TAD diccionarios:

- Tablas de hash
- Árbol binario de búsqueda (denominado como BST, ACB o ABB)
- Árbol binario de búsqueda equilibrado (denominado como AVL)

En este tema veremos los dos siguientes TAD cola de prioridad:

- Heaps (mín-heap y máx-heap) (denominado como montículo)

Por último, veremos un algoritmo de ordenación basado en los heaps, llamado HeapSort.

## 1. TAD Diccionario

Un diccionario es un conjunto finito de pares (k, v) con $k \in K$ (conjunto de claves) y $v \in V$ (conjunto de valores), donde no hay dos pares diferentes con la misma clave. El TAD diccionario se puede implementar eficientemente usando diferentes estructuras de datos que veremos a continuación.

## Tablas de hash

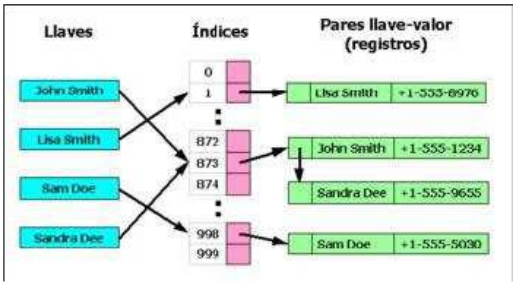
Una **tabla de hash** es una forma de implementar un diccionario que resulta útil sobre todo para búsquedas aleatorias de los elementos. Se suele implementar en tablas, aunque se pueden hacer implementaciones multi-dimensionales basadas en varias claves. Funciona transformando la clave con una **función hash** en un número que la tabla hash utiliza para localizar el valor deseado en la tabla. Una buena función hash es esencial para que no haya colisiones, es decir, dos claves asignadas con la función de hash a la misma posición del vector. Las colisiones son generalmente resueltas por algún tipo de búsqueda lineal, así que si la función hash tiende a generar valores similares, las búsquedas resultantes se vuelven lentas.

La resolución de colisiones se puede hacer de las siguientes dos maneras, entre otras:

Primero definimos MAX como el tamaño de la tabla y α como el factor de carga de la tabla de hash, es decir, $\alpha = n/MAX$.

### Tabla de hash con listas encadenadas (separate chaining)

En caso de colisión se crea una lista encadenada asociada a esa posición de la tabla, donde iremos guardando las colisiones asociadas a dicha posición.




Ejemplo de tabla de hash con listas encadenadas

Ventajas: borrado es fácil y no hace falta crecer la tabla

La tabla de hash con listas encadenadas tiene las siguientes operaciones:

- **crear:** reserva espacio para la tabla de tamaño MAX, por lo tanto tiene un coste θ**(1)**

- **buscar:** se aplica la función de hash a la clave del elemento a buscar y retornará la posición de la tabla donde se encuentra la lista donde estará el elemento.

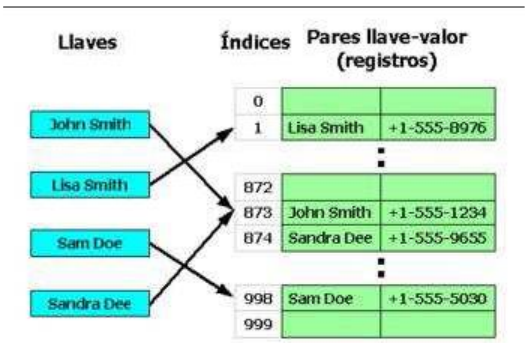
- **insertar:** se aplica la función de hash a la clave del elemento a insertar y retornará la posición de la tabla donde lo insertaremos al principio de la lista.

- **borrar:** primero buscamos el elemento y luego lo borramos de la lista encadenada.

En **caso peor para buscar y borrar** tiene un coste de θ**(n)**, que sucederá cuando todas las claves de los elementos van a la misma posición de la tabla (muchas colisiones => mala función de hash), por lo tanto, tendríamos una lista con todos los elementos. En **caso medio para buscar, borrar e insertar** y el **caso peor para insertar** tiene un coste de **O(**α**)**, donde α respecto a n equivale a una constante (mínimas colisiones => buena función de hash), es decir, **O(1)**.

## Tabla de hash con direccionamiento abierto (open addressing)

En caso de colisión se guarda el elemento a insertar en la siguiente posición libre de la tabla.




Ejemplo de tabla de hash con direccionamiento abierto

Desventajas: borrado es difícil y hace falta crecer la tabla cuando esté llena

La tabla de hash con direccionamiento abierto tiene las siguientes operaciones:

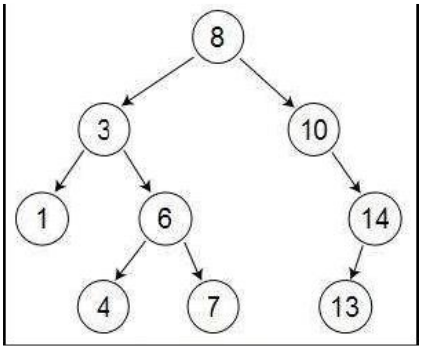
- **crear:** reserva espacio para la tabla de tamaño MAX, por lo tanto tiene un coste θ**(1)**
- **buscar:** se aplica la función de hash a la clave del elemento a buscar y retornará la posición de la tabla donde se encuentra el elemento, en caso de no estar en esa posición puede deberse a una colisión, entonces tendrá que buscar el elemento en la tabla siguiendo una búsqueda lineal.
- **insertar:** se aplica la función de hash a la clave del elemento a insertar y retornará la posición de la tabla donde lo insertaremos, en caso de colisión buscará la primera posición libre a partir de esta posición siguiendo una búsqueda lineal que será donde lo insertaremos.
- **borrar:** primero buscamos el elemento y luego lo borramos de la tabla.

En **caso peor para buscar, borrar e insertar** tiene un coste de θ**(n)**, que sucederá cuando todas las claves de los elementos van a la misma posición de la tabla (muchas colisiones => mala función de hash). En **caso medio para buscar, borrar e insertar** tiene un coste de **O(**α**)**, donde α respecto a n equivale a una constante (mínimas colisiones => buena función de hash), es decir, **O(1)**.
Otra resolución de colisiones es colocar un árbol binario de búsqueda (ABB) o un árbol binario de búsqueda equilibrado (AVL) asociado a cada una de las posiciones de la tabla. Así conseguiremos mejorar los casos peores, por lo tanto, en caso peor para buscar, borrar e insertar tendrá el coste que tenga el ABB o el AVL en el caso peor para buscar, borrar e insertar, respectivamente.

## Árbol Binario de Búsqueda

Un **ABB** es una forma de implementar un diccionario que resulta útil sobre todo para recorrer en orden los elementos según su clave. El ABB se define como un árbol binario que cumple las siguientes condiciones:

- si es un árbol binario vacío es un ABB
- si no es un árbol binario vacío se tiene que cumplir :
    - Los subárboles izquierdo y derecho son árboles binarios de búsqueda.
    - La clave de la raíz es mayor que toda clave del subárbol izquierdo y menor que toda clave del subárbol derecho.




Ejemplo de ABB

Si hacemos el recorrido en inorden, obtenemos los elementos en orden creciente de clave

*Ejemplo:* Si escribimos el ABB del dibujo en inorden obtenemos:
[1, 3, 4, 6, 7, 8. 10, 13, 14]

La definición usual en c++ del ABB es (las variables pueden tener otro nombre):

```
typedef node* abb;

struct node{
   int clave;
```

```
    Info info;
    node* hijoIzq;
    node* hijoDer;
}
```

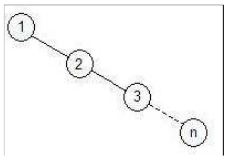
Como ya se comento en la definición de TAD diccionario, el ABB tiene las siguientes operaciones:

- **crear:** inicializa un árbol binario de búsqueda vacío, por lo tanto tiene un coste θ**(1)**

- **buscar:** según sea la clave Y del nodo que buscamos iremos por el hijo izquierdo (Y<clave) o por el hijo derecho (Y>clave), así sucesivamente hasta encontrarlo o llegar a un nodo nulo.

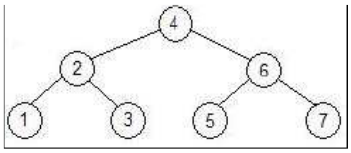
- **insertar:** según sea la clave Y del nodo que insertamos iremos por el hijo izquierdo (Y<clave) o por el hijo derecho (Y>clave), así sucesivamente hasta llegar a un nodo nulo que será donde insertaremos el nodo. Teniendo en cuenta que no puede haber dos nodos con la misma clave.
- **borrar:** según sea la clave Y del nodo que borramos iremos por el hijo izquierdo (Y<clave) o por el hijo derecho (Y>clave), así sucesivamente hasta encontrarlo (aprioris hay que asegurarse que existe el nodo con clave Y en el árbol) y lo borraremos. Al borrar hay que tener en cuenta:
    - Si el nodo tiene UNO de sus subárboles no vacío hay que reemplazarlo por la raíz de este subárbol.
    - Si el nodo tiene los DOS subárboles no vacíos hay que reemplazarlo por el nodo del subárbol derecho con menor clave.

El **caso peor para buscar, insertar o eliminar** es cuando el árbol en realidad se comporta como una lista, y tengamos que buscar, insertar o eliminar un elemento que se encuentre al final:



(lista) caso peor: θ**(n)**

El **caso medio para buscar, insertar o eliminar** tiene un coste **O(h)** donde la h indica la altura. La h puede tomar **como mucho el valor n**, como en el caso peor anterior (el árbol es una lista) o puede tomar **como poco el valor *log n*:** (el árbol está balanceado):



árbol balanceado

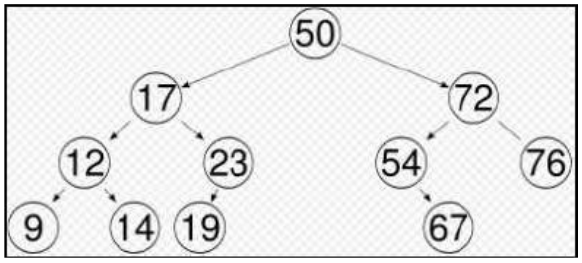
El **caso mejor para borrar, buscar o insertar** un elemento es θ**(1)**, el elemento a buscar o eliminar esta en la raíz del árbol; o el elemento a insertar esta a continuación de la raíz.

## Árbol binario de búsqueda equilibrado

Un **AVL** mejora al ABB en cuanto a su coste en caso peor, ya que intenta evitar que todos los elementos se puedan colocar en forma de lista. El AVL se define como un árbol binario que cumple las siguientes condiciones:

- si es un árbol binario vacío es un AVL

- si no es un árbol binario vacío se cumplen las mismas propiedades que para el ABB y además que para cada nodo del árbol hay una diferencia de altura entre su hijo izquierdo y su hijo derecho de cómo mucho 1.



Ejemplo de AVL

La definición usual en c++ del AVL es (las variables pueden tener otro nombre):

```
typedef node* avl;

struct node{
   int clave;
   Info info;
   node* hijoIzq;
   node* hijoDer;
   int h;
}
```

Como ya se comento en la definición de TAD diccionario, el AVL tiene las siguientes operaciones:

- **crear:** inicializa un árbol binario de búsqueda vacío, por lo tanto tiene un coste θ(1)
- **buscar:** funciona de la misma forma que la búsqueda en ABB
- **insertar:** funciona de la misma forma que la búsqueda en ABB, tan solo que después de insertar el elemento puede que el árbol quede cojo, es decir, que la diferencia de altura entre el hijo izquierdo y el hijo derecho de un/os nodo/s sea igual a dos, y por lo tanto, se deberá resolver dicha cojera.
- **borrar:** funciona de la misma forma que la búsqueda en ABB, tan solo que después del borrado nos podemos encontrar con el problema de la cojera.

El **caso peor y el caso medio para buscar, insertar y borrar** un elemento es θ**(log n)**, ya que al ser un árbol equilibrado se aproxima más al caso del árbol balanceado que al caso de la lista del ABB. El **caso mejor para buscar un elemento** es θ**(1)**, ya que puede encontrarse el elemento a buscar en la raíz del árbol; el **caso mejor para insertar o borrar** un elemento es θ**(logn)**, ya que los elementos se insertan en las hojas o al borrar necesitamos el elemento más grande del subárbol izquierdo o el elemento más pequeño del subárbol derecho.

**NOTA:** el recorrido en inorden de un AVL da como resultado una lista ordenada crecientemente de elementos, igual que el ABB.

**Como corregimos la cojera:** El reequilibrio se produce de abajo hacia arriba sobre los nodos en los que se produce el desequilibrio. Tenemos 4 casos:

**1. Rotación simple a la derecha:**
Ocurre cuando insertamos en subárbol izquierdo del hijo izquierdo. Formaremos un árbol nuevo con la raíz del hijo izquierdo, como hijo izquierdo colocamos el subárbol izquierdo del hijo izquierdo y como hijo derecho creamos un árbol que tendrá como raíz la raíz del árbol, como hijo izquierdo el subárbol derecho del hijo izquierdo y como hijo derecho el hijo derecho del árbol.

**2. Rotación simple a la izquierda:**
Ocurre cuando insertamos en subárbol derecho del hijo derecho. Formaremos un árbol nuevo con la raíz del hijo derecho, como hijo derecho colocamos el subárbol derecho del hijo derecho y como hijo izquierdo creamos un árbol que tendrá como raíz la raíz del árbol, como hijo izquierdo el hijo izquierdo del árbol y como hijo derecho el subárbol izquierdo del hijo derecho.
**3. Rotación doble a la derecha (rotación derecha-izquierda):**
Ocurre cuando insertamos en subárbol izquierdo del hijo derecho. Será como hacer una rotación simple a la derecha del hijo derecho y una rotación simple a la izquierda del árbol.
**4. Rotación doble a la izquierda (rotación izquierda-derecha):**
Ocurre cuando insertamos en subárbol derecho del hijo izquierdo. Será como hacer una rotación simple a la izquierda del hijo izquierdo y una rotación simple a la derecha del árbol

## 2. TAD Cola de prioridad

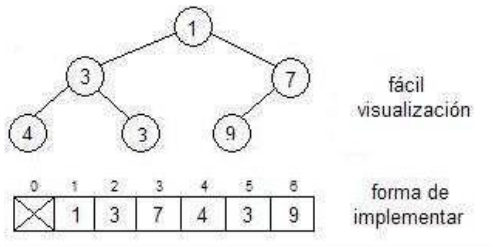
Un Cola de prioridad es una secuencia de elementos con un valor asociado(*prioridad)*, donde las consultas y eliminaciones sólo son del elemento de prioridad mínima.

## Heaps (mín-heap y máx-heap)

Un **Heap** es una forma de implementar una cola de prioridad. Con un heap conseguiremos buscar el mínimo (caso de mín-heap) o el máximo (caso del máx-heap) con un coste constante, es decir, θ(1). El mín-heap se define como un árbol binario que cumple las dos siguientes condiciones:

- si es un árbol binario vacío es un mín-heap
- si no es un árbol binario vació cumple las dos propiedades siguientes:
    - **propiedad de orden**: para todo nodo se cumple que su prioridad asociada es menor o igual que las prioridades asociadas a las raíces de sus dos hijos.
    - **propiedad de forma**: se trata de un árbol quasi-completo de tamaño n, es decir, la distancia de cualquier nodo a la raíz es como mucho $\lfloor \log n \rfloor$ y todos los nodos del último nivel del árbol se encuentran "lo más a la izquierda posible".

En el caso del máx-heap sólo cambia la propiedad de orden.
La forma de implementarse los Heaps más eficientemente es en una tabla de tamaño n+1 (la posición 0 de la tabla no se utiliza para no complicar el acceso), ya que al tener la propiedad de forma se puede conseguir pasar un árbol binario a una tabla por niveles sin que haya huecos vacíos en la tabla. Así conseguimos una mayor eficiencia al acceder al hijo izquierdo, hijo derecho y padre de un nodo cualquiera.




Ejemplo de un mín-heap

La forma de acceder usando la implementación de la tabla es la siguiente:

- Padre del nodo que está en la posición i está en la posición $\lfloor i/2 \rfloor$ donde $1 < i \leq n$
- Hijo izquierdo del nodo que está en la posición i está en la posición 2*i donde $2 * i \leq n$
- Hijo derecho del nodo que está en la posición i está en la posición 2*i+1 donde $2*i+1 \leq n$

Ejemplos: Padre(3)= $\lfloor 3/2 \rfloor$=1. HIZQ(3)=2*3=6. HDER(1)=2*1+1=3

Como ya se comento en la definición de TAD cola de prioridad, el mín-heap tiene las siguientes operaciones:

- **crear:** inicializa un tabla vacía, por lo tanto tiene un coste θ**(1)**
- **buscar mínimo:** accede a la posición 1 del vector, por lo tanto tiene un coste θ**(1)**
- **insertar:** inserta el elemento al final de la tabla o si me refiero al árbol lo inserta en el último nivel "lo más a la izquierda posible" (para cumplir la propiedad de forma) y luego hacemos que el elemento "flote" hasta su posición correcta, es decir, hacemos intercambios con el padre hasta encontrar un padre con prioridad igual o menor que su prioridad (para cumplir la propiedad de orden)
- **borrar mínimo:** borra el elemento de la posición 1 del vector, luego coge el elemento que esta al final de la tabla o si me refiero al árbol coge el elemento que está en el último nivel "lo más a la derecha posible" y lo coloca en la posición 1 del vector (para cumplir la propiedad de forma) y luego hacemos que el elemento se "hunda" hasta su posición correcta, es decir, hacemos intercambios con el hijo izquierdo o hijo derecho (el menor de los dos) hasta encontrar un hijo izquierdo e hijo derecho con prioridad igual o mayor que su prioridad (para cumplir la propiedad de orden)

El coste en **caso peor para insertar o borrar el mínimo** es θ**(log n)** ya que puede que tengamos que patearnos verticalmente el árbol porque al flotar/hundir su posición correcta se encuentre en la otra punta. En cambio, el **caso mejor para insertar o borrar el mínimo** es θ**(1)** ya que puede que no haga falta flotar o hundir el elemento ya que directamente se encuentra en su posición correcta. Para que se dé el caso mejor en borrar el mínimo todas las claves han de ser iguales.
A continuación se explica la operación de heapify, que sirve para que dada una tabla de n elementos (o un árbol quasi-completo) se trasforme a un Heap. Tenemos tres maneras:

- **manera directa:** una tabla ordenada es un mín-heap, por lo tanto, tendría un coste θ**(nlogn)** del quicksort por ejemplo
- **top_down:** simulamos que comenzamos con una tabla vacía e insertamos los n elementos con la operación de insertar de los Heaps. Insertaríamos n elementos y por cada elemento tendríamos el coste de logn de insertar, por lo tanto, el coste es θ**(nlogn)**
- **bottom_up:** el procedimiento de esta operación se explicará en clase. El coste que tiene es θ**(n)**, por lo tanto, resulta ser el mejor procedimiento para implementar la operación de heapify

## HeapSort

El **HeapSort** (ordenamiento por montículos) es un algoritmo de ordenación no recursivo.

Este algoritmo consiste en almacenar todos los elementos del vector a ordenar en un mín-heap, y luego aplicar la búsqueda del mínimo y borrado del mínimo en sucesivas iteraciones obteniendo el conjunto ordenado. Funciona ya que la cima del mín-heap es siempre el elemento mínimo.
El coste de construir el mín-heap es θ(n) (si usamos el procedimiento bottom_up) y el coste de borrar el mínimo es O(logn) en caso general, y como tiene que borrar n elementos tiene un coste de O(nlogn). Por lo tanto el **coste del HeapSort** es: θ(n)+O(nlogn)= máx{θ(n),O(nlogn)}= **O(nlogn)**

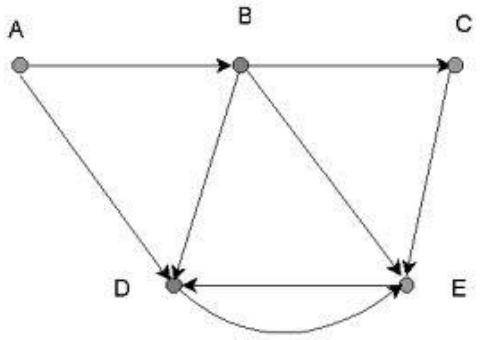
# Grafos

Un **grafo G = <V, E>** donde **V** es un conjunto finito de vértices (nodos) y **E** $\subseteq$ VxV es un conjunto de arcos, es decir, E={ (u,v) | u,v $\in$ V}.

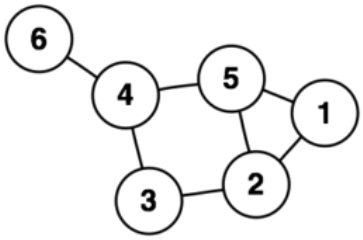
Un grafo puede ser:

- **dirigido o dígrafo:** dado un arco (u,v) sólo podemos navegar de u a v, es decir, v es **incidente** al vértice u:
  - $V \neq \varnothing$
  - $E \subseteq \{(a, b) \in V \times V: a \neq b\}$ es un conjunto de pares ordenados de elementos de V. Dada una arista (*a*,*b*), *a* es su *nodo inicial* y *b* su *nodo final*.

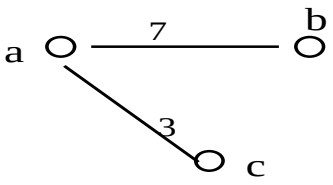
  Por definición, los grafos dirigidos no contienen *bucles*.



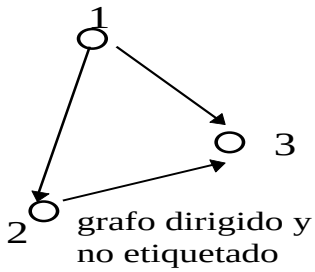

- **no dirigido:** dado un arista (u,v) podemos navegar de u a v y de v a u, es decir, u y v son **adyacentes**




**-Etiquetado:** las aristas tienen información asociada.




grafo NO dirigido y etiquetado




grafo dirigido y no etiquetado

En un grafo G = (V, E), decimos que un vértice $v \in V$ es **adyacente** a un vértice $u \in V$ si y sólo si:

- {u,v} ∈ E en caso de grafos no dirigidos.
- (u,v) ∈ E en caso de grafos dirigidos.

Un **camino C** de longitud n ≥ 0 en el grafo G es una sucesión de vértices, es decir, C=(v1, v2,….vk) donde v1 es el vértice inicial y vk es el vértice final. Además si:

- v1 ≠ vk => es un **camino abierto**
- v1 = vk => es un **ciclo o camino cerrado**

Un **camino** es **simple** si no se repiten aristas.

Un **camino** es **elemental** si no se repiten vértices, todos los vértices del camino, excepto quizás el primero y el último, son diferentes. Todo camino elemental es simple.

Un **ciclo** es un camino simple que comienza y acaba en el mismo vértice.

Un grafo G = (V, E) es **conexo** si para todo par de vértices u, v ∈ V existe un camino en el grafo G que comienza en u y acaba en v, sino es un grafo **disconexo**.

Decimos que un grafo dirigido es **fuertemente conexo** si cada par de vértices está conectado por al menos dos caminos disjuntos; es decir, es conexo y no existe un vértice tal que al sacarlo el grafo resultante sea disconexo y decimos que es **débilmente conexo** si el grafo resultante de convertir los arcos en aristas es conexo.

Un tipo especial de grafo conexo es el **árbol** que es un grafo no dirigido, conexo y acíclico. En cambio, si el grafo es desconexo se trata de un **bosque**, que es un grafo no dirigido, disconexo y acíclico. Propiedades del árbol:

- Un árbol con *n* vértices contiene exactamente n-1 aristas.
- Si se añade una única arista a un árbol, el grafo resultante contiene un único ciclo.
- Si se elimina una única arista de un árbol, entonces el grafo resultante deja de ser conexo.

Un grafo G es **euleriano** si existe un camino cerrado, de longitud mayor que cero, simple pero no necesariamente elemental que incluye a todas las aristas de G.

***Lema 1***: Un grafo no dirigido y conexo es euleriano si y sólo si el grado de todo vértice es par.

***Lema 2*****:** Un grafo dirigido y fuertemente conexo es euleriano si y sólo si el grado de todo vértice es 0.

Un grafo G es **hamiltoniano** si existe un camino cerrado y elemental que contiene todos los vértices de G. Si existe, el camino se llama **circuito hamiltoniano**.

En un grafo no dirigido el **grado de un vértice** es el número de vértices adyacentes a él y el **grado del grafo** es el máximo de los grados de sus vértices.

En un grafo dirigido el **grado de entrada de un vértice** es el número de sus vértices incidentes y el **grado de salida de un vértice** es el número de sus vértices adyacentes.

Un grafo es **completo** si existe el número máximo de aristas en el grafo, es decir, tsi existe arista entre todo par de vértices de V. El número de aristas (|E|) de un grafo completo es n (n-1) /2. Si se trata de un grafo dirigido |E| = n(n-1).

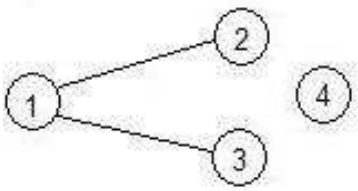
***Lema de las encajadas:*** La suma de todos los grados de los vértices de un grafo G = (V, E) es igual al doble de su medida:

$$\sum_{v \in V} g(v) = 2|E|$$

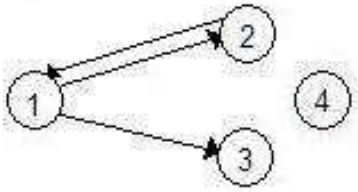
## Representación de grafos

Existen varias estructuras de datos que pueden utilizarse para representar los grafos dirigidos y no dirigidos. La elección de la estructura de datos adecuada depende del tipo de operaciones que se quieran aplicar al conjunto de vértices y aristas del grafo en cuestión. Las representaciones más comunes son las matrices de adyacencia y las listas de adyacencia.

Dado los dos grafos siguientes (dirigido y no dirigido) lo representaremos visualmente con una matriz de adyacencia y una lista de adyacencia:



grafo no dirigido



grafo dirigido

Si el número de aristas del grafo es elevado, las matrices de adyacencia tienen buenos costes espacial y temporal para las operaciones habituales.

Es conveniente usar listas de adyacencia cuando el grafo es poco denso y además el problema a resolver tiene que recorrerlas todas.

### Matriz de adyacencia

Se trata de una matriz A de tamaño VxV de booleanos, donde A[i][j] indicará si la arista (arco) que une al vértice *i* con el vértice *j* está en E. La matriz de adyacencias de un grafo no dirigido es una matriz simétrica que podemos ahorrar espacio (la mitad) guardando solo su parte inferior, en cambio, para un grafo dirigido necesitaremos toda la matriz.

**Ventaja**: es útil para acceder con frecuencia a si una determinada arista está presente en el grafo.

**Desventaja:** Siendo |V| = n, requiere un espacio de $\Omega(n^2)$ y por lo tanto, tan solo para leer la matriz requerirá un coste de $O(V^2)$, y para ver las adyacencias de un vértice tiene coste O(V). La alternativa para evitar estás desventajas es usar una lista de adyacencia. Un grafo no dirigido sólo necesita la mitad del espacio $(n^2-n)/2$.

| | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 1 | N | N | N | N |
| 2 | C | N | N | N |
| 3 | C | F | N | N |
| 4 | F | F | F | N |

grafo no dirigido

| | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 1 | N | C | C | F |
| 2 | C | N | F | F |
| 3 | F | F | N | F |
| 4 | F | F | F | N |

grafo dirigido

La definición usual en c++ de un grafo implementado con una matriz de adyacencia es:

```
typedef vector<vector<bool> > graf;
```

Ahora cuando declaremos una variable del tipo graf estaremos implementando el grafo con una matriz de adyacencia.
Ejemplo: `graf g(n,n);` // declaramos un grafo g de n vértices

Coste temporal de las operaciones básicas de un grafo no dirigido:

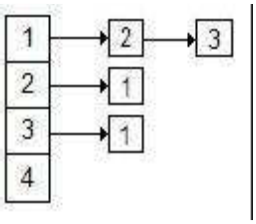
- **Crear** : se ha de inicializar la matriz de booleanos a falso. θ**($n^2$)**
- **Adyacentes:** para saber los vértices adyacentes a un vértice se ha de recorrer una fila de la matriz. θ**(n)**
- **Borrar vértice:** para borrar un vértice se han de borrar todas las adyacencias a ese vértice. θ**(n).**
- **Añadir arista:** para añadir una arista {u,v} se ha de acceder a la posición m[i][j] y ponerlo a true si estaba a false. θ**(1)**
- **Existe vértice:** Hemos de comprobar si el vértice se encuentra en el rango del tamaño de la matriz. θ**(1)**
- **Existe arista:** para saber si existe una arista {u,v} se ha de acceder a la posición m[i][j] y comprobar si está a true. θ**(1)**
- **Borrar arista:** para borrar una arista {u,v} se ha de acceder a la posición m[i][j] y ponerlo a false. θ**(1)**

## Lista de adyacencia

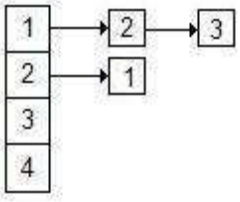
Se trata de un vector de tamaño V, donde cada posición del vector apuntará a una lista de los vértices adyacentes al vértice con dicha posición. La lista estará en orden creciente según el identificador de cada vértice.

**Ventaja**: la cantidad de memoria que requiere depende de V y de E, es decir, tiene coste de memoria θ(V+E) en el caso general. En el caso peor se trata de un grafo completo que tiene coste de memoria θ($V^2$).

**Desventaja**: determinar si una arista está o no en el grafo puede tomar un coste temporal de O(V) si la lista contiene los n-1 vértices, es decir, que el vértice es adyacente a todos los vértices.




grafo no dirigido




grafo dirigido

La definición usual en c++ de un grafo implementado con un vector de listas de adyacencias es:

```
typedef vector<list<int> > graf;
```

Ahora cuando declaremos una variable del tipo graf estaremos implementando el grafo con una vector de listas de adyacencias.

Ejemplo: `graf g(n);` // declaramos un grafo g de n vértices

Coste temporal de las operaciones básicas de un grafo no dirigido:

- **Crear** : se ha de inicializar el vector con listas vacías. θ**(n)**

- **Adyacentes:** para saber los vértices adyacentes a un vértice se ha de recorrer la lista del vertice. θ**(n)**

- **Borrar vértice:** para borrar un vértice se han de borrar todas las adyacencias a ese vértice. θ**(n).**

- **Añadir arista:** para añadir una arista {u,v} se ha de comprobar que la arista que se añade no exista previamente y luego añadirla si es el caso. Esto implica recorrer toda la lista asociada al vértice origen para detectar si ya existe. En el caso pero requerirá recorrer la lista completa, que puede tener un tamaño máximo de n-1 elementos, para efectuar posteriormente la inserción. Así obtenemos un coste θ**(n),** podemos reducir el coste de la operación si implementamos la lista de aristas en un AVL θ**(logn)**

- **Existe arista, Borrar arista:** Razonamiento similar al de añadir.

- **Añadir vértice:** Hemos de añadir el vértice al vector. θ**(1)**

- **Existe vértice:** Hemos de comprobar si el vértice se encuentra en el vector. θ**(1)**

// se accede a todos los vértices del grafo G
```
for(int u=0; u<G.size(); u++);
```

// se accede a todos las adyacencias del vértice u del grafo G
```
for(int v = 0; v < G[u].size(); ++v);
```

## Recorrido en profundidad

Un **recorrido en profundidad**, también denominado **DFS (depth-first search)**, sirve para recorrer todos los vértices del grafo. De aquí se pueden sacar diversas aplicaciones, para cuando tengamos que visitar todos los vértices del grafo. En cada paso se escoge seguir por una de las aristas que salen del vértice visitado más recientemente.

La implementación en c++ de un recorrido en profundidad de forma recursiva es la siguiente, donde retorna una lista de los vértices según su orden de visita en DFS:

```
void dfs_rec (const graph& G, int u, vector<boolean>& vis, list<int>& L){
   if (not vis[u]) {
      vis[u] = true;
      L.push_back(u);
      for (int i = 0; i < G[u].size(); ++i)
         dfs_rec(G,G[u][i],vis,L);
   }
}
```

// esta llamada inicial sirve por si se trata de un grafo no conexo

```
list<int> dfs_rec (const graph& G) {
   int n = G.size();
   list<int> L;
   vector<bool> visitado(n, false);
   for(int u = 0; u < n; ++u)
      dfs_rec(G,u,visitado,L);
   return L;
}
```

El **coste del DFS** es $\theta$**(|V|+|E|) =** $\theta$**(max{V,E})**: se recorre todos los vértices del grafo una única vez (V) y para cada vértice se recorre todas sus adyacencias, por lo tanto, se recorre todas las aristas del grafo una vez (E).
La implementación en c++ de un recorrido en profundidad de forma iterativa es la siguiente:

```
list<int> dfs_ite (const graph& G) {
   int n = G.size();
   list<int> L;
   stack<int> S;
   vector<bool> vis(n,false);
   for(int u = 0; u < n; ++u) {
      S.push(u);
      while (not S.empty()) {
         int v = S.top(); S.pop();
         if (not vis[v]) {
            vis[v] = true; L.push_back(v);
            for (int i = 0; i < G[v].size(); ++i)
               S.push(G[v][i]);
         }
      }
   }
   return L;
}
```

## Recorrido en anchura

Un **recorrido en anchura**, también denominado **BFS (breadth-first search)**, sirve para recorrerse todos los vértices del grafo a partir del vértice inicial v, pero de tal manera que recorremos los vértices más próximos a v antes que los más lejanos, por lo tanto, el BFS se utiliza para saber distancias de caminos mínimos de un vértice u al resto de vértices.

Una vez visto el recorrido en profundidad es sencillo obtener el algoritmo que efectúa el recorrido en anchura de un grafo. Es suficiente con obtener una versión iterativa del DFS y sustituir la pila, que mantiene los vértices que aún tienen adyacentes por visitar, por una cola. En la cola se almacenan los vértices adyacentes al último visitado y que aún están sin visitar. A lo sumo habrá vértices pertenecientes a dos niveles consecutivos del grafo, los que se encuentran a distancia mínima *k* y a distancia mínima *k+1* del vértice a partir del cual se ha iniciado el recorrido. También es posible que existan vértices repetidos en la cola.

La implementación en c++ de un recorrido en amplitud de forma iterativa para un grafo conexo es la siguiente, donde retorna un vector de las distancias desde el vértice inicial v al resto de vértices:

```
list<int> bfs (const graph& G, int v) {
   int n = G.size();
   vector<bool> visitado(n, false);
   list<int> L;

   visitado[v]=true;
   queue<int> Q;
   Q.push(v);
   while (not Q.empty()) {
      int w = Q.front();
      Q.pop();
      L.push_back(w);
      for (int i = 0; i < G[w].size(); ++i) {
         int x = G[w][i];
         if (not visitado[x] {
            Q.push(x);
            visitado[x] = true;
         }
      }
   }
   return L;
}
```

Si el grafo fuera disconexo habría que lanzar una llamada inicial parecida a la llamada inicial del DFS.

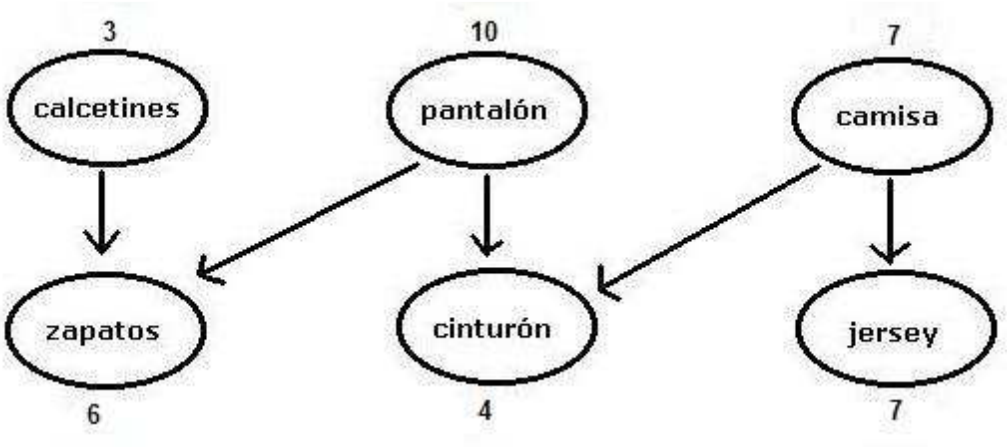
El **coste del BFS** es $\theta$**(V+E) =** $\theta$**(max{V,E})**: cada vértice del grafo se pone y se saca de la cola una única vez (V), por otro lado la lista de adyacencias de cada vértice se recorre entera cuando se saca el vértice de la cola, y esto sucede para todos los vértices así que se recorre todas las aristas del grafo (E).

## Ordenación topológica

Una **ordenación topológica** de un grafo acíclico *G* dirigido es una ordenación lineal de todos los nodos de *G* que conserva la unión entre vértices del grafo *G* original. La condición que el grafo no contenga ciclos es importante, ya que no se puede obtener ordenación topológica de grafos que contengan ciclos.

Usualmente, para clarificar el concepto se suelen identificar los nodos con tareas a realizar en la que hay una precedencia a la hora de ejecutar dichas tareas. La ordenación topológica por tanto es una lista en orden lineal en que deben realizarse las tareas.

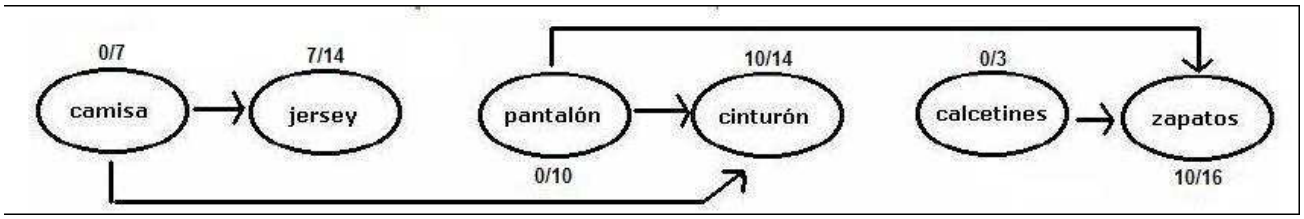
En el siguiente grafo ejecutaremos el algoritmo de ordenación topológica, donde el número que tiene cada vértice indica la duración de esa tarea




grafo acíclico y dirigido sobre el orden de las tareas al vestirse

La solución es la siguiente, donde el primer número de cada vértice indica el instante de tiempo para comenzar la tarea y el segundo número indica el instante de tiempo para acabar la tarea.

La duración de la tarea = tiempo de finalización – tiempo de comienzo.




solución usando ordenación topológica

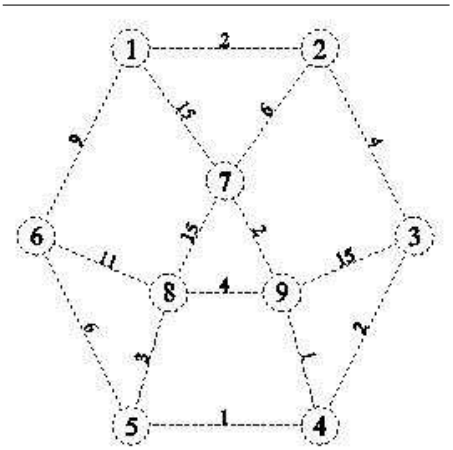
Partiendo de las versiones iterativa y recursiva del DFS, pueden determinarse dos algoritmos (iterativo y recursivo, respectivamente) eficientes para obtener una posible ordenación topológica de un grafo dirigido y acíclico dado, haciendo unas modificaciones.

```
list<int> ordenacio_topologica (graf& G) {
   int n = G.size();
   vector<int> ge(n, 0);
   for(int u = 0; u < n; ++u) {
      for (int i = 0; i < G[u].size(); ++i) {
         ++ge[G[u][i]];
      }
   }

   stack<int> S;
   for(int u = 0; u < n; ++u) {
      if(ge[u] == 0) S.push(u);
   }

   list<int> L;
   while (not S.empty()) {
      int u = S.top();
      S.pop();
      L.push_back(u);
      for (int i = 0; i < G[u].size(); ++i) {
         int v = G[u][i];
         if(--ge[v] == 0) S.push(v);
      }
   }
   return L;
}
```

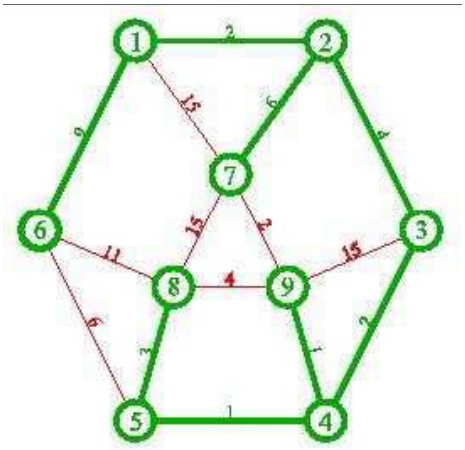
## Algoritmo de Dijkstra

El **Algoritmo de Dijkstra** dado un grafo G = <V,E> con pesos en las aristas te devuelve el camino mínima de un vértice inicial al resto.

Ejemplo:



Grafo inicial



Muestra el camino mínimo del vértice 1 al resto

### Definición en c++ de un grafo con pesos en las aristas

La definición usual en c++ de un grafo implementado con un vector de listas de adyacencias. Para el algoritmo de Dijkstra utilizaremos un grafo que tiene pesos en las aristas, por lo tanto, haremos una nueva definición de un grafo con pesos en las aristas con un vector de listas de adyacencias.

Una arista se define en c++ como: `Typedef pair<double, int> ArcP;`

Definiremos en c++ un grafo con pesos: `typedef vector<vector<ArcP> > wgraf;`

Tendremos las siguientes macros para mayor facilidad al programar:

```
void dijkstra (const wgraph& G, int u, vector<double>& d, vector<int>& p)
{
   int n = G.size();
   d = vector<double>(n, infinit); d[u] = 0;
   p = vector<int>(n, -1);
   vector<boolean> visto(n,false);
   priority_queue<ArcP, vector<ArcP>, greater<ArcP> > CP;

   CP.push(ArcP(0, u));
   // en la cola de prioridad se guardan pares(p, v)
   // v = vértice que añadimos a la cola.
   // p = distancia[v] el peso del camino mínimo que llega a v, es la
   //      prioridad del elemento en la cola de prioridad

   while (not CP.empty()) {
      ArcP a = CP.top(); CP.pop();
      int v = a.second;
      if (not visto[v]) {
         visto[v] = true;
         for (int i = 0; i < G[v].size(); ++i) {
            int w = G[v][i].second;
            double q = G[v][i].first;
            if (not visto[w]) {
               if (d[w] > d [v] + q) {
                  d [w] = d [v] + q;
                  p[w] = v;
                  CP.push(ArcP(distancia[w], w));
               }
            }
         }
      }
   }
}
```

NOTA: el **recorrido en amplitud BFS** nos devuelve el camino mínimo des de un vértice inicial al resto en un grafo SIN PESOS en las aristas. **Dijkstra** nos devuelve el camino mínimo des de un vértice inicial al resto en un grafo CON PESOS en las aristas.

# Búsqueda Exhaustiva

## Funcionamiento

Los **algoritmos de búsqueda exhaustiva** es una técnica general de exploración del espacio de soluciones, es decir, combina TODAS las soluciones posibles.

Se **caracteriza** por:
- los caminos tomados se pueden deshacer
- no utiliza ningún criterio para elegir el camino
- si el problema tiene solución la encuentra
- se trata generalmente de problemas de optimización, con o sin restricciones.
- La solución es expresable en forma de secuencia de decisiones.
- existe una función *factible*, que permite averiguar si una secuencia de decisiones, viola o no las restricciones.
- existe una función *solución,* que permite determinar si una secuencia de decisiones factible es solución al problema.

**Backtracking** es una búsqueda ciega: fijado un nodo del espacio, el siguiente nodo a visitar es el primer hijo sin visitar, en un recorrido en profundidad y el siguiente hermano, en un recorrido en anchura.
Sirve para:
- buscar una solución
- buscar todas las soluciones
- buscar la solución óptima

Las soluciones deben poder expresarse como n-tuplas $(x_1, x_2,..., x_n)$, $x_i \in S_i$. Siendo $S_i$ finito.
El Backtracking permite desestimar soluciones si sabemos que por ahí no llegamos a ninguna solución, con esto, se puede reducir sustancialmente el coste de la búsqueda.
Se recorre en **profundidad** el árbol de estados, es decir, se recorre en **Preorden.**
El esquema general del **backtracking recursivo** es el siguiente:

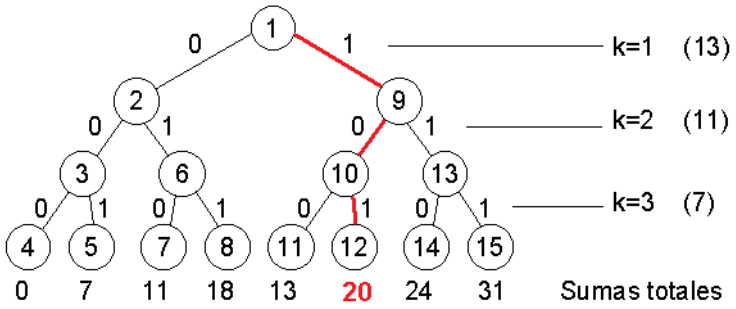
```
backtracking (d:datos problema; s:solución) {
   if(esSolucionFinal(s)) tratarSolucion;
   else {
      while(hayaDecisionesPorTomar) {
         decisión d = cogerDecision();
         if(esFactible(d)) {
            marcar(d);
            backtracking(d,s^{d});
            desmarcar(d);
         }
      }
   }
}
```

En cada iteración del backtracking primero comprobamos si la solución en curso es una solución final, en caso de que sea la tratamos. Si no es, tomamos todas las posibles decisiones que sean factibles (cumplan restricciones), y llamamos recursivamente con esta decisión incluida en la solución parcial. Así sucesivamente.

**Ejemplo:** Dado un conjunto de números enteros {13, 11, 7}, encontrar si existe algún subconjunto cuya suma sea exactamente 20. En cada nivel decidimos si el elemento i-ésimo está o no en la solución, representación de la solución: ($x_1$, $x_2$, $x_3$ ) donde

- $x_i = 0$ si $v_i$ no aparece en la solución
- $x_i = 1$ sino

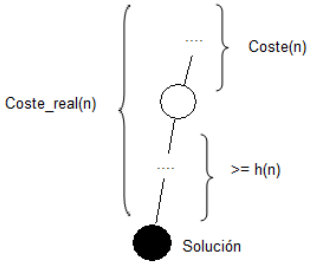
Árbol de estados




Cada nodo representa un paso del algoritmo, una solución parcial en cada momento. El árbol indica un orden de ejecución (recorrido en profundidad), pero no se almacena en ningún lugar. Una solución en un nodo hoja con valor de suma 20.

Podemos encontrar una solución mejor, si diseñamos funciones de cota, de forma que no se generen algunos estados si no van a conducir a ninguna solución. De esta forma se ahorra espacio en memoria y tiempo de ejecución. Para esto se usan **heurísticas**, este proceso poda el árbol de búsqueda antes de que se tome la decisión y se llame a la subrutina recursiva.

Coste_estimado(n) = coste(n) + h(n)




**Poda** basada en la mejor solución en curso:

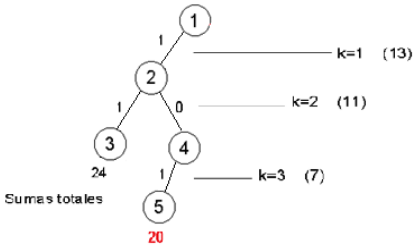
Si coste_estimado(n) > coste_mejor entonces podamos el nodo n.
En problemas de maximización la estimación ha de ser una cota superior al beneficio

Si beneficio_estimado(n) < beneficio_mejor entonces podamos el nodo n.
Volviendo al problema, para poder podar, miraremos si las sumas parciales no superan a la suma que queremos encontrar, de ser así, continuamos, en caso de superarla, podamos:

- **(restricción implícita)** : $\sum_{i=1}^{k} v_i \cdot x_i \leq 20$

- **(restricción explícita)** : $x_i \in \{0,1\}$

- **(completable)** : $\sum_{i=1}^{k} v_i \cdot x_i + \sum_{i=k+1}^{n} v_i \geq 20$



# Bloc 5: Búsqueda Exhaustiva

**Miriam**
**Telf: 93.204.62.56**

## El problema de las n-reinas

En el problema de las n-reinas nos dan un tablero de tamaño nxn donde tenemos que colocar n reinas de forma que ninguna de ellas se mate. La solución es de la forma ($x_1$, $x_2$, ...., $x_n$) donde $x_i$ = columna para la reina situada en la fila i-esima Como restricciones al problema tenemos que **(la función de factibilidad debe comprobarlas)**:

- **(restricción implícita)** ninguna reina puede estar fuera del tablero, las reinas no se pueden matar entre ellas
- **(restricción explícita)** $1 \leq x_i \leq 8$

El algoritmo de las 8-reinas es el siguiente (este algoritmo te encuentra todas las soluciones):

- Usamos como solución un vector de enteros de tamaño n, para guardar en cada posición "i", la columna donde hemos puesto la reina i-ésima.
- Para saber si la reina está colocada en una columna correcta tendremos 3 vectores de bool, uno para marcar las columnas que estamos usando (de tamaño n), otro para marcar la diagonal 1 ↖ y otro para marcar la diagonal 2 ↗ (los dos de tamaño 2*n-1).

```
void nReinas (int fila, vector<int>& sol, int n, vector<bool>& c,
vector<bool>& d1, vector<bool>& d2) {
//Pre: los vector c, d1, d2 estan inicializados con todas las posiciones
a falso y fila = 0
   if(fila == n)  escribirSolución(sol);
   else{
      for (int col = 0; col < n; ++col) {

         int diag1 = n - col - 1 + fila;

         int diag2 = col + fila;

//si la columna col, la diagonal diag1, y la diagonal diag2 no están
//usadas, la columna col es una columna válida para poner la reina
         if(not c[col] and not d1[diag1] and not d2[diag2]) {
            //marcaje: colocamos la reina
            sol[fila] = columna;

            c[col] = true;
            d1[diag1] = true;
            d2[diag2] = true;

            nReinas(fila+1, sol, n, c, d1, d2);

            //desmarcaje: descolocamos la reina
            c[col] = false;
            d1[diag1] = false;
            d2[diag2] = false;
         }
      }
   }
}
```

Si hemos colocado las 8 reinas (hemos colocado una reina en cada fila, es decir, fila = 8) tratamos la solución y continuamos buscando soluciones al problema. Si no hemos colocado las 8 reinas intentaremos colocar una reina en cada columna de la fila que estamos tratando, y en caso de que no se maten con alguna otra reina llamaremos recursivamente indicando que hemos colocado una reina en dicha fila (marcaje) y pasamos a la siguiente. A la vuelta de la recursividad descolocamos la reina que hemos colocado (desmarcaje), y continuaremos probando con las demás columnas. Así sucesivamente.

Esta solución nos da todas las posibles soluciones, si queremos dar una solución deberemos añadir un bool a la cabecera, ponerlo a true cuando lleguemos al caso directo (fila == n) y en el for poner la condición de not trobat.

```
void nReinas (int fila, vector<int>& sol, int n, vector<bool>& c,
vector<bool>& d1, vector<bool>& d2, bool& trobat) {
//Pre: los vector c, d1, d2 estan inicializados con todas las posiciones
a falso y fila = 0, not trobat

   if(fila == n) {
      escribirSolución(sol);
      trobat = true;
   }
   else{
      for (int col = 0; col < n and  not trobat; ++col) {
         …
         …
      }
   }
}
```

## El problema de la mochila

En el problema de la mochila nos dan como parámetros de entrada la capacidad C que puede soportar una mochila y también nos dan el peso Pi y valor Vi de n objetos. Se trata de **maximizar** el valor que contiene la mochila.

Como restricciones al problema tenemos que **(la función de factibilidad debe comprobarlas)**:

- **(restricción implícita)** introducimos k objetos en la mochila donde $1 \leq k \leq n$
- **(restricción explícita)**sumatorio de los pesos de los objetos introducidos en la mochila ≤ C

Del problema de la mochila se sacan dos variantes:

- **Mochila fraccionaria**: podemos introducir una parte de un objeto
- **Mochila entera**: introducimos un objeto o no, no hay más opciones

Para las dos variantes podemos tomar los siguientes **criterios (función de selección)** para elegir el siguiente objeto a introducir en la mochila de los candidatos restantes:

- por **orden creciente de peso**: confiamos en llenar la mochila lo más tarde posible
- por **orden decreciente de valor**: introducimos primero los objetos de más valor
- por **orden decreciente de relación valor/peso**: introducimos primero los objetos que tienen un mayor valor por unidad de peso

El mejor criterio para llegar a maximizar el valor es el tercero.

```
vector<bool> mochila (vector<double>& P, vector<double>& V, double C, int
n, vector<bool>& solucion, double& millor, double val, double pes, int i)
{
//i es el objeto que toca tratar, val es el valor acumulado y pes es el
peso acumulado, C es la capacidad de la mochila

   vector<bool> millorv;
   if (i == n) {
      if (val > millor) {
         millorv = solucion;
         millor = val;
      }
   }
   else {
      //miramos si podemos coger el objeto
      if (pes + P[i] <= C) {
      solucion[i] = true;
      mochila(P, V, C, n, solucion, millor, val+V[i], pes+P[i], i+1);
      }
      //no cogemos el objeto
      solucion[i] = false;
      mochila(P, V, C, n, solucion, millor, val, pes, i+1);
   }
   return millorv;
}
```

# Clases P y NP

## Funcionamiento

**Dominio de datos:** es un conjunto que asigna un nº natural a cada elemento del conjunto. Si *x* es un objeto del dominio de datos, el tamaño de *x* se escribe |x|.

**Problema computacional**: encontrar una solución al problema si tiene.
Viene dado por:

- Un conjunto E de entradas posibles
- Un conjunto S de salidas posibles
- Una relación R $\subseteq$ E x S, indica los pares (x, y) formados por una entrada x $\in$ E y una salida y $\in$ S que son soluciones válidas.

Dada una entrada x $\in$ E, encontrar, si hay, una salida y $\in$ S tal que (x, y) $\in$ R.

**Problema decisional**: ¿existe alguna solución al problema? Sí o no.
Viene dado por:

- Un conjunto E de entradas posibles
- Un subconjunto L $\subseteq$ E de instancias positivas (la propiedad que ha de satisfacer la entrada).

Dada una entrada x $\in$ E, determinar si x $\in$ L

**P:** clase de los problemas decisionales para los cuales existe un **algoritmo determinista** que los resuelve en tiempo polinómico.(voraces).

"Dado x $\in$ E, deteminar si x $\in$ P"

Decimos que un problema decisional es decidible en tiempo polinómico si existe un algoritmo polinómico A : E $\rightarrow$ {0,1} tal que $\forall$x $\in$ E : si x $\in$ P $\Leftrightarrow$ A(x) = 1

**NP:** clase de los problemas decisionales para los cuales existe un **algoritmo no determinista** que lo resuelve en tiempo polinómico (backtracking)

"Dado x $\in$ E, deteminar si x $\in$ P"

Decimos que un problema decisional es decidible en tiempo polinómico indeterminista si existe un algoritmo polinómico A, un dominio de datos E' A: E x E' $\rightarrow$ {0,1} y un polinomio p(n), tales que $\forall$x $\in$ E : si x $\in$ P $\Leftrightarrow$ A(x, y) = 1, para algún y $\in$ E' tal que |y| = p(|x|)

Llamamos a y en el caso de que x $\in$ P **Certificado o testimonio o prueba** ( es una solución a un problema).

Llamamos al algoritmo A: E x E' $\rightarrow$ {0,1} **verificador.**

Un problema de decisión $\in$ NP si puede ser resuelto con un algoritmo indeterminista en tiempo polinómico.

A $\in$ NP $\Leftrightarrow$ L = {x $\in$ E| $\exists$y $\in$ S: |y| $\leq$ P(|x|), (x, y) $\in$ R, R $\in$ P}

# Bloc 6: Clases P y NP

**Miriam**

**Telf: 93.204.62.56**

**Costes y algoritmos polinómicos:** Dado un algoritmo A que coge entradas de E y retorna salidas de S, para cada entrada $x \in E$, sea $T_A(x)$ el número de pasos que tarda el algoritmo A con entrada x. Para cada $n \in N$:

$t_A(n) = \max\{ T_A(x): x \in E, |x| = n\}$.

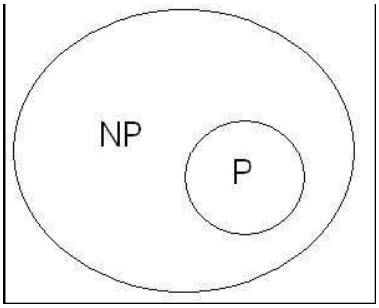
Decimos que A es un algoritmo polinómico, si existe un polinomio p(n) tal que $t_A(n) \leq p(n)$ para todo $n \in N$.

## Diferencias entre P y NP

| | **P** | **NP** |
|---|---|---|
| ¿existe alguna solución al problema? | Tiempo polinómico | Tiempo exponencial ** |
| ¿existe alguna solución al problema teniendo un testimonio? (coste de verificar el testimonio) | Tiempo polinómico | Tiempo polinómico |

**En realidad no está demostrado que un problema NP no se pueda resolver en tiempo polinómico. Si alguien demostrase que los problemas NP se pueden resolver en tiempo polinómico en realidad P y NP serían el mismo conjunto.

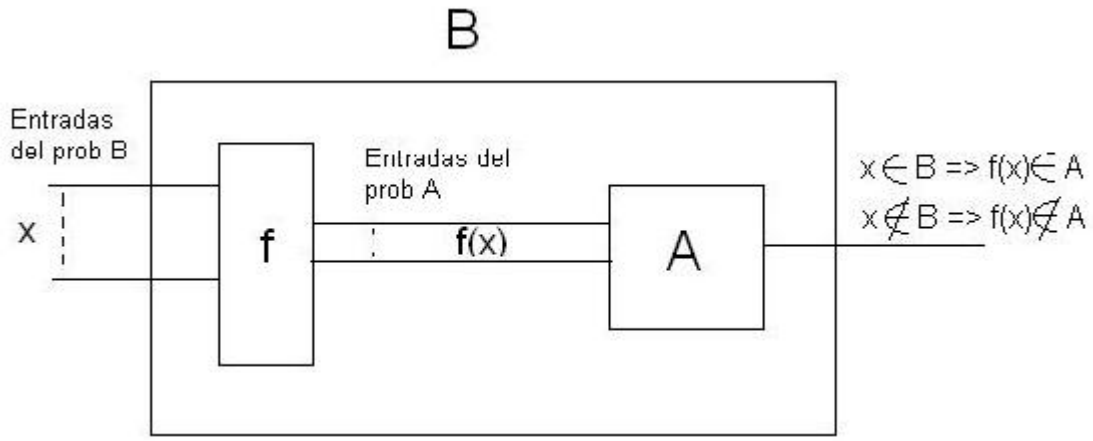
En la actualidad se considera que P≠NP.




## Reducciones

Una reducción de un problema B a un problema A se trata de resolver B a partir de A.

Para ello debe existir una función f de coste polinómico que transforme las entradas del problema B en entradas para el problemas A de tal manera que el problema A y B se comporten igual
=> B se reduce a A ($B \leq A$)




Si A € P /\ B ≤ A => B € P
Si A € NP /\ B ≤ A => B € NP

## Definición de NP-hard (más difíciles que NP)

L € NP-hard si TODOS los problemas de NP se pueden reducir a L.

## Definición de NP-completo (los más difíciles de NP)

L € NP-completo si L € NP-hard y L € NP.

El primer problema que se demostró que pertenecía a NP-completo fue el problema de la satisfactibilidad (SAT): dada una fórmula booleana decir si existe una asignación de las variables tal que la fórmula de cierto.

- Encontrar esta asignación tiene coste exponencial, así que € NP
- Según el **teorema de Cook** podemos expresar CUALQUIER problema (de NP) en una fórmula booleana, es decir, todos los problemas NP se pueden reducir a SAT.

**Teorema de Cook:** El Problema de satisfacibilidad booleana (SAT) es NP-completo.

SAT: Dado un circuito C con puertas lógicas AND, OR, NOT, variables $x_1$, $x_2$, ..., $x_n$ y una única salida, determinar si C es satisfactible.

A partir de aquí actualmente se van conociendo más problemas de NP que resultan que están

Si A € NP /\ SAT ≤ A => A € NP-completo
Si B € NP /\ A ≤ B => B € NP-completo

**www.asesacademia.com**
**González Tablas, 7**
Telèfon 93.204.62.56
**Acadèmia Ases**