# Algorísmia FME: Teoria

## 1 Sessió 1: Eficiència dels algorismes

Donada una solució a un problema algorísmic, és natural preguntar-se si es podria millorar. Per respondre aquesta pregunta, cal una manera de mesurar l'eficiència d'un algorisme i poder comparar-lo amb altres.

Una definició bàsica d'eficiència seria "la quantitat de recursos que utilitza del sistema", i aquests recursos són principalment el temps i l'espai (memòria).

Podríem mesurar aquesta eficiència experimentalment, però llavors el resultat dependria de factors externs a l'algorisme, com ara el hardware on s'executa, el llenguatge de programació que es fa servir... A més, volem una manera de predir com es comportarà l'eficiència de l'algorisme davant de *inputs* de mida arbitrària.

Per això, l'eficiència en temps d'un algorisme es mesura com la quantitat d'operacions elementals (sumes, assignacions, comparacions, productes, divisions...) que s'executen. L'eficiència en espai es mesura com la quantitat de memòria requerida per executar-lo.

Definirem el cost d'un algorisme com una funció que a tota entrada possible de l'algorisme li assigna el seu cost en temps i espai. Necessitem també que el cost de l'algorisme sigui funció del tamany de l'entrada, que anomenarem $n$.

$$\begin{aligned} T_n \colon A_n &\longrightarrow R^+ \\ a_n &\longmapsto T_n(a_n) \end{aligned}$$

## 1.1 Cas pitjor, millor i mitjà

- Cas pitjor: De totes les entrades possibles, prenem la que resulti en el màxim temps i/o memòria utilitzades.

$$T_{pitjor}(n) = \max_{a_n \in A_n} \{T_n(a_n)\}$$

- Cas millor: De totes les entrades possibles, prenem la que resulti en el mínim temps i/o memòria utilitzades.

$$T_{millor}(n) = \min_{a_n \in A_n} \{T_n(a_n)\}$$

- Cas mitjà: Calculem la mitjana ponderada segons la probabilitat que l'entrada sigui una de donada. Necessitarem, és clar, la distribució de probabilitats de les entrades.

$$T_{avg}(n) = \sum_{a_n \in A_n} \Pr(a_n) * T_n(a_n)$$

## 1.2 Big O, Big Omega, Big Theta

Sigui $f : \mathbb{N} \to \mathbb{R}^+$.

- Definim el conjunt de funcions $O(f)$ com:

  $O(f) = \{g : \mathbb{N} \to \mathbb{R}^+ \text{ tals que } \exists c > 0, \exists n_0 \in \mathbb{N}, \forall n > n_0, f(n) \geq c \cdot g(n)\}$

- Definim el conjunt de funcions $\Omega(f)$ com el simètric de $O(f)$:

  $\Omega(f) = \{g : \mathbb{N} \to \mathbb{R}^+ \text{ tals que } f \in O(g)\}$

- Definim el conjunt de funcions $\Theta(f)$ com la intersecció de $O(f)$ i $\Omega(f)$:

  $\Theta(f) = O(f) \cap \Omega(f)$

Informalment, diem que $O(f)$ són les funcions que són asimptòticament menors o iguals que $f$, $\Omega(f)$ són les funcions que són asimptòticament majors o iguals que $f$, i que $\Theta(f)$ són les funcions asimptòticament iguals que $f$, tot això llevat d'una constant.

**Notació.** En comptes de dir $g \in O(f)$, sovint farem un abús de notació i direm $g = O(f)$.

**Exemple 1: Selection Sort**

Si tenim un vector de $n$ elements, fem $n-1$ comparacions per trobar el primer element, $n-2$ per trobar el segon, etc. Això suma un total de $O(n^2)$ comparacions. En canvi, fem $O(n)$ intercanvis. El terme dominant és el de les comparacions, i per tant el cost de l'algorisme serà de $O(n^2)$.

**Exemple 2: Insertion Sort**

El cas millor és de fer $n-1$ comparacions i cap intercanvi (vector ja ordenat). Per tant, $T_{millor}(n) = \Theta(n)$.

El cas pitjor és de fer $O(n^2)$ comparacions i $O(n)$ intercanvis. Per tant, $T_{pitjor}(n) = \Theta(n^2)$

**Exemple 3: Cerca dicotòmica**

El cas millor és de fer una sola comparació (trobar-lo a la primera). $T_{millor}(n) = \Theta(1)$.

El cas pitjor és de fer de l'ordre de $\log n$ comparacions. $T_{pitjor}(n) = \Theta(\log n)$

**Observació.** La base del logaritme no importa en la notació asimptòtica ja que $\log_a(n) = \log_b(n) \cdot \log_a(b)$, on $\log_a(b)$ és una constant. Per tant, $\log_a(n) = \Theta(\log_b(n))$ per tot $a, b > 1$.

# 2 Sessió 2: Càlcul de cost d'algorismes

A la sessió anterior hem vist com es defineix el cost d'un algorisme formalment, amb les notacions Big O, Big Theta i Big Omega. En aquesta sessió veurem regles per calcular el cost d'algorismes concrets, tant iteratius com recursius.

## 2.1 Anàlisi d'algorismes iteratius

- El cost de les operacions elementals és de $\Theta(1)$.
- Composició seqüencial: Donats dos fragments de codi $s_1, s_2$ amb costos $f_1, f_2$, el cost del fragment de codi que resulta de concatenar $s_1$ i $s_2$ és $f_1 + f_2$.

```
1    s1;
2    s2;
3
```

- Composició condicional: Sigui $A$ una expressió booleana amb cost per avaluar-la $f_A$. Siguin $s_1, s_2$ dos fragments de codi amb cost $f_1$ i $f_2$. El cost, en cas pitjor, del fragment de codi següent és $f_A + \max\{f_1, f_2\}$

```
1    if (A) {
2        s1;
3    }
4    else {
5        s2;
6    }
7
```

- Composició iterativa: Sigui $A$ una expressió booleana amb cost per avaluar-la a la i-èsima iteració $A_i$, i $s$ un fragment de codi amb cost a la i-èsima iteració $s_i$. El cost, en el cas pitjor, d'executar $n$ iteracions del fragment de codi següent és

$$\sum_{i=1}^{n}(A_i + s_i) = O(n \cdot (\max A_i + \max s_i))$$

```
1    while (A) {
2        s;
3    }
4
```

## 2.2 Anàlisi d'algorismes recursius

En el cas dels algorismes recursius, el cost ve descrit per una recurrència. Algunes d'aquestes recurrències es poden resoldre mitjançant certs Teoremes, que en donen el cost asimptòtic. Dos tipus importants de recurrències són les substractores i les divisores.

### 2.2.1 Recurrències substractores

Diem que una recurrència és substractora si té la forma

$$T(n) = \begin{cases} f & \text{si } n \leq c \\ a \cdot T(n-c) + g & \text{si } n > c \end{cases}$$

on $c \geq 1$, $g = \Theta(n^k)$.

**Teorema Mestre** per a recurrències substractores

Donada una recurrència de la forma anterior, tenim que

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

### 2.2.2 Recurrències divisores

Diem que una recurrència és divisora si té la forma

$$T(n) = \begin{cases} f & \text{si } n \leq b \\ a \cdot T(n/b) + g & \text{si } n > b \end{cases}$$

on $a \geq 1$, $b \geq 2$ i $g = \Theta(n^k)$.

**Teorema Mestre** per a recurrències divisores

Donada una recurrència de la forma anterior, sigui $\alpha = \log_b(a)$. Llavors, tenim que

$$T(n) = \begin{cases} \Theta(n^k) & \text{si } \alpha < k \\ \Theta(n^k \log n) & \text{si } \alpha = k \\ \Theta(n^\alpha) & \text{si } \alpha > k \end{cases}$$

**Exemple 1: Cerca dicotòmica**

La cerca dicotòmica té un cost que ve donat per la recurrència divisora següent:

$$T(n) = \begin{cases} \Theta(1) & \text{si } n \leq 2 \\ 1 \cdot T(n/2) + \Theta(1) & \text{si } n > 2 \end{cases}$$

Per tant, $\alpha = \log_2(1) = 0$. D'altra banda, $g = \Theta(1) = \Theta(n^0)$, i per tant $k = 0$.

Com que $\alpha = k$, ens trobem en el segon cas de Teorema Mestre per a recurrències divisores, i $T(n) = \Theta(n^0 \log n) = \Theta(\log n)$

**Exemple 2: Exponenciació ràpida**

L'exponenciació ràpida té un cost que ve donat per la recurrència divisora següent:

$$T(n) = \begin{cases} \Theta(1) & \text{si } n \leq 2 \\ 1 \cdot T(n/2) + \Theta(1) & \text{si } n > 2 \end{cases}$$

Per tant, $\alpha = \log_2(1) = 0$. D'altra banda, $g = \Theta(1) = \Theta(n^0)$, i per tant $k = 0$.

Altra vegada, $\alpha = k$, ens trobem en el segon cas de Teorema Mestre per a recurrències divisores, i $T(n) = \Theta(n^0 \log n) = \Theta(\log n)$

**Exemple 3: Nombres de Fibonacci**

Farem servir l'algorisme (molt ineficient) de cridar recursivament les funcions per a $n-1$ i $n-2$ sense guardar cap resultat en memòria. De manera exacta, la recurrència és:

$$T(n) = \begin{cases} \Theta(1) & \text{si } n \leq 2 \\ T(n-1) + T(n-2) + \Theta(1) & \text{si } n > 2 \end{cases}$$

Com que el Teorema Mestre per a recurrències substractores no ens resol aquest cas, el millor que podem fer es fitar-ne el cost superiorment i inferiorment, de la manera següent:

$$T_{sup}(n) = \begin{cases} \Theta(1) & \text{si } n \leq 1 \\ 2 \cdot T(n-1) + \Theta(1) & \text{si } n > 1 \end{cases}$$

$$T_{inf}(n) = \begin{cases} \Theta(1) & \text{si } n \leq 2 \\ 2 \cdot T(n-2) + \Theta(1) & \text{si } n > 2 \end{cases}$$

Tenim que, pel Teorema Mestre, $T_{inf}(n) = \Theta(2^{n/2})$, i $T_{sup}(n) = \Theta(2^n)$.

Així, per definició de Big O i Big Omega,

$$T(n) = O(2^n) \text{ i } T(n) = \Omega((\sqrt{2})^n)$$

.

Amb el que sabem de Fibonacci podem determinar que el cost real és d'aproximadament $\Theta(\varphi^n)$, on $\varphi \approx 1.61$ és la constant àuria.

# 3 Sessió 3: Dividir i vèncer

Els algorismes de dividir i vèncer es basen en que, per certs problemes, trencar el problema en subproblemes, resoldre recursivament aquests subproblemes i després combinar les seves solucions pot ser una manera eficient d'afrontar-los. Veurem alguns exemples d'aquests algorismes.

## 3.1 Exponenciació ràpida

### 3.1.1 Exponenciació ràpida de naturals

Volem resoldre una operació del tipus $a^n$, on $a \in \mathbb{N}$ és una constant, en temps $\Theta(\log n)$. Per fer-ho, ens aprofitarem del fet que $a^{2k} = (a^2)^k$, per obtenir una recurrència divisora.

**Algorisme**

```
1  int exponenciacio_rapida(int a, int n) {
2      //Casos base
3      if (n == 0) return 1;
4      if (n == 1) return a;
5
6      //Cas recursiu divisor
7      if (n % 2 == 0) {
8          return exponenciacio_rapida(a*a, n/2);
9      }
10
11     //Cas recursiu substractor
12     else {
13         return a*exponenciacio_rapida(a, n-1);
14     }
15 }
```

Ens adonem que, en aquesta recurrència, hi ha una crida recursiva divisora i una de substractora. Ens interessa que la substractora es cridi el menys possible, ja que aquestes recurrències

creixen molt més ràpidament. Analitzem, doncs, quantes vegades es pot cridar com a màxim la recurrència substractora:

Sigui $n = n_s n_{s-1} ... n_1 n_0$ l'expressió binària de $n$, en la qual $n_i \in \{0, 1\}$. Aleshores, si $n_0 = 0$, el nombre serà parell i podrem cridar la recurrència divisora, quedant-nos així amb el nombre $n' = n_s n_{s-1} ... n_1$. D'altra banda, si $n_0 = 1$, el nombre serà senar i per tant haurem de cridar la recurrència substractora, quedant-nos amb el nombre $n'' = n_s n_{s-1} ... n_1 0$.

Llavors, si l'objectiu és arribar al cas base, que és un nombre d'un sol bit, veiem que el pitjor cas és que l'expressió binària de $n$ sigui $n = 111...111$. En aquest cas, caldran $2(s-1) = 2s-2$ crides recursives per arribar a tenir un sol bit, ja que per cada parell de crides, la primera transforma l'últim bit en 0, i la segona l'elimina dividint per 2.

Finalment, només cal veure que $s = \lfloor (\log_2 n) \rfloor$, ja que és el nombre de bits de $n$. Així, fem un total de $\Theta(\log n)$ crides recursives, amb un cost de $\Theta(1)$ cadascuna (un producte, un mòdul i una comparació). En total, el cost de l'algorisme és de $\Theta(\log n)$.

### 3.1.2 Exponenciació ràpida de matrius

L'algorisme d'exponenciació ràpida de matrius segueix exactament la mateixa idea que el de naturals, amb l'única diferència que estem intentant calcular $A^n$, on $A \in \mathcal{M}_m(\mathbb{R})$. L'anàlisi de cost és exactament la mateixa, exceptuant que el producte de dues matrius quadrades $mxm$ és d'ordre $\Theta(m^3)$ (aplicant l'algorisme senzill de producte de matrius), i per tant el cost total serà de $\Theta(m^3 \log n)$. En el codi següent suposarem que l'operació producte de matrius està implementada en l'operador $*$, i que disposem d'una funció identitat($m$) que retorna una matriu identitat de mida $m$.

#### Algorisme

```
1 //El tamany de A es m x m, m esta definida fora de la funcio
2 Matriu exponenciacio_rapida(int A, int n) {
3     //Casos base
4     if (n == 0) return identitat(m);
5     if (n == 1) return A;
```

```
6
7       //Cas recursiu divisor
8       if (n % 2 == 0) {
9           return exponenciacio_rapida(A*A, n/2);
10      }
11
12      //Cas recursiu substractor
13      else {
14          return A*exponenciacio_rapida(A, n-1);
15      }
16  }
```

### 3.1.3 Fibonacci en O(log n)

A la sessió anterior hem vist com calcular els nombres de Fibonacci amb cost $\Theta(\varphi^n)$, que és pràcticament la manera menys eficient possible de calcular-los. Hi ha diverses opcions de calcular-los en temps $\Theta(n)$, com ara amb un bucle iteratiu o amb programació dinàmica. En aquesta secció veurem com aprofitar l'algorisme d'exponenciació ràpida de matrius per calcular-los de la manera més eficient possible, en temps $\Theta(\log n)$.

La clau d'aquest algorisme es troba en la matriu següent:

$$A = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}$$

Es pot demostrar (fàcilment per inducció) que l'element de la segona fila i segona columna de $A^n$ és igual a l'enèsim nombre de Fibonacci. Així, l'algorisme per calcular els nombres de Fibonacci es redueix a exponenciar aquesta matriu $A$ amb l'algorisme anterior. El cost d'aquest programa és $\Theta(m^3 \log n) = \Theta(2^3 \log n) = \Theta(8 \log n) = \Theta(\log n)$.

## 3.2 Merge Sort

En les sessions anteriors hem vist dos algorismes d'ordenació, el Selection Sort i l'Insertion Sort, que tenien els dos un cost en el pitjor cas de $\Theta(n^2)$. Ara veurem un algorisme més eficient, que en el pitjor cas té cost $\Theta(n \log n)$. El Merge-Sort, o ordenació per fusió, es basa en els principis de dividir i vèncer, aprofitant que la fusió de dos vectors ordenats (de manera que mantingui l'ordre) es pot fer en temps lineal.

En l'algorisme següent assumirem que tenim una funció `void merge(vector<int>& v, int left, int mid, int right)`, que ens fusiona dos subvectors [left, mid-1] i [mid, right] ja ordenats en temps lineal respecte a la suma de les mides dels dos subvectors.

### Algorisme

```
1  // left val inicialment 0, right val inicialment v.size()-1. El
       segment a ordenar es [left, right]
2  void merge_sort(vector<int>& v, int left, int right) {
3      // Cas base: Si te mida 0 o 1 ja esta ordenat
4      if (right <= left) return v;
5
6      //Cas recursiu
7      int mid = (left+right)/2;
8      merge_sort(v, left, mid-1)
9      merge_sort(v, mid, right);
10     merge(v, left, mid, right);
11 }
```

Vegem que el cost d'aquest algorisme és $\Theta(n \log n)$.

Assumim que $n$ és una potència de 2 per simplificar els càlculs. En tot cas, això no altera la complexitat de l'algorisme. En fer les subdivisions del vector $v$, estem creant un arbre binari complet (perquè $n$ és potència de 2). Aquest arbre té profunditat $\log_2(n) + 1 = \Theta(\log n)$.

En el nivell $i$-èsim de l'arbre ($i = 0, 1, ..., \log_2(n)$), tenim $2^i$ subvectors. Cadascun d'aquests subvectors té $\frac{n}{2^i}$ elements. Fusionar dues d'aquestes llistes té cost $\Theta(\frac{n}{2^{i-1}})$ ja que és la suma

de les mides de dos subvectors. Aquesta operació s'ha de fer $2^{i-1}$ vegades a cada nivell, una per cada parella de vectors a ordenar. Així, el cost total de fusionar totes les parelles de subvectors d'un nivell és $\Theta(n)$, i no depèn del nivell.

Com que hem vist que hi ha $\Theta(\log n)$ nivells, el cost de l'algorisme és $\Theta(n \log n)$.

## 3.3 Arbres de decisió

Suposem en aquesta secció que tots els elements del vector a ordenar són diferents, i que per tant els únics resultats possibles d'una comparació són $a < b$ i $a > b$.

Tots els algorismes d'ordenació es poden representar com un arbre binari de decisió. Cada node representa el conjunt de possibles ordenacions d'acord amb les comparacions que s'han fet fins al moment, i de cada node en surten dos vèrtexs que representen els dos possibles resultats de la comparació actual.

El nombre màxim de comparacions necessàries per assegurar que el vector està ordenat ve donat per la màxima profunditat d'una fulla de l'arbre.

Com que una fulla representa una ordenació dels elements, un arbre de decisió sobre un vector de mida $n$ té $n!$ fulles, una per cada permutació dels elements. La profunditat de l'arbre, per tant, ha de ser com a mínim $\log(n!)$, que seria el cas d'un arbre complet.

**Teorema.** El cost en cas pitjor dels algorismes d'ordenació basats en comparacions és de $\Omega(n \log n)$.

Per demostrar aquest teorema, hem vist que la profunditat de l'arbre de decisió ha de ser d'almenys $\log(n!) > \log[(n/2)^{n/2}] = (n/2)\log(n/2) = \Omega(n \log n)$. Per tant, no existeix un algorisme d'ordenació basat en comparacions amb un cost asimptòtic menor a $n \log n$.

**Corol·lari.** El cost de Merge-Sort és òptim asimptòticament.

# 4 Sessió 4: Dividir i vèncer [2]

## 4.1 Algorisme de Floyd-Rivest

Tenim un vector no ordenat de mida $n$, i volem trobar-hi, donada $0 <= k < n$, el $k$-èsim element més petit. D'entrada, l'opció més senzilla seria d'ordenar el vector en temps $\Theta(n \log n)$ mitjançant Merge-Sort o qualsevol altre algorisme eficient d'ordenació, i després obtenir el $k$-èsim element accedint-hi en temps constant. Hi ha, però, una manera més eficient d'arribar a la solució, i és amb l'algorisme de Floyd-Rivest.

Exposarem l'algorisme en pseudocodi, i després n'analitzarem el cost asimptòtic.

**Algorisme**

```
1. Dividir el vector en subvectors de 5 elements. L'ultim pot tenir
   menys de 5 elements.
2. Crear un vector de mida n/5 amb la mediana de cada subvector
3. Aplicar recursivament tot l'algorisme al vector de les medianes
   amb k = n/10, per obtenir la mediana de les medianes. Anomenem-la
    M.
4. Dividir el vector original el 3 subvectors: els elements menors
   que M (v1), iguals a M (v2), majors que M (v3).
5. Si v1.size() >= k, es que el k-esim element es troba a v1.
   Apliquem recursivament l'algorisme al vector v1 amb la k original
6. Si no, si v1.size() + v2.size() >= k, es que el k-esim element es
    troba a v2. Retornem M.
7. Si no, es que el k-esim element es troba a v3. Apliquem
   recursivament l'algorisme a v3 amb k' = k - v1.size() - v2.size()
```

Per què té cost lineal aquest algorisme? L'analitzarem per inducció.

Veurem que $T(n) \leq 20 \cdot c \cdot n$, on $c$ és una constant. La recurrència que dóna $T$ és:

$$T(n) \leq \begin{cases} \Theta(1) & \text{si } n \leq 50 \\ T(\frac{n}{5}) + T(\frac{3n}{4}) + c \cdot n & \text{si } n > 50 \end{cases}$$

Procedirem per inducció forta. El cas base, $n < 50$, és cert ja que l'algorisme és de temps constant en aquest cas.

Suposem que per tots els valors $51, 52, ..., n-1$ és certa la hipòtesi d'inducció: $T(k) \leq 20 \cdot c \cdot k$ per tot $k < n$.

Aleshores,

$$T(n) \leq T(\frac{n}{5}) + T(\frac{3n}{4}) + c \cdot n \leq \frac{20 \cdot c \cdot n}{5} + \frac{20 \cdot c \cdot 3n}{4} + c \cdot n = 20 \cdot c \cdot n$$

I per inducció hem provat el resultat.

## 4.2 Longest Increasing Subsequence

En aquest problema, se'ns dóna com a entrada una seqüència $S = s_1, ..., s_n$, i se'ns demana la longitud $l$ de la subseqüència creixent més llarga de $S$.

Direm que $S'$ és subseqüència de $S$ si es pot obtenir esborrant elements de $S$.
Direm que $S'$ és creixent si $S'_i < S'_{i+1}$ per tot $i = 0, ..., k-1$, on $k$ és la mida de $S'$.

En aquest algorisme, suposarem que tenim implementada una funció

```
int cerca_dic(vector<int>& v, int x, int left, int right);
```

que cerca en temps logarítmic l'element $x$ al vector $v$.

La idea de l'algorisme és tenir un vector $B$ en el qual anem guardant, en la posició $i$-èsima, l'element més petit que podria ser l'últim d'una subseqüència creixent de mida $i$, amb els elements que hem visitat fins al moment. Òbviament, en la primera iteració el primer element sempre serà el més petit que pot acabar una subsuccessió creixent de mida 1. En les altres

iteracions, mirem que l'element no sigui el més petit ni el més gran dels ja visitats, i després fem una cerca dicotòmica per trobar a quina posició aniria l'element actual.

### Algorisme

```
int LIS(vector<int>& S) {
    int n = S.size();
    vector<int> B;
    B.push_back(S[0]);
    int l = 0;
    for (int i = 1; i < n; i++) {
        if (S[i] < B[0]) B[0] = S[i];
        else if (S[i] > B[l]) {
            l++;
            B.push_back(S[i]);
        }
        else {
            int k = cerca_dic(B, S[i], 0, L);
            B[k] = S[i];
        }
    }
    return l+1;
}
```

La complexitat d'aquest algorisme és de $\Theta(n \log n)$. Això s'obté fàcilment de l'expressió següent:

$$T(n) = \sum_{i=1}^{n} \log i = \log(n!) = \Theta(n \log n)$$

on l'última igualtat l'hem vista prèviament quan hem estudiat Merge-Sort.

# 5 Sessió 5: Dividir i vèncer [3]

## 5.1 Multiplicació d'enters grans (Karatsuba i Ofman)

Si tenim un enter $x$ que ocupa $n$ bits (suposem $n$ potència de 2, si no ho és posem zeros a l'esquerra fins que ho sigui), el podem expressar com a $x_L \cdot 2^{n/2} + x_R$, on $x_L$ i $x_R$ són enters que ocupen $n/2$ bits cadascun.

Si tenim $x$ i $y$ dos enters de $n$ bits cadascun, la manera trivial de multiplicar $x$ i $y$ és mitjançant l'algorisme de la multiplicació bit a bit, que té un cost de $\Theta(n^2)$. Volem millorar aquest cost, i per això busquem la manera d'aplicar el principi de dividir i vèncer amb la separació obtinguda anteriorment.

$$x \cdot y = (x_L \cdot 2^{n/2} + x_R) \cdot (y_L \cdot 2^{n/2} + y_R) = x_L \cdot y_L \cdot 2^n + (x_L \cdot y_R + x_R \cdot y_L) \cdot 2^{n/2} + x_R \cdot y_R$$

Si analitzem aquesta expansió, resulta que tenim quatre productes d'enters de $n/2$ bits. Per tant, $T(n) = 4T(n/2) + O(n)$. Aplicant el Teorema Mestre, resulta que $T(n) = \Theta(n^2)$. Per tant, amb aquesta expansió no hem aconseguit cap millora en l'eficiència. Necessitem reduir el nombre de multiplicacions necessàries, de 4 a 3. Per això, farem servir una altra manera d'expressar $x$ i $y$.

$$x \cdot y = x_L \cdot y_L \cdot 2^n + [(x_L + x_R) \cdot (y_L + y_R) - x_L \cdot y_L - x_R \cdot y_R] \cdot 2^{n/2} + x_R \cdot y_R$$

En aquesta expressió, només tenim 3 productes únics (n'hi ha dos de repetits, que no cal calcular dues vegades). Per tant, $T(n) = 3T(n/2) + O(n)$, i pel Teorema Mestre, $T(n) = n^{\log_2(3)} \approx n^{1.58}$. Aquesta és una manera de multiplicar enters grans que millora l'eficiència de l'algorisme de la multiplicació bit a bit.

**Observació.** Multiplicar per una potència de 2 és equivalent a fer un *bit shifting* a l'expressió binària, i es pot considerar $\Theta(1)$.

## 5.2 Multiplicació de matrius grans (Strassen)

Si tenim dues matrius $X, Y \in \mathcal{M}_n(\mathbb{R})$, les podem dividir en quatre blocs cadascuna: $A, B, C, D, E, F, G, H \in \mathcal{M}_{n/2}(\mathbb{R})$, de manera que:

$$X = \begin{pmatrix} A & B \\ C & D \end{pmatrix} \qquad Y = \begin{pmatrix} E & F \\ G & H \end{pmatrix}$$

Si fem el producte per blocs, obtenim que:

$$X \cdot Y = \begin{pmatrix} A & B \\ C & D \end{pmatrix} \begin{pmatrix} E & F \\ G & H \end{pmatrix} = \begin{pmatrix} AE + BG & AF + BH \\ CE + DG & CF + DH \end{pmatrix}$$

Si analitzem el cost de fer les 8 multiplicacions de matrius més petites (i el cost quadràtic de les sumes), obtenim:

$$T(n) = 8T(\frac{n}{2}) + \Theta(n^2)$$

que pel Teorema Mestre resulta en

$$T(n) = n^3$$

Per tant, és un algorisme cúbic, igual que el producte de matrius tradicional. No hem guanyat res, i això es deu a que fem 8 productes de matrius més petites. Caldria reduir aquest nombre a 7, i ho podem fer mitjançant l'algorisme de Strassen:

$$X \cdot Y = \begin{pmatrix} P_5 + P_4 - P_2 + P_6 & P_1 + P_2 \\ P_3 + P_4 & P_1 + P_5 - P_3 - P_2 \end{pmatrix}$$

on les matrius $P_i$ són:

$P_1 = A(F - H)$

$P_2 = (A + B)H$

$P_3 = (C + D)E$

$P_4 = D(G - E)$

$P_5 = (A + D)(E + H)$

$P_6 = (B - D)(G - H)$

$P_7 = (A - C)(E + F)$

Amb aquest algorisme, el cost és $T(n) = 7T(\frac{n}{2}) + \Theta(n^2) = \Theta(n^{\log_2(7)}) \approx \Theta(n^{2.81})$.

## 5.3 Quick-Sort

Així com en el Merge-Sort, dividíem els vectors i ordenàvem els subvectors, de manera que només calia fusionar vectors ordenats, en el Quick-Sort fem el contrari: separem el vector original en dos vectors, de manera que tots els elements del primer són més petits o iguals que tots els elements del segon. Així, una vegada fetes totes les crides recursives, només cal concatenar els vectors resultants i obtindrem el vector ordenat.

Una qüestió important és com separem els elements del primer vector en dos subvectors. La manera ideal seria trobar la mediana, i separar segons si són més petits o més grans que la mediana, perquè així sempre tindríem la meitat dels elements en cada subvector, que és òptim. Però trobar la mediana és costós, i no podem fer-ho mantenint el cost de $\Theta(n \log n)$ desitjat. Per tant, triarem com a pivot el primer element del vector a ordenar.

Aquest algorisme té com a cost mitjà $\Theta(n \log n)$, però hi ha certes ordenacions inicials que fan que el pitjor cas sigui de $\Theta(n^2)$. Per exemple, en el cas de triar com a pivot el primer element, si el vector està ordenat des del principi l'algorisme tindrà un cost quadràtic. Molts llenguatges de programació tenen en la llibreria estàndard el quick-sort com a algorisme d'ordenació, però prenen mesures per detectar quan és probable que tingui cost quadràtic i canvien a merge-sort si és el cas. En el cas millor, l'algorisme és $\Theta(n \log n)$, que és el cas en què el pivot és la mediana cada vegada.

### Algorisme

```
1. Triar com a pivot el primer element del vector
2. Inicialitzar index i a la posicio del primer element - 1
3. Inicialitzar index j a la posicio de l'ultim element + 1
4. Mentre v[++i] sigui mes petit que p, no fer res
5. Mentre v[--j] sigui mes gran que p, no fer res
6. Si i < j, intercanviar v[i] per v[j]
7. Si no, cridar recursivament la funcio sobre els intervals [inici,
    j] i [j+1, final]
```

# 6 Sessió 6: Estructures de dades de la STL

En aquesta sessió veurem algunes estructures de dades de la Standard Library de C++ i el seu funcionament bàsic.

## 6.1 Stack

A la llibreria `<stack>`, hi trobem l'estructura de dades coneguda com a pila. Aquesta estructura és de tipus LIFO (Last In First Out), és a dir, funciona com una pila de papers: l'últim element en entrar és el primer en sortir. Direm que l'element més recent de la pila està "a dalt".

| | | |
|---|---|---|
| **Construcció** d'una pila d'elements del tipus `T` | `stack<T> s` | $O(1)$ |
| **Està buida?** | `s.empty()` | $O(1)$ |
| **Nombre d'elements** | `s.size()` | $O(1)$ |
| **Inserció** d'un element `x` de tipus `T` | `s.push(x)` | $O(1)$ |
| **Consulta** de l'element de dalt de tot | `s.top()` | $O(1)$ |
| **Esborrat** de l'element de dalt de tot | `s.pop()` | $O(1)$ |

## 6.2 Queue

A la llibreria `<queue>`, hi trobem l'estructura de dades coneguda com a cua. Aquesta estructura és de tipus FIFO (First In First Out), és a dir, funciona com una cua en la realitat: el primer element en entrar és el primer en sortir. Direm que l'element més antic de la cua està "al davant".

| | | |
|---|---|---|
| **Construcció** d'una cua d'elements del tipus `T` | `queue<T> q` | $O(1)$ |
| **Està buida?** | `q.empty()` | $O(1)$ |
| **Nombre d'elements** | `q.size()` | $O(1)$ |
| **Inserció** d'un element `x` de tipus `T` | `q.push(x)` | $O(1)$ |
| **Consulta** de l'element de davant de tot | `q.front()` | $O(1)$ |
| **Consulta** de l'element de darrere de tot | `q.back()` | $O(1)$ |
| **Esborrat** de l'element de dalt de tot | `q.pop()` | $O(1)$ |

## 6.3 Set

A la llibreria `<set>`, hi trobem l'estructura de dades coneguda com a conjunt. Aquesta estructura conté elements ("claus") d'un tipus `T`, i permet fer les operacions d'introduir i esborrar claus i de buscar claus al conjunt eficientment. És una estructura de dades ordenada i es pot recórrer. Internament, fan servir arbres binaris de cerca equilibrats (es veuen a les sessions següents).

| | | |
|---|---|---|
| **Construcció** d'un conjunt de claus del tipus `T` | `set<T> s` | $O(1)$ |
| **Està buit?** | `s.empty()` | $O(1)$ |
| **Nombre d'elements** | `s.size()` | $O(1)$ |
| **Inserció** d'un element `x` de tipus `T` | `s.insert(x)` | $O(\log n)$ |
| **Cerca** d'un element `x` de tipus `T` | `s.find(x)` | $O(\log n)$ |
| **Esborrat** d'un element `x` de tipus `T` | `s.erase(x)` | $O(\log n)$ |

**Observació.** La funció `find()` retorna un iterador. Si l'element no és al conjunt, retorna `s.end()`. Com a alternativa, la funció `count()` retorna 1 si l'element és al conjunt i 0 si no.

**Exemple** de recorregut d'un conjunt:

```
1 for (set<int>::iterator i = s.begin(); i != s.end(); i++) {
2     cout << *i << endl;
3 }
```

**Exemple** alternatiu a l'anterior (requereix C++11):

```
1 for (int x : s) {
2     cout << x << endl;
3 }
```

A la llibreria `<unordered_set>` hi ha una estructura de dades, els `unordered_set`, que es comporten igual que un conjunt però permeten inserció, esborrat i consulta en $O(1)$ a canvi de perdre l'ordenació. Aquests conjunts no es poden recórrer. Internament, fan servir *hash tables*, que es veuen a la sessió següent.

## 6.4 Map

A la llibreria `<map>`, hi trobem l'estructura de dades coneguda com a diccionari. que conté parelles `(clau, valor)` (que poden ser de tipus diferents) de manera que, sabent una clau, es pot obtenir el seu valor associat eficientment. Estan implementats amb arbres binaris de cerca, igual que els conjunts, però cada node de l'arbre passa a ser una parella.

| | | |
|---|---|---|
| **Construcció** d'un diccionari de tipus `(T1, T2)` | `map<T1, T2> m` | $O(1)$ |
| **Està buit?** | `m.empty()` | $O(1)$ |
| **Nombre d'elements** | `m.size()` | $O(1)$ |
| **Inserció** d'un `pair<T1, T2> p` | `m.insert(p)` | $O(\log n)$ |
| **Cerca** d'un parell amb clau `x` | `m.find(x)` | $O(\log n)$ |
| **Esborrat** d'un parell amb clau `x` | `m.erase(x)` | $O(\log n)$ |

**Observació.** La funció `find()` retorna un iterador. Si l'element no és clau de cap parell del diccionari, retorna `m.end()`. Com a alternativa, la funció `count()` retorna 1 si l'element és una clau del diccionari i 0 si no.

**Exemple** de recorregut d'un diccionari:

```
1 for (map<string, int>::iterator i = m.begin(); i != m.end(); i++) {
2     cout << i->first << " " << i->second << endl;
3 }
```

**Exemple** alternatiu a l'anterior (requereix C++11):

```
1 for (pair<string, int> p : m) {
2     cout << p.first << " " << p.second << endl;
3 }
```

A la llibreria `<unordered_map>` hi ha una estructura de dades, els `unordered_map`, que es comporten igual que un diccionari però permeten inserció, esborrat i consulta en $O(1)$ a canvi de perdre l'ordenació. Aquests diccionaris no es poden recórrer. Internament, fan servir *hash tables*.

## 6.5 Priority Queue

A la llibreria `<queue>`, a més de la cua, hi trobem les cues de prioritat. Aquestes estructures, que estan implementades internament amb *heaps* (es veuen a la sessió següent), permeten accedir a "l'element més gran" (depèn de com estigui definit l'operador $<$ en la classe), inserir i esborrar elements de manera eficient.

| | | |
|---|---|---|
| **Construcció** d'una cua de prioritat de tipus `T` | `priority_queue<T> q` | $O(1)$ |
| **Està buida?** | `q.empty()` | $O(1)$ |
| **Nombre d'elements** | `q.size()` | $O(1)$ |
| **Inserció** d'un element `x` de tipus `T` | `q.push(x)` | $O(\log n)$ |
| **Consulta** de l'element més gran | `q.top()` | $O(1)$ |
| **Esborrat** de l'element més gran | `q.pop()` | $O(\log n)$ |

# 7 Sessió 7: Heaps i hash tables

## 7.1 Heaps

**Definició.** Un *min-heap* (respectivament *max-heap*) és un arbre binari gairebé complet (tots els nivells estan complets excepte potser l'últim) que compleix que per cada node, el valor que conté és menor o igual al dels seus dos fills (respectivament, major o igual).

**Observació**. En un min-heap, l'arrel és l'element més petit. En un max-heap, és el més gran.

A partir d'ara parlarem només de max-heaps. Els min-heaps funcionen anàlogament.

Les operacions que permet fer un max-heap són:

1. Consultar el valor màxim, en temps constant.
2. Eliminar el valor màxim, en temps $O(\log n)$.
3. Inserir un element, en temps $O(\log n)$.

Representarem un heap d'altura $h$ en un vector $v$ de mida $2^h - 1$. Els fills de $v[i]$ (indexat des de 0) seran $v[2 * i + 1]$ i $v[2 * i + 2]$.

L'operació de consultar el valor màxim és fàcil, només cal retornar $v[0]$.

L'operació d'eliminar el valor màxim té el problema que, una vegada eliminat, ens queden dos arbres binaris on cadascun d'ells compleix les propietats d'un heap, i volem unir-los en un de sol. Per això, una vegada eliminat $v[0]$, agafarem l'última posició no buida de $v$ i la mourem a $v[0]$. Aleshores, ens quedarà un arbre binari que no és un heap.

Per arreglar-ho, crearem una funció anomenat `heapify`, que rebrà com a paràmetre una posició del vector, representant un node de l'arbre binari. Aquesta funció mirarà si el node és més gran que els seus dos fills. Si no ho és, intercanviarà el node pel més gran dels seus fills i cridarà la funció recursivament sobre aquell fill. D'aquesta manera, amb un màxim de $\log n$ crides (una per cada nivell de l'arbre) haurem convertit l'arbre binari en un heap.

Per últim, l'operació d'afegir un element té el problema que es perd la propietat del heap. Per arreglar-ho, farem servir un mètode semblant al de `heapify`, però en el sentit contrari. Guardarem el nou element a la primera posició buida del vector, i comprovarem recursivament si és més gran que el seu pare, intercanviant-los si és necessari. Com abans, com a molt tindrem $\log n$ crides recursives, si s'arriba fins a l'arrel.

Un dels usos dels heaps és la implementació de la priority queue. De fet, les tres operacions que permet el nostre heap són exactament les operacions bàsiques d'una priority queue: `top()`, `pop()` i `push()`.

## 7.2 Hash tables

L'objectiu d'aquest apartat és tenir una manera eficient d'emmagatzemar, inserir i esborrar nombres de mida arbitrària. Per això, farem servir funcions i taules de hash.

**Definició.** Una funció de hash és qualsevol funció que es pugui calcular en $\Theta(1)$, que prengui *inputs* de mida arbitrària i retorni un nombre d'un rang fixat $[0, ..., M)$.

Un exemple de funció de hash és el mòdul: $\text{mod}(x, M)$ retorna sempre un nombre de $[0, M-1]$.

**Definició.** Una taula de hash és una vector de vectors de longitud fixada $M$, on cadascun dels seus vectors pot tenir longituds diferents.

Si tenim un conjunt de $n$ enters ("claus") de mida arbitrària, aplicarem una funció de hash sobre ells i farem servir el resultat com a índex per guardar-los en una taula de hash. Així, podem inserir elements en temps constant.

Per consultar si un element es troba a la taula, només cal aplicar-li la funció de hash i buscar-lo a la posició corresponent. El temps que es triga en buscar-lo és proporcional a la quantitat d'elements que estan guardats en aquell índex, que serà aproximadament $\frac{n}{M}$ si la taula està equilibrada. Anomenem aquest valor $\alpha = \frac{n}{M}$ el factor de càrrega de la taula. De mitjana, una inserció tindrà cost $\Theta(1 + \alpha)$, i un esborrat tindrà el mateix cost.

**Observació.** Existeix la possibilitat que molts elements vagin a parar al mateix índex: com a

exemple, si la funció de hash és $\text{mod}(x, 2)$ i tots els nombres del conjunt són parells, tots aniran a parar a la posició 0. Per això, una bona funció de hash és aquella que enviï aproximadament la mateixa quantitat d'elements a cada índex de la taula.

# 8 Sessió 8: Grafs i arbres

**Definició.** Un graf és un conjunt $V$ de vèrtexs, juntament amb un conjunt $E \subseteq V \times V$ d'arestes entre els vèrtexs. Escriurem $G = (V, E)$ per denotar que $G$ és un graf format per $V$ i $E$. Com a notació, farem servir $n = |V|, m = |E|$.

Els grafs que considerarem seran sempre simples i sense llaços, és a dir, entre dos vèrtexs diferents hi ha com a molt una aresta, i no hi ha cap aresta que uneixi un node amb si mateix.

Un graf és no dirigit si per tot $x, y \in V, (x, y) \in E \iff (y, x) \in E$. Si no, diem que és dirigit.

Les dues representacions més habituals d'un graf en un programa són:

- Matriu d'adjacència: Una matriu A de mida $n \times n$, on l'entrada $A_{ij}$ val 1 si l'aresta entre els vèrtexs $i$ i $j$ existeix, i 0 si no.
- Llistes d'adjacència: Un vector de mida n, on la component $i$-èsima és un vector que conté tots els vèrtexs adjacents a $i$ (aquests vectors poden ser de mides diferents per diferents $i$).

**Definició.** Diem que un graf és un arbre quan compleix qualsevol de les següents caracteritzacions equivalents:

- És connex i $m = n - 1$
- És connex i acíclic
- És acíclic i $m = n - 1$

## 8.1 Spanning trees

**Definició.** Donat un graf connex no dirigit $G = (V, E)$, un spanning tree de $G$ és un graf $G' = (V, E')$ tal que $G'$ és un arbre i $E' \subset E$.

Si tenim un graf connex i volem trobar-ne un spanning tree, l'algorisme que seguirem és el següent:

```
1. Ordenar les arestes d'alguna manera
2. Per cada aresta,
    2.1. Si connecta dos vertexs no connectats, l'agafem
    2.2. Si no, no l'agafem
    2.3. Si hem agafat n-1 arestes, hem acabat
```

Vist així, no queda del tot clar com fer l'operació "saber si dos vèrtexs estan connectats" de manera eficient. Per això, farem servir una estructura de dades anomenada MF-set (també Merge-Find, o Union-Find). Com el seu nom indica, aquesta estructura de dades ens permetrà fer dues operacions eficientment: unir dos conjunts, i saber a quin conjunt pertany un determinat element.

El funcionament de MF-set és el següent:

Tenim un vector `pare` de $n$ elements, on l'$i$-èsim element és el representant de la component connexa de la qual forma part el vèrtex $i$. Inicialment, com que no hi ha cap aresta, totes les posicions del vector estan a $-1$, que indica que cada vèrtex és el representant de la seva component connexa.

Si volem trobar el representant d'un vèrtex, caldria fer `pare[x]`, `pare[pare[x]]`, ..., fins que el resultat sigui $-1$. Això ens pot donar un nombre lineal d'iteracions, per exemple en el cas on cada vèrtex és el pare de l'anterior, formant una línia. Tot i així, podem aprofitar aquestes iteracions per escurçar el camí des de cada vèrtex fins al seu representant, de manera que el següent càlcul sigui molt més ràpid. Per exemple, aquesta és una implementació de `find` que ho fa.

```
int find(int x) {
    if (pare[x] == -1) return x; //x es el seu propi representant
    else {
        int y = find(pare[x]);
        pare[x] = y;            //aixi escurcem el cami

```

```
7           return y;
8       }
9   }
```

O equivalentment, però amb un codi més concís,

```
1 int find(int x) {
2     return (pare[x] == -1 ? x : pare[x] = find(pare[x]));
3 }
```

Si volem unir dos vèrtexs $x$ i $y$, les seves components connexes s'uniran. En aquest cas, el que farem serà trobar els representants de $x$ i de $y$, $rx$ i $ry$ respectivament, i fer `pare[rx] = ry`. [1]

En el cas particular de l'algorisme que estem plantejant, per saber si dos vèrtexs $x$ i $y$ estan connectats n'hi ha prou amb fer `find(x) == find(y)`.

Amb això, queda explicat el funcionament de MF-set, i es pot resoldre eficientment el problema de trobar un Spanning Tree d'un graf.

### 8.1.1 Minimum Spanning Tree

**Definició.** Donat un graf connex no dirigit $G$ amb pesos a les arestes, el minimum spanning tree (MST) de $G$ és el spanning tree de $G$ amb la suma de pesos de les arestes escollides més petita.

El MST d'un graf pot no ser únic, si dues arestes tenen el mateix pes.

Un algorisme per trobar un MST d'un graf és el de Kruskal. Aquest algorisme és igual al que hem descrit a l'apartat anterior, però amb la diferència que al pas 1, les arestes s'ordenen creixentment per pesos.

[1] Es podria fer una mica més eficientment aquesta operació si ens guardéssim la profunditat dels arbres que representen les components connexes de $rx$ i $ry$, i decidíssim quin dels dos ordres possibles és millor, però el codi donat és també prou eficient.

## 8.2 Binary Search Tree i AVL Tree

**Definició.** Un arbre binari és aquell on cada vèrtex té exactament dos fills, que poden ser l'arbre buit.

**Observació.** Dins d'aquesta secció, tractarem els arbres com a estructures de dades, i direm que el "valor" d'un vèrtex és el valor de la dada que emmagatzema.

**Definició.** Un arbre binari de cerca, o binary search tree (BST), és un arbre binari que compleix que per cada vèrtex, el valor del vèrtex és major o igual a tots els valors dels vèrtexs del seu fill esquerre, i menor o igual a tots els valors dels vèrtexs del seu fill dret.

**Definició.** El factor d'equilibri d'un vèrtex $v$ d'un arbre binari, si $L$ és el fill de l'esquerra de $V$ i $R$ és el fill de la dreta de $V$, és equilibri$(v)$ = altura$(R)$ − altura$(L)$. Informalment, si aquest factor és petit per tots els vèrtexs de l'arbre, direm que és un arbre equilibrat. Altrament, direm que és desequilibrat.

Els arbres binaris de cerca, si estan equilibrats, fan que sigui molt eficient l'operació de cercar un valor en l'arbre, ja que a cada iteració es pot descartar la meitat de l'arbre, fent que el cost sigui logarítmic. Si no estan equilibrats, però, en el pitjor cas el cost és lineal (un arbre binari completament desequilibrat és simplement una llista ordenada). Per això, fem servir el concepte de AVL trees, definits a continuació.

**Definició.** Un arbre AVL, o AVL tree, és un BST que compleix que per cada vèrtex $v \in V$, equilibri$(v) \in \{-1, 0, 1\}$. És a dir, que la diferència d'altures entre els fills de qualsevol vèrtex de l'arbre és com a molt 1 en valor absolut.

**Teorema.** L'alçada d'un arbre AVL de $n$ nodes és $\Theta(\log n)$.

*Demostració.* Sigui $h$ l'altura de l'arbre. Veurem primer que $h \in \Omega(\log n)$.

En un arbre complet d'altura $h$, hi ha $2^0 + 2^1 + ... + 2^{h-1} = 2^h - 1$ vèrtexs. Per tant, en general, $n \leq 2^h - 1$, i per tant $\log(n + 1) \leq h$. Així, $h \in \Omega(\log n)$.

Vegem ara que $h \in O(\log n)$.

Sigui $N_h$ el mínim nombre de vèrtexs que té un arbre AVL d'altura $h$. Llavors, es compleix $N_h = 1 + N_{h-1} + N_{h-2}$, perquè com a mínim cal un vèrtex d'arrel, un arbre d'alçada $h-1$ en un costat i un d'alçada $h-2$ a l'altre per no violar la condició d'equilibri.

Si $f_h$ és el terme $h$-èsim de la successió de Fibonacci, que ve definida per $f_h = f_{h-1} + f_{h-2}$, és clar que $N_h > f_h$, i $f_h > 2^{\frac{h}{2}}$ com hem vist en altres sessions. Així, $2 \log N_h \geq h$, i $h \in O(\log n)$. □

### 8.2.1 Insercions en un AVL

Aquest apartat està explicat en aquest link, ja que pels dibuixos no és fàcil d'explicar per text.

# 9 Sessió 9: Indecidibilitat

**Definició.** Un problema computacional està format per un conjunt d'entrades possibles $E$, un conjunt de sortides possibles $S$, una relació $R \subseteq E \times S$, i una funció de talla $|.|$ que mesura la mida de l'entrada. La relació $R$ determina quines sortides ha de produir cada entrada.

**Definició.** Un problema decisional està format per un conjunt d'entrades possibles $E$, un subconjunt $L \subseteq E$ d'entrades positives, i una funció de talla $|.|$. El problema tracta de determinar si una $x \in E$ donada pertany o no a $L$.

**Definició.** Direm que un problema decisional és decidible si existeix un algorisme que el resolgui (no es té en compte el cost de l'algorisme).

## 9.1 Halting problem

El problema de l'aturada, o halting problem, consisteix en determinar, donat un programa $p$ i una entrada $x$, si $p$ s'atura amb entrada $x$ o no, és a dir, si no entra en cap bucle infinit.

**Teorema.** El halting problem és indecidible.

*Demostració.* Ho provarem per contradicció. Suposarem que existeix una funció `bool atura(string P, string x)` que retorna `true` si i només si la funció $p$ s'atura amb l'entrada $x$.

Suposem que tenim una funció `void penja(string p)` que entra en un bucle infinit, i una funció `void no_pengis(string p)` que no fa res. Farem servir `atura` per implementar una altra funció:

```
1 void contradiccio(string p) {
2     if (atura(p, p)) penja(p);
3     else no_pengis(p);
4 }
```

Aleshores, si cridem `contradiccio(contradiccio)`, pot ser que s'aturi o que no s'aturi.

Si s'atura, aleshores `atura(contradiccio, contradiccio)` és cert, però llavors s'executa `penja(contradiccio)` i no s'atura.

Si no s'atura, aleshores `atura(contradiccio, contradiccio)` és fals, però llavors s'executa `no_pengis(contradiccio)` i s'atura.

Per tant, en qualsevol dels dos casos es produeix una contradicció, i aquesta s'ha originat quan hem assumit que la funció `atura` podia existir. □

## 9.2 Totalitat

El problema de la totalitat consisteix a, donats un programa $p$, determinar si aquest programa s'atura per tota entrada $x$.

**Teorema.** Totalitat és indecidible.

*Demostració.* Suposem que existeix una funció `bool totalitat(string p)` que retorna cert si i només si el programa $p$ s'atura per totes les entrades.

Definim la funció `string transforma(string p, string x)`, de manera que donat un programa $p$ i una entrada $x$, retorna el programa $p'$, que és $p$ restringit a l'entrada $x$ (és a dir, que per tota entrada $y$, $p'(y)$ retorna $p(x)$). Aleshores, podem fer:

```
1 bool atura(string p, string x) {
2     string q = transforma(p, x);
3     return totalitat(q);
4 }
```

i hem resolt així el halting problem. Però abans hem vist que el halting problem és indecidible, i per tant hem arribat a una contradicció, que s'ha originat quan hem assumit que la funció `totalitat` podia existir. □

## 9.3 Equivalència

El problema de l'equivalència consisteix a, donats dos programes $p$ i $q$, determinar si per tota entrada $x$, $p(x) = q(x)$.

**Teorema.** Equivalència és indecidible.

*Demostració.* Suposem que existeix una funció `bool equivalencia(string p, string q)`, que retorna cert si i només si les funcions $p$ i $q$ coincideixen en totes les seves entrades.

Aleshores, per qualsevol funció $f$ i qualsevol entrada $x$, creem les dues funcions següents:

```
1 void first(string f, string x) {
2     f(x);
3     cout << "yes" << endl;
4 }
5 void second(string f, string x) {
6     cout << "yes" << endl;
7 }
```

Llavors, veiem que aplicar `equivalencia(first, second)` és el mateix que resoldre el halting problem. Com que aquest és indecidible, hem arribat a una contradicció, que s'ha originat quan hem assumit que la funció `equivalencia` podia existir. □

# 10 Sessió 10: P i NP

## 10.1 Definicions bàsiques

Com ja hem vist en la secció de Big O i Big Theta, resulta útil de classificar els problemes segons la seva "dificultat", és a dir, com d'eficient és la seva solució òptima (o la més eficient que coneguem). Les classes de problemes P i NP són una altra classificació de la dificultat d'un problema.

**Definició.** P és la classe de tots els problemes decisionals decidibles que es poden decidir en temps polinòmic determinista. És a dir, un problema $p$ és de P si existeix $k \in \mathbb{N}$ tal que el problema es pot resoldre en $O(n^k)$.

**Definició.** NP és la classe de tots els problemes decisionals decidibles que es poden decidir en temps polinòmic no determinista. Això és equivalent a que es puguin verificar en temps polinòmic: donada una instància del problema i una proposta de solució, es pot veure si la proposta és una solució vàlida del problema en temps polinòmic en la mida de l'entrada.

**Observació.** Trivialment, $P \subseteq NP$. La inclusió contrària és un problema obert.

**Exemples**

- **Graf hamiltonià.** Donat un graf $G$ de $n$ vèrtexs, determinar si $G$ conté un cicle hamiltonià (un cicle de longitud $n$ tal que cada vèrtex hi aparegui una vegada).

  Aquest problema és de NP perquè donades una instància del problema (un graf) i una proposta de solució (un cicle) es pot determinar en temps lineal si és un cicle vàlid, si té longitud $n$ i si passa per totes les arestes. En canvi, no sabem si és a P perquè no s'ha trobat cap algorisme polinòmic que sigui capaç de trobar un cicle hamiltonià en un graf arbitrari.

- **SAT.** Donada una fòrmula en forma normal conjuntiva amb $m$ clàusules i $n$ variables, com per exemple $(x_1 \vee x_2 \vee x_3) \wedge (\bar{x_1} \vee x_4) \wedge (x_2 \vee x_4)$, determinar si existeix una combinació $(x_1, ..., x_n) \in \{TRUE, FALSE\}^n$ tal que avaluï l'expressió a cert.

Aquest problema també és NP. En efecte, donats una instància del problema (una fòrmula) i una proposta de solució (una combinació de valors booleans) es pot determinar en temps polinòmic (en $n$ i $m$) si l'expressió avalua a cert, ja que només cal comprovar per cadascuna de les $m$ clàusules si alguna de les variables (de les quals n'hi ha $\leq n$) és certa en la combinació donada. Altra vegada, no es coneix cap algorisme que resolgui el problema SAT en temps polinòmic, així que no sabem si és a P.

**Observació.** Definim el problema $k$-SAT com una restricció de SAT on cada clàusula té exactament $k$ variables. 2-SAT és a P, però per tot $k \geq 3$, no sabem si $k$-SAT és a P. Clarament, tots són a NP, perquè SAT hi és.

## 10.2 NP-hard i NP-complet

**Definició.** Siguin $L_1$ i $L_2$ dos problemes decisionals. Diem que $L_1$ es pot reduir a $L_2$ en temps polinòmic, i escrivim $L_1 \leq_p L_2$, si existeix un algorisme polinòmic $A$ tal que transformi instàncies de $L_1$ en instàncies de $L_2$, i que per tota instància $I$ de $L_1$, $I$ és positiva per $L_1 \iff A(I)$ és positiva per $L_2$.

**Definició.** Diem que un problema $L$ és NP-hard si tots els problemes de NP es poden reduir a $L$ en temps polinòmic.

**Observació.** $L \in$ NP-hard no implica que $L \in$ NP. Això motiva la definició següent:

**Definició.** Diem que un problema $L$ és NP-complet si és a NP i a NP-hard. Això és, NP-complet = NP $\cap$ NP-hard.

**Teorema** (Cook). SAT és NP-complet.

(Demostració no donada a classe)

**Definició.** Definim el problema CLIQUE com: Donats un graf $G$ i un enter $k \geq 3$, determinar si existeix $G'$ subgraf de $G$ tal que $G'$ és el graf complet de mida $k$.

**Teorema.** CLIQUE és NP-complet.

*Demostració.* Vegem primer que CLIQUE és NP.

Donada una instància del problema (un graf $G$ i un enter $k$) i una proposta de solució (un subgraf $G'$ de $G$), es pot determinar en temps polinòmic si compleix les condicions del problema: en temps $\Theta(k)$ podem comprovar si tots els vèrtexs de $G'$ són a $G$ i si n'hi ha $k$, i en temps $\Theta(k^2)$ podem comprovar si totes les arestes entre parelles de vèrtexs de $G'$ són a $G$.

Ara hem de veure que CLIQUE és NP-hard. Com que disposem del Teorema de Cook, sabem que SAT és NP-hard, i per tant hem de veure que podem reduir SAT a CLIQUE en temps polinòmic i haurem acabat la demostració.

Tenim una instància del problema SAT i hem de generar una instància del problema CLIQUE, és a dir, hem de crear un graf i un enter $k$. Sigui $m$ el nombre de clàusules de la instància de SAT, triarem $k = m$. Aleshores, per cada clàusula, crearem un vèrtex per cada literal de la clàusula, i aquests seran els vèrtexs del nostre graf. Hi haurà una aresta entre dos vèrtexs si i només si no pertanyen a la mateixa clàusula i no es contradiuen entre ells (és a dir, no són de la forma $x_i$ i $\bar{x}_i$).

Aquesta instància de CLIQUE és equivalent a la instància de SAT inicial, ja que:

Si la instància de CLIQUE és positiva, aleshores existeix un subconjunt de $m$ literals tals que cap d'ells es contradiu amb cap altre i estan en $m$ clàusules diferents. Per tant, existeix una combinació de valors booleans que fa que cada clàusula resulti certa, i la instància de SAT és positiva.

Si la instància de CLIQUE és negativa, aleshores per totes les possibles tries de $m$ literals, un de cada clàusula, existeix almenys una contradicció entre dos d'ells, i per tant cap combinació de valors booleans satisfarà l'expressió. Per tant, la instància de SAT és negativa.

Aquest algorisme de reducció és clarament polinòmic ja que el graf tindrà com a molt $nm$ vèrtexs, i caldrà temps com a molt quadràtic per comprovar per cada parella de vèrtexs si existeix o no l'aresta que els uneix. □

**Teorema.** 3-SAT és NP-complet.

*Demostració.* Aquest problema és NP per ser un cas particular del problema SAT, que és NP.

Vegem que podem reduir una instància de SAT a una de 3-SAT:

Per cada clàusula de la instància original, aquesta tindrà la forma $c_i = (a_1 \vee a_2 \vee ... \vee a_k)$ per algun $k$ (no tenen per què ser tots iguals). La transforem en una conjunció $c_i'$ de clàusules de longitud 3 que seran equivalents a $c_i$, tal com definim a continuació:

$$c_i' = (a_1 \vee a_2 \vee y_1) \wedge (\bar{y_1} \vee a_3 \vee y_2) \wedge (\bar{y_2} \vee a_4 \vee y_3) \wedge ... \wedge (\bar{y_{k-3}} \vee a_{k-1} \vee a_k)$$

on definim les variables $y_j$ artificials com:

Si com a mínim algun $a_j$ val 1, prenem $j$ com el menor índex que valgui 1 i fem $y_k = 1$ per tot $k \leq j - 2$ i $y_k = 0$ per tot $k > j - 2$. Aquestes variables satisfan $c_i'$, i per tant si $c_i$ és satisfactible $c_i'$ també ho és.

En canvi, si tots els $a_j$ valen 0, no hi ha cap manera de triar els $y_k$ de manera que $c_i'$ sigui certa, i per tant si $c_i$ no és satisfactible, $c_i'$ tampoc.

Si fem això per cada clàusula de la instància de SAT, obtenim una instància equivalent de 3-SAT. Aquest algorisme és clarament polinòmic ja que només cal passar una vegada per cada literal de la instància de SAT (com a màxim n'hi ha $nm$) per crear la instància de 3-SAT equivalent. □

# 11 Sessió 11. Algorismes diversos

## 11.1 BFS i DFS

Donat un graf $G = (V, E)$, volem recórrer el graf començant des d'un vèrtex $v$ qualsevol. Per fer-ho, hi ha dues maneres freqüentment utilitzades: la cerca en amplada (Breadth-First Search, BFS) i la cerca en profunditat (Depth-First Search, DFS).

### 11.1.1 BFS

Aquest algorisme visita els vèrtexs en ordre de distància del vèrtex inicial.

La idea és de posar a la cua el vèrtex inicial, i mentre la cua no estigui buida, posar tots els veïns no visitats a la cua i marcar-los com a visitats. Suposant que tenim els veïns de cada vèrtex guardats en forma de llistes d'adjacència en la matriu $G$, i que volem fer l'operació `visita()` sobre tots els vèrtexs del graf, el codi queda així:

```
1  void bfs(int inicial) {
2      queue<int> Q;
3      vector<bool> used(n, false);
4      Q.push(inicial);
5      used[inicial] = true;
6      while (not Q.empty()) {
7          int v = Q.front();
8          Q.pop();
9          visita(v);
10         for (int i = 0; i < (int)G[v].size(); i++) {
11             int w = G[v][i];
12             if (not used[w) {
13                 Q.push(w);
14                 used[w] = true;
15             }
16         }
17     }
```

```
18 }
```

### 11.1.2 DFS

Aquest algorisme explora fins tan lluny com pugui del vèrtex inicial, i després torna enrere fins que pot tornar a avançar.

El codi és molt semblant al del BFS, però amb una pila enlloc d'una cua.

```
1  void dfs(int inicial) {
2      stack<int> S;
3      vector<bool> used(n, false);
4      S.push(inicial);
5      used[inicial] = true;
6      while (not S.empty()) {
7          int v = S.front();
8          S.pop();
9          visita(v);
10         for (int i = 0; i < (int)G[v].size(); i++) {
11             int w = G[v][i];
12             if (not used[w]) {
13                 S.push(w);
14                 used[w] = true;
15             }
16         }
17     }
18 }
```

## 11.2 Dijkstra

Donat un graf amb pesos a les arestes, un vèrtex inicial i un vèrtex final, volem trobar el camí de menys cost entre aquests dos vèrtexs. Per això, farem servir una cua de prioritat (amb l'element més petit a dalt) que guardarà el cost acumulat de cada camí que hem calculat fins

al moment. La idea és de provar d'afegir al camí actual totes les arestes des del vèrtex actual, i inserir-les a la cua de prioritat, que ens donarà en cada moment el camí de cost mínim dels que hem calculat.

Suposem que tenim una constant `INF` prou gran, que tenim els veïns de cada vèrtex en llistes d'adjacència a la matriu $G$, i que començant a $x$ volem arribar a $v$. El codi d'aquest algorisme és el següent.

```
int dijkstra(int x, int v) {
  priority_queue<pair<int, int>> Q;
  vector<int> dist(n, INF);
  Q.push(pair<int, int>(0, x));
  dist[x] = 0;
  while (not Q.empty()) {
    pair<int, int> a = Q.top(); Q.pop();
    int d = -a.first;
    int y = a.second;
    if (d == dist[y]) {
      if (y == v) return d;
      for (pair<int, int> arc : G[y]) {
        int c = arc.first;
        int z = arc.second;
        int d2 = d + c;
        if (d2 < dist[z]) {
          dist[z] = d2;
          Q.push(pair<int, int>(-d2, z));
        }
      }
    }
  }
  return INF;
}
```

**Observació.** Com que per defecte la cua de prioritat de C++ té el màxim a dalt, podem inserir els costos canviats de signe per tal d'invertir l'ordenació.

## 11.3 Max-flow

Donats un graf dirigit connex $G$ amb pesos a les arestes, on cada pes representa la capacitat d'aquella aresta, i dos vèrtexs especials anomenats *source* (origen, representat per $s$) i *target* (objectiu, representat per $t$), volem saber quina és la màxima quantitat de flux que es pot fer sortir de $s$ i arribar a $t$ de manera que respecti les capacitats de les arestes i que hi hagi un equilibri de flux a cada vèrtex excepte a $s$ i $t$.

L'algorisme que farem servir pot fer un BFS (Edward-Karp) o bé un DFS (Ford-Fulkerson). El nostre farà servir BFS perquè està demostrat que és més eficient. Si el graf té $m$ arestes i $n$ vèrtexs, el cost amb BFS és de $O(e^2n)$, mentre que amb DFS el cost és $O(e \cdot \text{maxflow})$, que depèn del valor de la funció objectiu maxflow i pot arribar a ser molt ineficient.

La idea de l'algorisme és de trobar un camí per on encara pugui passar flux, repetidament, fins que ja no es pugui. Per fer això, ens guardarem en una matriu la capacitat *actual* de cada aresta, és a dir, tenint en compte el flux que s'hi ha fet passar fins ara. A més, farem que sigui possible que el flux "torni enrere", ja que si fem passar $k$ unitats de flux en l'aresta $xy$, li sumarem $k$ unitats de capacitat a l'aresta (possiblement inexistent) $yx$, per poder fer-ho.

El codi complet es pot trobar a Atenea.

## 11.4 Min-cut

Donats un graf dirigit connex $G = (V, E)$ amb pesos a les arestes, i dos vèrtexs especials $s$ i $t$, definim un *tall* com un conjunt d'arestes $S$ tal que:

- El graf $G' = (V, E \setminus S)$ té dues components connexes $C_1$ i $C_2$.
- $s \in C_1$, $t \in C_2$.

El problema del Min-cut ens demana que trobem el tall de mínim cost, és a dir, tal que la suma dels pesos de totes les arestes de $S$ sigui mínima.

Aquest problema està molt relacionat amb el Max-flow. De fet, és el seu dual. Es compleix

que per tot flux $f$ que es pugui passar pel graf, i per tot tall $c$ que es pugui fer al graf, $f \leq c$. A més, es compleix que $\text{maxflow}(G) = \text{mincut}(G)$ per tot graf $G$ connex.

Per resoldre aquest problema, podem fer servir l'algorisme del Max-flow modificat de manera que vagi iterant fins que no trobi cap manera de fer passar més flux, i llavors es guardi les arestes saturades que han impedit el pas del flux. Aquestes arestes formen un tall, i de fet formen el tall mínim del graf.

## 11.5 Matching màxim en un graf bipartit

Una aplicació del Max-flow és la següent:

Donat un graf bipartit no dirigit, trobar el nombre màxim d'arestes $(a, b)$, sense cap $a$ ni cap $b$ repetits, entre les dues parts del graf. Això és equivalent a trobar el màxim nombre de "parelles compatibles" possible, on cada vèrtex representa una persona, cada persona pot tenir com a màxim una parella, i les arestes adjacents a una persona indiquen les persones amb qui és compatible.

Podem transformar aquest problema en un Max-flow fàcilment afegint un vèrtex $s$ i unint-lo amb tots els vèrtexs d'una part del graf, i un vèrtex $t$ a la dreta i unint-lo amb tots els vèrtexs de l'altra part del graf. Si assignem a cada aresta capacitat 1, en aplicar Max-flow sobre aquest graf, obtenim el nombre màxim de parelles possibles.

## 11.6 Cerca de subparaules en un text, versió 1

Donat un `string s`, de mida $n$, i un `string x`, de mida $m \leq n$, volem saber si $x$ és una subparaula de $s$. La manera trivial de fer-ho seria d'anar iterant des de $i = 0$ fins a $n - m$ i veure amb un bucle si la subparaula de $m$ lletres que comença a $s[i]$ és efectivament $x$. Això, en el pitjor cas ens costa $O(mn)$. Volem reduir aquest cost a $O(m + n)$.

Per fer-ho, fent servir el sistema de numeració de base 26, podem assignar a cada subparaula de mida $m$ de $s$ un nombre, de manera que la subparaula que comença al caràcter $k$ té valor

$$\sum_{i=0}^{m-1}(s[k+i] - `a') \times 26^{m-i}$$

És a dir, cada lletra de la subparaula funciona com un dígit en base 26, on 'a' actua com a zero i 'z' actua com a 25. Aquest procediment assigna un valor únic a cada paraula, però té l'inconvenient que es fa molt gran molt ràpidament. Per això, després de cada operació apliquem l'operació mòdul un nombre primer prou gran per evitar col·lisions.

L'avantatge d'aquest mètode és que, una vegada calculat el valor d'una subparaula, afegir una lletra al final i eliminar-ne una al principi permet recalcular el valor de la nova paraula en temps constant. En efecte, només cal restar el pes de la lletra que eliminarem, multiplicar el valor resultant per 26, i sumar el valor de la nova lletra que afegim, i aquest nombre d'operacions no depèn de $m$.

Hi ha un problema: podria haver-hi falsos positius, ja que en aplicar l'operació mòdul, dos paraules diferents podrien resultar en el mateix valor. Per això diem que aquest mètode és probabilístic, tot i que podem reduir la probabilitat de col·lisió tant com vulguem, augmentant el nombre primer pel qual fem mòdul.

## 11.7 Cerca de subparaules en un text, versió 2 (algorisme KMP)

En aquest algorisme, com en l'anterior, farem una cerca d'una subparaula en un text en temps lineal en la longitud del text, però en aquest cas no hi haurà un factor de probabilitat. Abans de començar amb l'algorisme, cal introduir algunes definicions.

**Definició.** Prefix d'una paraula: subparaula que té com a caràcter inicial el primer caràcter de la paraula.

**Definició.** Sufix d'una paraula: subparaula que té com a caràcter final l'últim caràcter de la paraula.

**Definició.** Frontera d'una paraula: subparaula diferent de la paraula completa que és a la

vegada un prefix i un sufix de la paraula.

**Proposició.** Sigui $y$ una frontera de $z$. Aleshores, si $x$ és una frontera de $y$, $x$ és una frontera de $z$. Això es pot veure com que la propietat de ser frontera és transitiva.

*Demostració.* Si $x$ és frontera de $y$, aleshores és prefix de $y$, i com que $y$ és prefix de $z$, $x$ és prefix de $z$.

Si $x$ és frontera de $y$, aleshores és sufix de $y$, i com que $y$ és sufix de $z$, $x$ és sufix de $z$.

Per tant, $x$ és frontera de $z$. □

Aleshores, si volem cercar la paraula $x$ en $s$, per cada prefix $s_i$ de $s$, li assignem la longitud de la frontera més llarga de $s_i$ que acaba en el caràcter $s[i]$. Guardarem aquest resultat en un vector anomenat $W$, de la mateixa mida que $s$.

**Exemple.** Sigui $s$ la paraula `abcabdabcab`. Aleshores,

- A la subparaula `a`, el caràcter $a$ és la subparaula $s_0$ completa, i per tant no és frontera de $s_0$. $W[0] = 0$.
- A la subparaula `ab`, no hi ha cap prefix que també sigui un sufix. $W[1] = 0$.
- A la subparaula `abc`, no hi ha cap prefix que també sigui un sufix. $W[2] = 0$.
- A la subparaula `abca`, el prefix $a$ és una frontera. $W[3] = 1$.
- A la subparaula `abcab`, el prefix $ab$ és una frontera. $W[4] = 2$.
- A la subparaula `abcabd`, no hi ha cap prefix que també sigui un sufix. $W[5] = 0$.
- A la subparaula `abcabda`, el prefix $a$ és una frontera. $W[6] = 1$.
- A la subparaula `abcabdab`, el prefix $ab$ és una frontera. $W[7] = 2$.
- A la subparaula `abcabdabc`, el prefix $abc$ és una frontera. $W[8] = 3$.
- A la subparaula `abcabdabca`, el prefix $abca$ és una frontera. $W[9] = 4$.
- A la subparaula `abcabdabcab`, el prefix $abcab$ és una frontera. $W[10] = 5$.

Fent servir propietats com la transitivitat de la frontera i que el valor de $W[i+1] \leq W[i]+1$, es pot calcular $W$ en temps lineal.

Per aplicar això a la cerca de la subparaula $x$ en el text $s$, només cal generar la paraula $s' = x\#s$, on concatenem $x$ amb un caràcter de separació (en aquest cas $\#$) i amb $s$, i aplicar l'algorisme anterior a $s'$. Després, busquem $m$ (la mida de $x$) en el vector $W$ obtingut. Per cada vegada que aparegui $m$ en $W$ tindrem una aparició de $x$ en $s$. Aquest procediment també triga temps lineal, i per tant l'algorisme és, en global, lineal.

# 12 Sessió 12. Teoria de jocs

**Definició.** Joc del Nim. Tenim dos jugadors i unes quantes piles de pedres. Cada torn, el jugador corresponent tria una pila i un nombre diferent de zero de pedres, i les treu de la pila. Si un jugador es troba sense pedres a cap pila, ha perdut.

En aquesta sessió analitzarem el joc del Nim, però els resultats obtinguts són de fet vàlids per a tot joc imparcial, definit a continuació.

**Definició.** Diem que un joc és imparcial quan compleix totes les característiques següents:

- Dos jugadors
- Per torns
- Amb informació completa
- Sense atzar
- Sense peces pròpies de cap jugador
- Sense possibilitat d'empats
- Finit i sense bucles
- Qui no pot jugar, perd

A partir d'aquest punt, es referirà només al Nim, però els resultats valen per qualsevol joc imparcial.

**Definició.** Sigui $X$ una posició. Escriurem $g(X)$ per voler dir "la posició $X$ és guanyadora" i $\bar{g}(X)$ per dir "la posició $X$ és perdedora". Es defineix la posició buida, $\varnothing$, com a perdedora. A partir d'aquí, una posició guanyadora és aquella en què existeix un moviment que porta l'adversari a una posició perdedora, i una posició perdedora és aquella en què tot moviment porta l'adversari a una posició guanyadora.

**Definició.** Siguin $X$ i $Y$ dos posicions. Aleshores, $X \oplus Y$ és la unió de les dues posicions, és a dir, en el cas del Nim, una partida que tingui les piles de pedres de $X$ i les de $Y$.

**Proposició.** Si $\bar{g}(X)$ i $\bar{g}(Y)$, aleshores $\bar{g}(X \oplus Y)$.

*Demostració.* Suposem que juguem en una pila de $X$, ja que la situació és simètrica. Aleshores, l'adversari té una jugada guanyadora en $X$. Si la juga, estarem en aquesta mateixa situació. Com que les partides són finites i sense bucles, en algun moment arribarem a que $X = \varnothing$, haurem perdut la partida $X$, i tindrem una posició perdedora en $Y$. Per tant, eventualment haurem perdut la partida $X \oplus Y$. □

**Proposició.** Si $g(X)$ i $\bar{g}(Y)$, aleshores $g(X \oplus Y)$.

*Demostració.* Tenim almenys una jugada guanyadora en $X$. Si la juguem, deixem l'adversari en la situació de la Proposició anterior, que és perduda. □

**Proposició.** Si $g(X)$ i $g(Y)$, no podem afirmar res sobre $g(X \oplus Y)$

*Demostració.* És suficient amb donar dos exemples que es contradiguin.

Cas 1: Dues piles d'una pedra cadascuna. Cada partida està guanyada, però la unió està perduda.

Cas 2: Dues piles d'una i dues pedres. Cada partida està guanyada, i jugant el moviment "treure una pedra de la pila de dos" la unió està guanyada. □

Com hem vist, saber si cadascuna de les partides està guanyada no és suficient informació per saber si la unió també ho està. Enlloc d'un booleà, necessitem un nombre natural per descriure cada posició. Aquest és el que definim com a *nimber* de la posició.

**Definició.** Donada una posició $X$, el seu nimber, $\text{nim}(x)$, es defineix de la manera següent:

- $\text{nim}\, \varnothing = 0$
- Altrament, $\text{nim}\, X$ és el mínim nimber al qual no es pot arribar amb una jugada des de $X$.

**Proposició.** $g(X) \iff \text{nim}\, X \neq 0$

*Demostració.* Si $X = \varnothing$, es compleix que $\operatorname{nim} X = 0$ i que és posició perdedora.

Sigui $X$ tal que $\operatorname{nim} X \neq 0$. Aleshores, per definició de nimber, podem arribar a una posició $Y$ amb $\operatorname{nim} Y = 0$ en una jugada.

Sigui $X$ tal que $\operatorname{nim} X = 0$. Aleshores, per definició de nimber, totes les jugades ens porten a una posició $Y$ amb $\operatorname{nim} Y \neq 0$.

Per tant, si estem en una posició amb nimber 0, l'adversari pot moure de manera que al nostre següent torn, tornem a tenir nimber 0. Si estem en una posició amb nimber diferent de 0, podem moure de manera que al nostre següent torn, tornem a tenir nimber diferent de zero.

Com que el joc és finit i sense bucles, eventualment s'arribarà a la posició buida, amb nimber 0. Si s'han seguit les normes anteriors, hi arribarà el jugador que té nimber 0 durant tota la partida. Per tant, la partida està guanyada si i només si tenim nimber diferent de zero. □

Els nimbers, de moment, no semblen útils perquè no tenim manera fàcil de calcular-los directament, donada una posició qualsevol. El següent teorema ens dóna aquesta manera.

**Teorema.** $\operatorname{nim}(X \oplus Y) = \operatorname{nim} X \oplus \operatorname{nim} Y$, on $a \oplus b$ representa el XOR bit a bit de $a$ i $b$.

*Demostració.* Recordem la definició de XOR: donats dos nombres en binari $x$ i $y$, del mateix nombre de bits (si no, es poden afegir zeros a l'esquerra per alinear-los), si denotem $z_i$ com el bit i-èsim de $z$, el resultat $z = x \oplus y$ compleix que $z_i = 1$ si $x_i \neq y_i$, i $z_i = 0$ altrament.

XOR és una operació commutativa, associativa, amb element neutre ($0 \oplus x = x$) i on cada nombre és el seu propi invers ($x \oplus x = 0$). L'invers d'un nombre és únic.

Siguin $X$ i $Y$ posicions, amb els seus nimbers respectius $x = \operatorname{nim} X$ i $y = \operatorname{nim} Y$. Tenim també la seva unió, $X \oplus Y$, amb el seu nimber $\operatorname{nim}(X \oplus Y)$, que volem veure que és igual a $z = x \oplus y$.

Vegem primer que des de $X \oplus Y$ no podem arribar a una posició amb nimber $x \oplus y$.

Si juguem a $X$, obtenim $X'$, amb nimber $\operatorname{nim} X' = x'$, i $x' \neq x$ (per definició de nimber,

no podem arribar a una posició amb el mateix nimber d'on hem començat). Suposem que $x' \oplus y = x \oplus y$. Si apliquem XOR amb $y$ per la dreta, com que és la seva pròpia inversa, obtenim $x' = x$, que és una contradicció.

Si juguem a $Y$, obtenim $Y'$, amb nimber $\operatorname{nim} Y' = y'$, i $y' \neq y$. Suposem que $x \oplus y' = x \oplus y$. Si apliquem XOR amb $x$ per l'esquerra, obtenim $y' = y$, que és una contradicció.

Vegem ara que des de $X \oplus Y$ es pot arribar a qualsevol nimber més petit que $z = x \oplus y$. Sigui $k$ un nombre natural més petit que $z$. Sigui $i$ la primera posició (des de l'esquerra) en què les representacions binàries de $z$ i $k$ difereixen. En aquesta posició, com que $z > k$, tenim $z_i = 1$, $k_i = 0$.

Com que $z_i = 1$, per definició de XOR, o bé $x_i = 1, y_i = 0$ o bé $x_i = 0, y_i = 1$. Suposem sense pèrdua de generalitat que $x_i = 1, y_i = 0$ (si no, intercanviem els papers de $x$ i $y$). Aleshores, per definició de nimber, des de la posició $X$ podem arribar a qualsevol nimber menor que $x$. Per tant, podem arribar a qualsevol nimber que, en binari, sigui igual que $x$ fins a la posició $i$ i tingui un zero a la posició $i$. Triem el moviment que ens porti a $X'$ amb nimber $x' = \operatorname{nim} X'$ que compleixi que $x' \oplus y = k$. [2].

Per tant, hem demostrat que $z = x \oplus y$ compleix les propietats que ha de complir $\operatorname{nim}(X \oplus Y)$, i en efecte són iguals. □

**Corol·lari.** $g(X \oplus Y) \iff \operatorname{nim} X \neq \operatorname{nim} Y$

*Demostració.* Fent ús de la proposició i teorema anteriors, i de que cada nombre té una única inversa per XOR, tenim:

$g(X \oplus Y) \iff \operatorname{nim}(X \oplus Y) \neq 0 \iff \operatorname{nim} X \oplus \operatorname{nim} Y \neq 0 \iff \operatorname{nim} X \neq \operatorname{nim} Y$ □

[2] Aquest nombre existeix: a l'esquerra del bit $i$ tots els bits coincideixen, i a la dreta del bit $i$, podem triar la combinació de bits adequada per tal que coincideixin.