# Algorítmia

Abel Doñate Muñoz
abel.donate@estudiantat.upc.edu

## Contents

# 1 Costes computacionales

## 1.1 Notación

Consideramos la función de coste $c(x)$, que nos devolverá el coste del algoritmo en función del input $|x| = n$. Como el coste no siempre es constante (no solo depende de $n$, sino también del input en sí) definimos:

$$\begin{cases} c_{peor}(n) & = \max_x c(x) : |x| = n \\ c_{mejor}(n) & = \min_x c(x) : |x| = n \\ c_{media}(n) & = \frac{1}{N} \sum_{x:|x|=n} c(x) \end{cases}$$

En cuestión de notación asintótica:

$$\begin{cases} f = \mathcal{O}(g) \iff \exists c : f < cg \\ f = \Omega(g) \iff \exists c : f > cg \\ f = \Theta(g) \iff f = \mathcal{O}(g) \text{ y } f = \Omega(g) \end{cases}$$

## 1.2 Teoremas maestros

**Teorema de las recurrencias substractivas**

$$T(n) = aT(n-c) + g(n),\ g(n) = \Theta(n^k) \implies T(n) = \begin{cases} \Theta(n^k) & \text{si } a < 1 \\ \Theta(n^{k+1}) & \text{si } a = 1 \\ \Theta(a^{\frac{n}{c}}) & \text{si } a > 1 \end{cases}$$

**Teorema de las recurrencias divisorias**

$$T(n) = aT\left(\frac{n}{b}\right) + g(n),\ g(n) = \Theta(n^k),\ \alpha = \log_b a \implies T(n) = \begin{cases} \Theta(n^k) & \text{si } \alpha < k \\ \Theta(n^k \log n) & \text{si } \alpha = k \\ \Theta(n^\alpha) & \text{si } \alpha > k \end{cases}$$

## 1.3 Divide y vencerás

La idea de divide y vencerás es computar subproblemas por separado y luego mezclarlos. Iterando el proceso podemos conseguir algoritmos más eficientes.

**Algoritmo:** *Mergesort* $\mathcal{O}(n \log n)$

Este es el ejemplo por excelencia. Para ordenar un vector podemos dividirlo en dos y ordenar cada una de sus partes. Luego juntamos los elementos. Repetimos este algoritmo recursivamente con cada parte.

**Algoritmo:** *Exponenciación rápida* $\mathcal{O}(\log n)$

Para calcular $a^n$ podemos calcular $a^{\frac{n}{2}}$ recursivamente y multiplicarlo. Iterando llegamos a este algoritmo de coste $\log n$

**Algoritmo:** *Karatsuba-Offmann para producto de dos números* $\mathcal{O}(n^{\log_2 3})$

Podemos multiplicar dos números $x, y$ (binarios) de la siguiente forma:
Sea $x = a \times 2^{\frac{n}{2}} + b, y = c \times 2^{\frac{b}{2}} + d \implies xy = [(a+b)(c+d) - ac - bd] \times 2^{\frac{n}{2}} + ac \times 2^n + bd$, que solo son 3 operaciones.

**Algoritmo:** *Algoritmo de Strassen para multiplicación de matrices cuadradas* $\mathcal{O}(n^{\log_2 7})$

Para multiplicar $AB$ dividimos las matrices de la manera indicada, y solo tenemos que realizar 7 productos.

$$AB = \begin{pmatrix} A_1 & A_2 \\ A_3 & A_4 \end{pmatrix} \begin{pmatrix} B_1 & B_2 \\ B_3 & B_4 \end{pmatrix}$$

**Algoritmo:** *Fibonacci* $\mathcal{O}(\log n)$

Podemos reescribir la recurrencia como:

$$\begin{pmatrix} F_{k+1} \\ F_k \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^k \begin{pmatrix} 1 \\ 0 \end{pmatrix}$$

Y resolver la exponenciación de matrices con una exponenciación rápida.

# 2 Algoritmos de ordenación

## 2.1 Sin cotas

| | Coste Peor | Coste Mejor | Coste Medio | In Situ | Estable |
|---|---|---|---|---|---|
| **Selection** | $\theta(n^2)$ | $\theta(n^2)$ | $\theta(n^2)$ | SI | NO |
| **Insertion** | $\theta(n^2)$ | $\theta(n)$ | $\theta(n^2)$ | SI | SI |
| **Merge** | $\theta(n \log n)$ | $\theta(n \log n)$ | $\theta(n \log n)$ | NO | |
| **Quick** | $\theta(n^2)$ | $\theta(n \log n)$ | $\theta(n \log n)$ | SI | NO |
| **Bubble** | $\theta(n^2)$ | $\theta(n^2)$ | $\theta(n^2)$ | SI | SI |
| **Heap** | $\theta(n \log n)$ | $\theta(n \log n)$ | $\theta(n \log n)$ | SI | NO |

## 2.2 Acotadas

Podemos ordenarlos en tiempo lineal si sabemos que los números serán enteros en $[0, \ldots, k]$

### 2.2.1 Counting-sort

- Guardamos los números en un vector C de longitud $k$ donde cada posición indica el número de elementos $\leq i$
- Empezamos por el final de B y si la diferencia con el anterior es $\geq 1 \implies$ C[x]- -; B[x]++ . Si es 0, miramos el anterior.

### 2.2.2 Radix-sort

- En cada iteración ordenamos por el $i$-ésimo dígito con counting sort.
- Como counting sort es estable, terminamos con el vector ordenado.

### 2.2.3 Bucket-sort

Se realiza sobre una distribución uniforme del intervalo $[0, 1)$

- Dividimos el intervalo en subintervalos de la misma longitud llamados buckets.
- Repartimos los elementos
- Ordenamos cada bucket con un algoritmo de ordenación

# 3 Programación dinámica

Nos serviremos siempre de la misma estructura. Construiremos una función recursiva que llame a un vector si en la posición de este vector no hay un $-1$, y en caso contrario haga una relación de recursión.

```
using namespace std;
#include<iostream>
#include<vector>

using ll = long long;
using VL = vector<ll>;
using VVL = vector<VL>;

VL F = VL(101,-1);

ll f(int i){
  if(i==0) return 0;
  if(F[i]!=-1) return F[i];
  return (cosas con f(i-1), f(i-2), ...);
}
```

# 4 Estructuras de datos

## 4.1 Stacks y Queues

Tabla comparativa de estas dos

| | Funcionamiento | Funcines |
|---|---|---|
| **Stack** | Puedes acceder al último elemento guardado | .size(), .push(), .pop(), .top(), .empty() |
| **Queue** | Puedes acceder al primer elemento guardado. | .size(), .push(), .pop(), .front(), empty() |
| **P.Queue** | Puedes acceder al mayor elemento. | .size(), .push(), .pop(), .top(), empty() |

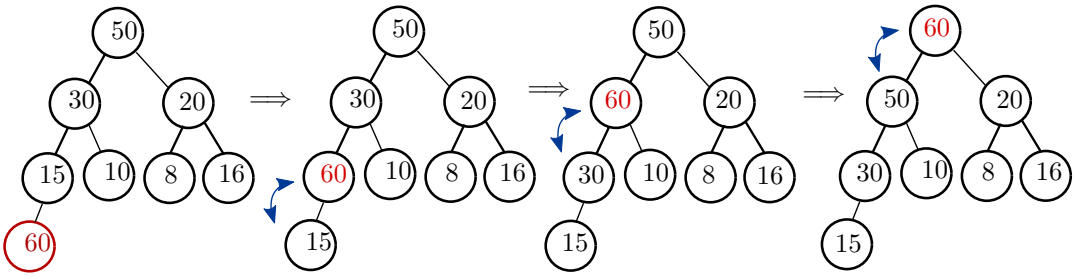
## 4.2 Maxheap

Una un maxheap es una estructura de datos similar a un vector con un orden parcial. Si lo visualizamos como un árbol binario se cumple que los hijos siempre son $\geq$ que los padres.

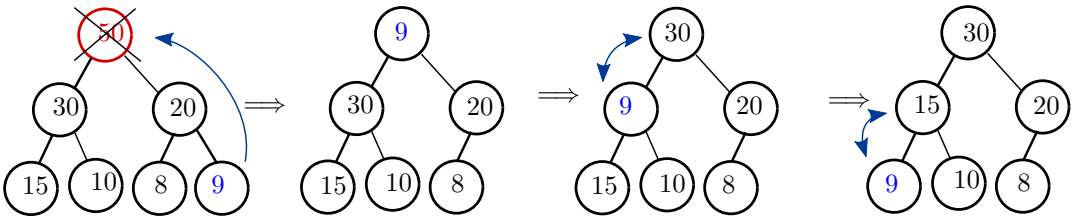
Las operaciones básicas son:

- **Inserción** $O(\log n)$**:** Insertamos el elemento en la primera hoja disponible y hacemos un Heapyfy
- **Borrado** $O(\log n)$**:** Cambiamos el primer elemento (la raíz) por el último y hacemos un Heapyfy.
- **Heap-Sort** $O(n \log n)$**:** En cada paso borra la raíz y la guarda en un vector. El resultado es un vector ordenado.
- **Heapify** $O(n)$**:** Desde un árbol binario creamos un Maxheap. El procedimiento es crear Maxheaps recursivamente de abajo hacia arriba

Vemos como hacer la inserción del número 60:

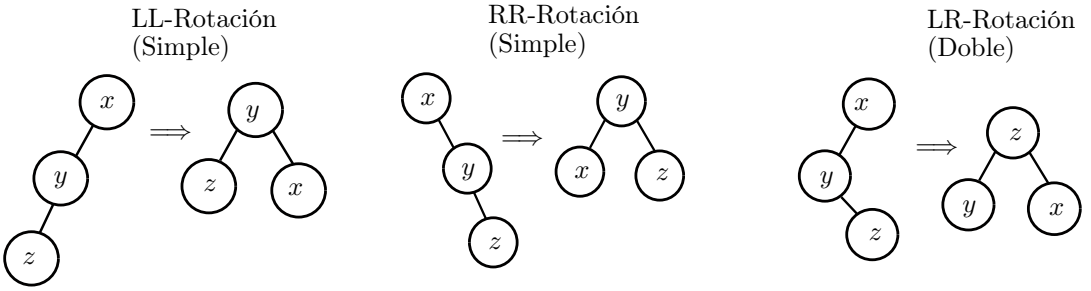



Vemos como hacer el borrado del máximo elemento:

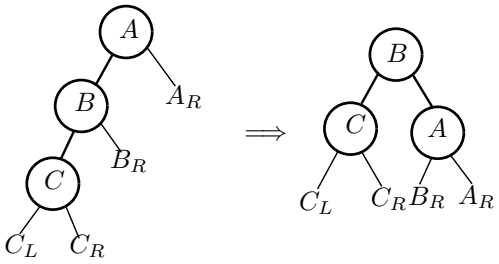



## 4.3 Árboles AVL

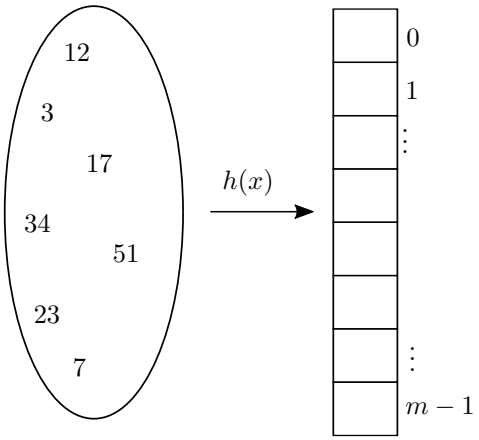
Un AVL es un árbol binario ordenado (los hijos de la derecha son $\geq$ al padre y los de la izquierda $\leq$ al padre) y balanceado (la diferencia de alturas entre cualesquiera dos hijos es $\leq 1$). Podemos balancear cualquier árbol binario para convertirlo en un AVL con las siguientes operaciones llamadas rotaciones. Distinguimos tres tipos:




Vemos ahora como aplicarlo a un árbol general cuando $A, B, C$ están desbalanceados y hay que hacer una rotación sobre ellos.



## 4.4 Tablas de Hash




Las tablas de hash sirven para almacenar claves del primer set en una tabla de menor longitud fijada $m$. Esta operación se realiza para que la búsqueda de elementos sea más eficiente, ya que haremos menos operaciones para encontrar un elemento.

La función de hash $h(x)$ mapea cada elemento del primer conjunto en un hueco del segundo (es decir, otorga a cada numero otro número del 0 al $m-1$). Si queremos buscar si un número está en el conjunto original, solo debemos calcular su hash y mirar si está en el hueco de la tabla correspondiente.

Una función de hash $h(x)$ se dice que es **universal** si la probabilidad de que una clave caiga en un hueco es la misma para todos los huecos. En la práctica buscaremos estas funciones, porque nos aseguran el menor número de colisiones en la tabla.

**Rolling Hash**
Se trata de un algoritmo para búsqueda de texto en un documento de texto. Suponemos que queremos buscar la cadena $abc$ dentro de una cadena de texto $s$. El algoritmo naive sería colocarse en el índice $i$ del texto y mirar si $s[i] = a$. Si lo es, mirar si $s[i+1] = b$, y si lo sigue siendo, si $s[i+2] = c$. Si se cumplen las tres, hay un match. Hacemos esto para $i = 0, \ldots, n-3$. Este algoritmo se realiza en $\mathcal{O}(nm)$.

Un algoritmo mucho más eficiente es el rolling hash:
Nos ayudamos de una función de hash $h$ que mapea cada cadena de texto a un natural. Hacemos $h(abc) = H$. Escogemos una función de hash de la forma

$$h(s[i+1, \ldots, i+m]) = s[i+1]p^k + s[i+2]p^{k-1} + \ldots + s[i+m]p^0$$

donde $p$ es primo. Una propiedad importante de este hash es que

$$h(s[i+1, \ldots, i+m]) = p(h(s[i, \ldots, i+m-1]) - s[i]p^{m-1}) + s[i+m]$$

Aprovechando esta propiedad, sabiendo el hash de $s[i,\ldots,i+m-1]$ podemos calcular el de $s[i+1,\ldots,i+m$ en $\mathcal{O}(1)$. Una vez que tenemos el hash lo comparamos con $H$, y si es igual y el hash está bien distribuido, no habrá muchas colisiones y el algoritmo es $\mathcal{O}(n)$ (observamos que no depende de la longitud $m$ de la cadena que queremos buscar).

Ejemplo:

Queremos buscar $abc$ en el texto $dabc$. Tomamos $p = 3$ y $s \begin{cases} a \to 0 \\ b \to 1 \\ c \to 2 \\ d \to 3 \end{cases}$

Calculamos $H = h(abc) = 0 \times 3^2 + 1 \times 3^1 + 2 \times 3^0 = 5$.

$$\begin{array}{c} \underline{dab}c \\ h(dab) = 3 \times 9 + 0 \times 3 + 1 = 28 \\ h(dab) \neq H \text{ no match} \end{array} \quad \Longrightarrow \quad \begin{array}{c} d\underline{abc} \\ h(abc) = 3 \times (h(dab) - 3 \times 9) + 2 = 5 \\ h(abc) = H \text{ match} \end{array}$$

# 5 Grafos

## 5.1 Algunas definiciones previas

- **DAG** Grafo dirigido acíclico

En cuanto a definiciones en el lenguaje de la estructura del grafo:

- VVI M; Matriz de adyacencia
- VVI G; Lista de aristas
- VP V; Pair con las aristas
- VVP GP; Conjunto de vértices en pairs con costes ( pair<coste, arista> )
- int n = M.size(); Nodos
- int m = G.isze(); Aristas
- int INF = 1e9;

## 5.2 DFS y BFS

DFS $\Longrightarrow$ recorre un camino en profundidad y va tirando hacia detrás.
BFS $\Longrightarrow$ recorre el grafo por orden de proximidad.

**Algoritmo:** *DFS* $\mathcal{O}(n+m)$

- Creamos un vector booleano visited que guarde los nodos visitados.
- Miramos los nodos adyacentes.
  - Si visited $\Longrightarrow$ return
  - Si not visited $\Longrightarrow$ aplicamos la función recursivamente sobre el adyacente.

**Algoritmo:** *BFS* $\mathcal{O}(n+m)$

- Creamos un vector booleano visited que guarde los nodos visitados.
- Creamos una queue q y añadimos el nodo origen.
- while(not q.empty()) miramos los nodos adyacentes y si not visited los añadimos a la q

Ejemplo: Distancia de un nodo a cualquier otro
Construimosun vector dist sobre los nodos y lo iniciamos a $-1$.
Realizamos un BFS sobre el grafo y en cada paso sumamos uno al adyacente.

Ejemplo: Camino mínimo entre dos nodos

## 5.3 Shortest path en ponderados

Cuando tenemos un grafo ponderado podemos utilizar Dikjstra para encontrar el camino más corto a cualquier nodo

**Algoritmo:** *Dikjstra* $\mathcal{O}(m + n \log n)$

- Consideramos el vector GP. Construimos los vectores sobre los nodos dist, father y la priority queue de distancias q . Inicializamos las distancias a INF .
- Inicializamos dist[ini]=0, q.push(P(0, ini))
- while(not q.empty()) seleccionamos el nodo de la q con menos distancia. Si esa distancia es igual a dist $\implies$ para cada nodo adyacente:
  - Miramos si la suma de la distancia al original mas el peso de la arista es menor que la dist de la adyacente
  - Si es menor, la insertamos en q , en dist y decimos que su father es la original

Si queremos buscar la distancia entre dos nodos cualesquiera podemos usar Floyd-Warshall

**Algoritmo:** *Floyd-Warshall* $\mathcal{O}(n^3)$

- Definimos el vector M[i, j] , que inicialmente es la matriz de adyacencia de G .
- Iteramos sobre $k, i, j$, y en cada iteración hacemos M[i][j] = min(M[i][j], M[i][k] + M[k][j]) .

Si en lugar de buscar el camino más corto buscamos el más largo desde un nodo, aplicamos un longest path:

**Algoritmo:** *Longest path* $\mathcal{O}(n + m)$

- Definimos un vector dist que nos el camino más largo y lo inicializamos a $-1$.
- Si dist[x]!=-1 retornamos dist[x] (programación dinámica)
- Si dist[x]==-1 retornamos max(1+y) para y vecino de x y lo igualamos a dist[x] .

(terminar)

## 5.4 Árbol Generador

Se trata de, dado un grafo $G$ calcular el subconjunto $F \subset E$ mínimo tal que el grafo sea un árbol.

**Algoritmo:** *Árbol generador en un grafo no dirigido sin pesos* $\mathcal{O}()$ Definiremos un representante de cada componente conexa y crearemos una función repr(x) que nos diga cuál es ese representante. Además usaremos compresión de caminos para que los nodos estén a una distancia menor a sus representantes.

- Creamos el vector pare , que nos dirá el nodo padre de cada nodo. Creamos la función repr() .
- Si pare(x) == -1 $\implies$ x es el nodo representante.
- Si no lo es, pare[x]=repr(pare[x]) y devuelve pare[x] .

Una vez implementada la función repr() vamos con la función principal:

- Definimos el entero comp=n , que nos indicará el número de componentes conexas.
- Iteramos sobre el conjunto de aristas (definidas en pairs):
  - Si los repr(x) != repr(y) $\implies$ hacemos que uno sea padre del otro.
  - Retornamos la arista.
  - Restamos 1 a comp . Si es 1 retornamos.

Si queremos hacer el Árbol mínimo de expansión en un grafo con pesos usaremos el algoritmo de Kruskal.
**Algoritmo:** *Kruskal* $\mathcal{O}(m \log m)$ Similar al de Árbol generador:

- Ordenamos las aristas por coste.
- Al realizar el algoritmo anterior iteramos sobre las aristas por coste: si están en diferente componente conexa, la añadimos, si no, cogemos la siguiente.

## 5.5 Ordenación topológica

La ordenación topológica es una ordenación lineal de todos los nodos que satisface que si G contiene la arista dirigida uv entonces el nodo u aparece antes del nodo v .

**Algoritmo:** *Ordenación Topológica* $\mathcal{O}(n+m)$

- Construimos el vector deg , que cuenta el grado de entrada de cada nodo y la stack pila
- for(nodo) si el deg del nodo tiene grado 0 $\implies$ añádelo a pila .
- while(not pila.empty()) print del nodo de la pila y restarle uno al deg sus adyacentes. Si alguno de esos adyacentes tiene deg=0 $\implies$ añádelo a pila .

## 5.6 Componentes fuertemente conexas (SCC)

$$[x] = [y] \text{ (están en la misma componente fuertemente conexa )} \iff \begin{cases} x \to y \\ y \to x \end{cases}$$

Todo grafo dirigido es un DAG de sus SCC

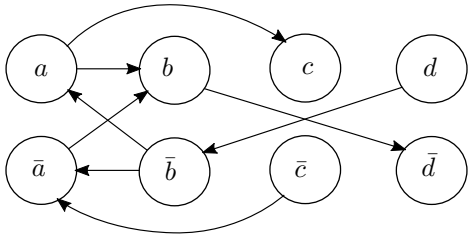
**Algoritmo:** *Identificación de las SCC* $\mathcal{O}(n+m)$

- Realizar un contador DFS sobre el grafo y guardar los vértices resultantes en en una stack S .
- Construir el Grafo transpuesto I invirtiendo las direcciones de las aristas
- Sobre I, miramos el elemento de la stack S y hacemos un DFS. Los nodos que recorra serán miembros de la misma componente conexa. Realizamos este proceso hasta que se vacíe S

Ejemplo: 2-SAT problem
Pongamos que queremos ver si se pueden elegir bools $a, b, c, d$ tal que:

$$(a \vee b) \wedge (\bar{a} \vee c) \wedge (\bar{a} \vee b) \wedge (\bar{b} \vee \bar{d}) = true \implies \begin{cases} \bar{a} \implies b, & \bar{b} \implies a \\ a \implies c, & \bar{c} \implies \bar{a} \\ a \implies b, & \bar{b} \implies \bar{a} \\ b \implies \bar{d}, & d \implies \bar{b} \end{cases}$$

Podemos hacer un grafo dirigido con $2n$ nodos (los bools y sus complementarios)
Si este grafo tiene una componente fuertemente conexa, entonces no se pueden asignar los booleanos.



## 5.7 Flujo máximo

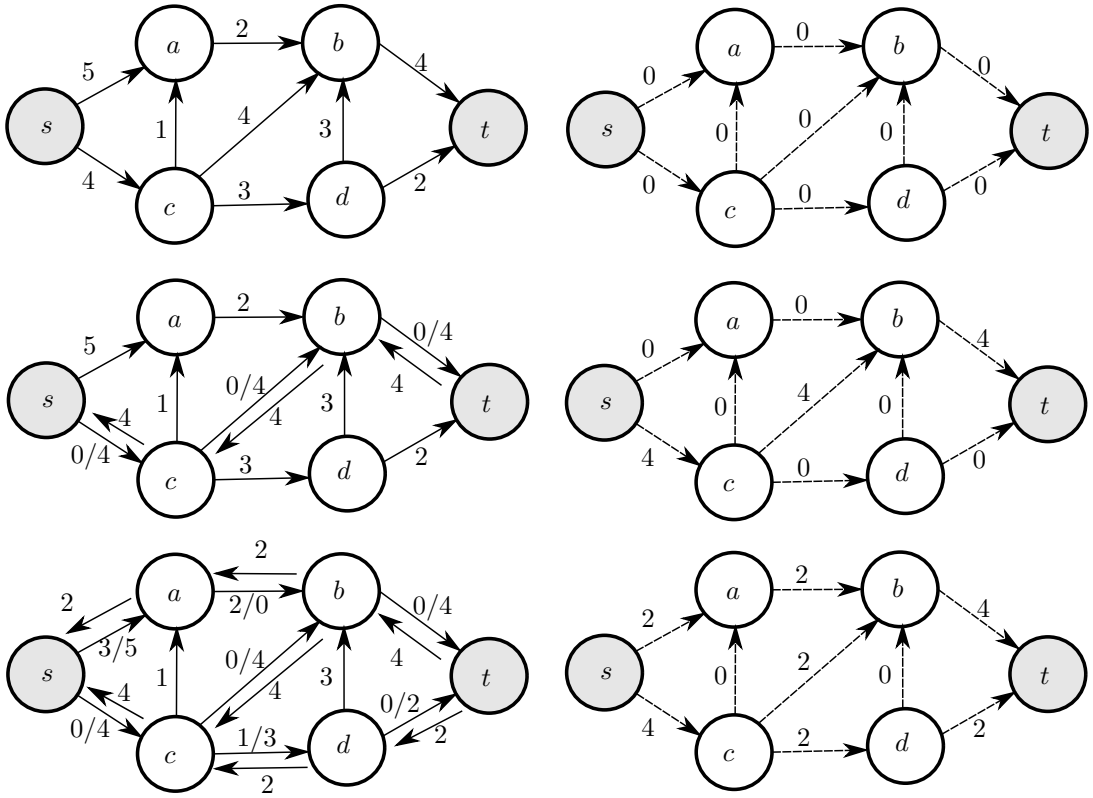
Suponemos que tenemos ciertas ciudades (nodos) y carretaras (aristas dirigidas) con una cierta capacidad. Consideramos ir de una ciudad origen (fuente) a una final (pozo), cumpliendo, además las restricciones en las capacidades de las carreteras. La solución a la capacidad máxima es el *flujo máximo*.

**Algoritmo:** *Flujo máximo* $\mathcal{O}()$

Empezamos construyendo un grafo $F$ con la misma estructura pero inicializando a 0 los pesos de las aristas.
Para cada iteración Hacemos:

- Buscamos un camino $s \to t$ y hacemos pasar el máximo flujo posible (habrá una arista que haga de cuello de botella).
- Sumamos ese flujo en el camino en $F$. Restamos en $G$ ese flujo y además lo ponemos en sentido contrario.
- Repetimos el proceso mientras exista un camino donde pasar flujo $s \to t$.

Se adjunta un ejemplo sencillo. La primera columna corresponde al grafo $G$ y la segunda al $F$ conforme se van actualizando.



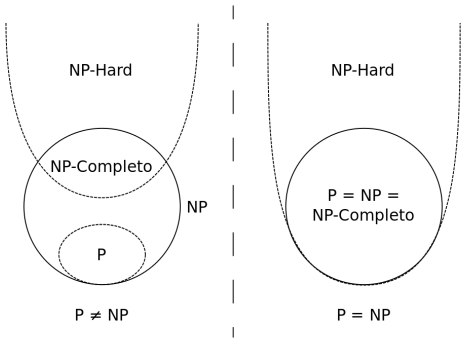

Ejemplo: Matching máximo
Buscamos en un grafo bipartido $V = E + D$ el máximo número de aparejamientos.
Podemos replantear el problema añadiendo los nodos $s$ y $t$ tal y las aristas $s \to E, D \to t$. Realizando el maxflow del nuevo grafo de $s$ a $t$, la solución será el número de parejas.

# 6 Roura out of context

## 6.1 P versus NP

### 6.1.1 Definiciones

- **P:** Se pueden resolver en tiempo polinómico.
- **NP:** Dada una solución se puede comprobar en tiempo polinómico.
- **NP-Hard:** Todo problema NP se puede reducir a este problema.
- **NP-Completo:** Se cumple que es NP y NP-Hard.




La pregunta es si $P = NP$ o $P \neq NP$. En función de la respuesta tenemos los dos tipos de diagramas de Venn.

### 6.1.2 Ejemplos de Problemas NP-completos

- **3-SAT** Problema de decisión booleana. Similar al 2-SAT visto en la sección de algoritmos, pero con 3 variables en cada cláusula
- **Knapsack** Dados unas cantidades de diferentes objetos, sus pesos y su valor, buscar el máximo de valor que se pueden meter en una mochila con capacidad de peso máxima dada.
- **Subset Sum** dado un conjunto de números buscar un subconjunto de ellos que sume un determinado número
- **Camino Hamiltoniano** Buscar un camino Hamiltoniano (que pase por todos los vértices una sola vez) en un grafo
- **Clique** Buscar cliques (subgrafos completos) en un grafo
- **Subgraph isomorphism** dados dos grafos $G_1, G_2$ ver si algún subgrafo de $G_1$ es isomorfo a $G_2$
- **Vertex covering** Buscar el covering (subconjunto de vértices tal que toda arista tiene al menos un vértice en el) mínimo en un grafo
- **Graph coloring** Mínimo numero de colores con el que se puede colorear un grafo

## 6.2 Problema de la parada

Este teorema nos dice que es imposible diseñar un algoritmo que nos diga si un cierto código dado como input parará en un tiempo finito.

Supongamos que si que tenemos tal algoritmo. Definimos el algoritmo como

$$halt(x) = \begin{cases} 1 \text{ si el código } x \text{ se para} \\ 0 \text{ si el código } x \text{ no se para} \end{cases}$$

Entonces podemos implementar la función $g()$ de la siguiente manera:

```
void g(){
  if(halt(g)){
        loop_forever();
  }
}
```

Observamos que si $halt(g) = 1$, eso significaría que la función g llama a $\text{loop}_f orever() \implies$ contradicción. Por otro lado, si $halt(g) = 0$, eso significaría que la función se para $\implies$ contradicción.

Por tanto no existe tal algoritmo.

### 6.3 Jocs imparcials

Definimos los **nimbers** como la función recursiva:

$$nim(x) = \min\ n \notin \{nim(X') : X \to X'\}$$

Donde $X$ es el conjunto de las posiciones posibles del juego.

Se cumple que una posición $x$ es perdedora si y solo si $nim(x) = 0$. Esto lo podemos ver por la definición de la función $nim$: si una posición es perdedora, no se podrá llegar desde ella a ninguna posición ganadora y viceversa.

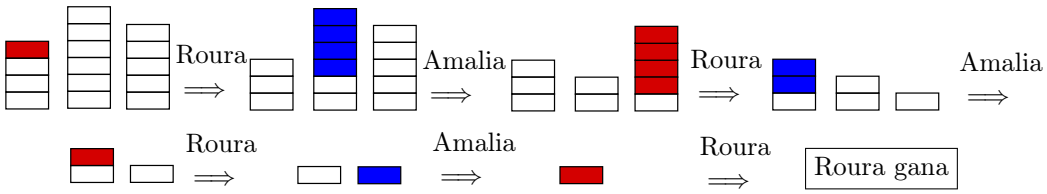
Podemos hacer el "producto" de dos juegos imparciales representado por la operación $\bigoplus$. Si tenemos los juegos $J_i$ y tenemos una posición de cada juego $x_i \in J_i$, entonces la posición del producto de juegos es $x_1 \oplus \ldots \oplus x_n \in J_1 \bigoplus \ldots \bigoplus J_n$.

Un resultado interesante es que $nim(x \oplus y) = nim(x) \wedge nim(y)$ donde $\wedge$ es la operación exclusive or.

El juego por excelencia donde se aplica es el **Joc del Nim** y sus diversas variantes. La mas sencilla de ellas es la siguiente:

*Joc del Nim*
Tenemos $m$ pilas con un número de fichas cada una y dos jugadores que se enfrentan. En cada turno al jugador que le toca puede quitar las fichas que quiera de una sola pila. Gana quien quite la última ficha en su turno.

Se adjunta a continuación un posible juego entre Roura y Amalia (en rojo y azul las fichas que quitan respectivamente):




Roura tiene una estrategia ganadora. Vamos a verlo calculando los nimbers de cada pila por separado.

El caso base $nim(0) = 0.nim(1) = 1$ por la fórmula recursiva ... por inducción $nim(k) = k$. Por tanto tenemos:

$$\begin{cases} nim(4) = 4 = 100_2 \\ nim(6) = 6 = 110_2 \\ nim(5) = 5 = 101_2 \end{cases} \implies nim(4 \wedge 6 \wedge 5) = nim(100_2 \wedge 110_2 \wedge 101_2) = 111 \neq 0 \implies \boxed{\text{posición ganadora}}$$

# 7 Apéndice de los códigos en C++

```
using namespace std;
#include <iostream>
#include <queue>
#include <stack>
#include <vector>

using ll = long long;
using P = pair<int, int>;
using VP = vector<P>;
using VVP = vector<VP>;
```

```
using VI = vector<int>;
using VVI = vector<VI>;
using VB = vector<bool>;

VVI M;  // Matriz de adyacencia
VVI G;  // Lista de aristas
VP V;   // Pair con las aristas
VVP GP; // Conjunto de veritces en pairs (con costes)
int n = M.size();
int m = G.size();
int INF = 1e9;

//=============================================
// DFS
// O(n+m)
VI visited(n, false);
void DFS(int u) {
  if (visited[u])
    return;
  visited[u] = true;
  cout << u << endl;
  for (int v : G[u]) {
    DFS(v);
  }
}

//=============================================
// BFS
// O(n+m)
void BFS(int u) {
  VI visited(n, false);
  queue<int> Q;
  visited[u] = true;
  Q.push(u);
  while (not Q.empty()) {
    int x = Q.front();
    Q.pop();
    cout << x << endl;
    for (int y : G[x])
      if (not visited[y]) {
        visited[y] = true;
        Q.push(y);
      }
  }
}

//=============================================
// Dijkstra
// O(n+ln(m))
void Dijkstra(int ini) {
  VI dist(n, INF);
  VI pare(n);
  priority_queue<P> Q;
  dist[ini] = 0;
  Q.push(P(0, ini));
  while (not Q.empty()) {
    P p = Q.top();
    Q.pop();
```

```
      int d = -p.first;
      int x = p.second;
      if (d == dist[x]) {
        for (P arc : GP[x]) {
          int c = arc.first;
          int y = arc.second;
          int d2 = d + c;
          if (d2 < dist[y]) {
            dist[y] = d2;
            pare[y] = x;
            Q.push(P(-d2, y));
          }
        }
      }
    }
}

//=============================================
// Distancia minima entre dos vertices
// O(n^3)
VVI FloydWarshall() {
  for (int k = 0; k < n; ++k)
    for (int i = 0; i < n; ++i)
      for (int j = 0; j < n; ++j)
        M[i][j] = min(M[i][j], M[i][k] + M[k][j]);

  return M;
}

//=============================================
// Longitud del camino mas largo en DAG (Dinamica)
// O(n+m)
VI R(n, -1);
int f(int x) {
  int &res = R[x];
  if (res != -1)
    return res;
  res = 0;
  for (int y : G[x])
    res = max(res, 1 + f(y));
  return res;
}

//=============================================
//Arbol generador
VI pare(n);
int repre(int x) {
  if (pare[x] == -1)
    return x;
  int r = repre(pare[x]);
  pare[x] = r;
  return r;
}

void arbre_generador() {
  pare = VI(n, -1);
  int compo = n;
  for (P p : V) {
```

```
    int x = p.first;
    int y = p.second;
    int rx = repre(x);
    int ry = repre(y);
    if (rx != ry) {
      pare[ry] = rx;
      // pare[y] = x; error tipico
      cout << x << ' ' << y << endl;
      if (--compo == 1)
        return;
    }
  }
  cout << "graf no connex!" << endl; // n > 1
}

//=============================================
// Ordenacion topologica
// O(n+m)
void TopoSort() {
  VI deg(n, 0); // grados de cada nodo (rellenar al leer)
  stack<int> pila;
  for (int x = 0; x < n; ++x) {
    if (deg[x] == 0)
      pila.push(x);
    while (not pila.empty()) {
      int x = pila.top();
      pila.pop();
      cout << x << endl;
      for (int y : G[x]) {
        if (--deg[y] == 0)
          pila.push(y);
      }
    }
  }
}

int main() {}
```