# Algorithmique avancée

Version 3.7

Michaël Guedj

# Table des Matières

# 1 Rappels de logique

## 1.1 Logique propositionnelle

| $A$ | $B$ | $A$ **and** $B$ |
|---|---|---|
| 1 | 1 | 1 |
| 1 | 0 | 0 |
| 0 | 1 | 0 |
| 0 | 0 | 0 |

| $A$ | $B$ | $A$ **or** $B$ |
|---|---|---|
| 1 | 1 | 1 |
| 1 | 0 | 1 |
| 0 | 1 | 1 |
| 0 | 0 | 0 |

| $A$ | $\neg A$ |
|---|---|
| 1 | 0 |
| 0 | 1 |

$$A \Rightarrow B \quad \equiv \quad \Big(\neg A \ \textbf{or} \ B\Big)$$

| $A$ | $B$ | $A \Rightarrow B$ |
|---|---|---|
| 1 | 1 | 1 |
| 1 | 0 | 0 |
| 0 | 1 | 1 |
| 0 | 0 | 1 |

$$A \iff B \quad \equiv \quad \Big(A \Rightarrow B \ \textbf{and} \ B \Rightarrow A\Big)$$

| $A$ | $B$ | $A \iff B$ |
|---|---|---|
| 1 | 1 | 1 |
| 1 | 0 | 0 |
| 0 | 1 | 0 |
| 0 | 0 | 1 |

## 1.2 Logique des prédicats

$$\neg\Big(\forall x \in E,\ P(x)\Big) \quad \equiv \quad \exists x \in E,\ \neg P(x)$$

$$\neg\Big(\exists x \in E,\ P(x)\Big) \quad \equiv \quad \forall x \in E,\ \neg P(x)$$

# 2 Algorithmes sur tableaux

## 2.1 Un tableau est-il vide ?

**Algorithm 1** `est_vide` ($t$ : tableau, $n$ : taille du tableau)

```
1: if n = 0 then
2:     return True
3: else
4:     return False
5: end if
```

Complexité : $O(1)$.

## 2.2 Afficher les éléments d'un tableau

**Algorithm 2** `afficher_tableau` ($t$ : tableau, $n$ : taille du tableau)

```
1: for i ← 0, ..., n − 1 do
2:     print(t[i])
3: end for
```

Complexité : $O(n)$.

## 2.3 Afficher les éléments positifs d'un tableau

**Algorithm 3** `afficher_positifs` ($t$ : tableau, $n$ : taille du tableau)

```
1: for i ← 0, ..., n − 1 do
2:     if t[i] ≥ 0 then
3:         print(t[i])
4:     end if
5: end for
```

Complexité : $O(n)$.

### 2.4 Retourner l'éléments maximum d'un tableau

**Algorithm 4** `maximum` ($t$ : tableau, $n$ : taille du tableau)

```
1: max ← t[0]                                   ▷ on suppose n > 0
2: for i ← 1, ..., n − 1 do
3:     if t[i] ≥ max then
4:         max ← t[i]
5:     end if
6: end for
7: return max
```

Complexité : $O(n)$.

### 2.5 Retourner l'indice de l'éléments maximum d'un tableau

**Algorithm 5** `inidice_maximum` ($t$ : tableau, $n$ : taille du tableau)

```
1: max ← t[0]                                   ▷ on suppose n > 0
2: iMax ← 0
3: for i ← 1, ..., n − 1 do
4:     if t[i] ≥ max then
5:         max ← t[i]
6:         iMax ← i
7:     end if
8: end for
9: return iMax
```

Complexité : $O(n)$.

### 2.6 Retourner la somme des éléments d'un tableau

**Algorithm 6** `somme` ($t$ : tableau, $n$ : taille du tableau)

```
1: res ← 0
2: for i ← 0, ..., n − 1 do
3:     res ← res + t[i]
4: end for
5: return res
```

Complexité : $O(n)$.

## 2.7 Rechercher un élément dans un tableau

**Algorithm 7** `recherche` ($t$ : tableau, $n$ : taille du tableau, $x$ : élément)

```
1: for i ← 0, ..., n − 1 do
2:     if t[i] = x then
3:         return True
4:     end if
5: end for
6: return False
```

Complexité : $O(n)$.

# 3 Algorithmes sur matrices

## 3.1 Afficher les éléments d'une matrice

**Algorithm 8** `afficher_matrice` ($A$ : matrice $n \times m$)

```
1: for i ← 0, ..., n − 1 do                          ▷ parcours des lignes
2:     for j ← 0, ..., m − 1 do                      ▷ parcours des colonnes
3:         print(A_{i,j})
4:     end for
5:     print("\n")                                  ▷ échappement pour une nouvelle ligne
6: end for
```

Complexité : $O(n \times m)$.
Cas d'une matrice carré $n \times n$ : $O(n^2)$.

## 3.2 Additionner deux matrices

**Algorithm 9** `additionner` ($A, B$ : matrice $n \times m$)

```
1: C ← matrice n × m
2: for i ← 0, ..., n − 1 do
3:     for j ← 0, ..., m − 1 do
4:         C_{i,j} ← A_{i,j} + B_{i,j}
5:     end for
6: end for
7: return C
```

Complexité : $O(n \times m)$.

## 3.3 La matrice est-telle diagonale ?

**Définition.** La matrice carré $n \times n$, soit $M$, est diagonale si :

$$\forall i, j \in \{0, ..., n-1\},\ i \neq j \Rightarrow M_{i,j} = 0$$

**Lemme.** *La matrice carré $n \times n$, soit $M$, n'est pas diagonale si :*

$$\exists i, j \in \{0, ..., n-1\},\ i \neq j \ \textbf{and}\ M_{i,j} \neq 0$$

*Preuve.*

$$A \equiv\ \textbf{not}\ \Big(\forall i, j \in \{0, ..., n-1\},\ i \neq j \Rightarrow M_{i,j} = 0\Big)$$

$$A \equiv \exists i, j \in \{0, ..., n-1\}, \quad \textbf{not} \ \Big( i \neq j \Rightarrow M_{i,j} = 0 \Big)$$

$$A \equiv \exists i, j \in \{0, ..., n-1\}, \quad \textbf{not} \ \Big( \ \textbf{not} \ (i \neq j) \ \textbf{or} \ (M_{i,j} = 0) \Big)$$

$$A \equiv \exists i, j \in \{0, ..., n-1\}, \ (i \neq j) \ \textbf{and} \ \ \textbf{not} \ (M_{i,j} = 0)$$

$$A \equiv \exists i, j \in \{0, ..., n-1\}, \ i \neq j \ \textbf{and} \ M_{i,j} \neq 0$$

□

**Algorithm 10** `est_diagonale` ($M$ : matrice $n \times n$)

```
1: for i ← 0, ..., n − 1 do
2:     for j ← 0, ..., n − 1 do
3:         if i ≠ j and M_{i,j} ≠ 0 then
4:             return False
5:         end if
6:     end for
7: end for
8: return True
```

Complexité : $O(n^2)$.

# 4 Complexité en temps

## 4.1 Approximation asymptotique

**Définition** (Notation grand O). Soient $f$ et $g$, deux suites de $\mathbb{N} \to \mathbb{R}^+$.
$f \in O(g)$ si :

$$\exists K \in \mathbb{R}^{*+}, \quad \exists n_0 \in \mathbb{N}, \quad \text{tels que :}$$

$$\forall n \geq n_0, \quad f(n) \leq K.g(n)$$

Autrement dit, $f(n) \leq K.g(n)$ à partir d'un certain rang.

**Exemples.**

- $7n - 3 \in O(n)$
- $7n - 3 \in O(n^2)$

**Remarque 1.** *Le but est de trouver l'approximation la plus petite possible.*

## 4.2 Complexités en temps classiques

| **Complexité** | **Notation asymptotique** | **Exemple** |
|---|---|---|
| Logarithmique | $O(\log n)$ | Recherche dichotomique dans un tableau trié. |
| Linéaire | $O(n)$ | Recherche séquentielle dans un tableau. |
| Quasi-linéaire | $O(n \log n)$ | Tri fusion. |
| Quadratique | $O(n^2)$ | Tri sélection; tri à bulles. |
| Polynomiale | $O(n^k), k \geq 0$ | ... |
| Exponentielle | $O(\alpha^n), \alpha > 1$ | Algorithme récursif pour Fibonacci. |
| Factorielle | $O(n!)$ | Résolution des $n$-reines par *backtracking*. |

# 5 Tris quadratiques

## 5.1 Algorithme d'échange

**Algorithm 11** `echanger`($t$ : tableau, $i, j$ : entiers)

1: $tmp \leftarrow t[i]$
2: $t[i] \leftarrow t[j]$
3: $t[j] \leftarrow tmp$

## 5.2 Tri par sélection

**Algorithm 12** `tri_selection`($t$ : tableau, $n$ : taille du tableau)

1: **for** $i \leftarrow 0, ..., n-2$ **do**
2: $\quad i_{min} \leftarrow$ `indice_min_sous_tab`$(t, i, n-1)$
3: $\quad$ `echanger`$(t, i, i_{min})$
4: **end for**

**Algorithm 13** `indice_min_sous_tab`($t$ : tableau, $a, b$ : entiers)

1: $i_{min} \leftarrow a$
2: **for** $i \leftarrow a+1, ..., b$ **do**
3: $\quad$ **if** $t[i] < t[i_{min}]$ **then**
4: $\quad\quad i_{min} \leftarrow i$
5: $\quad$ **end if**
6: **end for**
7: **return** $i_{min}$

**Théorème.** *La complexité de* `tri_selection` *est en* $O(n^2)$.

*Preuve.* Calcul du nombre de comparaisons, soit $\mathcal{C}(n)$, de `tri_selection` (ligne 3 de l'algorithme 13).

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

$$\mathcal{C}(n) = \sum_{i=0}^{n-2} \Big( (n-1) - (i+1) + 1 \Big)$$

$$\mathcal{C}(n) = \sum_{i=0}^{n-2} \big(n-1-i-1+1\big)$$

$$\mathcal{C}(n) = \sum_{i=0}^{n-2} \big(n-1-i\big)$$

$$\mathcal{C}(n) = \sum_{i=0}^{n-2} (n-1) - \sum_{i=0}^{n-2} i$$

$$\mathcal{C}(n) = (n-1) \sum_{i=0}^{n-2} 1 - \sum_{i=0}^{n-2} i$$

$$\mathcal{C}(n) = (n-1)(n-1) - \sum_{i=0}^{n-2} i$$

$$\mathcal{C}(n) = (n-1)^2 - \sum_{i=0}^{n-2} i$$

$$\mathcal{C}(n) = (n-1)^2 - \frac{(n-2)(n-1)}{2}$$

$$\mathcal{C}(n) = (n-1)\Big((n-1) - \frac{(n-2)}{2}\Big)$$

$$\mathcal{C}(n) = (n-1)\Big(\frac{2.(n-1) - (n-2)}{2}\Big)$$

$$\mathcal{C}(n) = (n-1)\Big(\frac{2n-2-n+2}{2}\Big)$$

$$\mathcal{C}(n) = (n-1)\frac{n}{2}$$

$$\mathcal{C}(n) = \frac{n^2}{2} - \frac{n}{2} = O(n^2)$$

□

### 5.3 Tri à bulles

**Algorithm 14** `tri_à_bulles`($t$ : tableau, $n$ : taille du tableau)

```
1: for i ← n − 1, ..., 1 do
2:     for j ← 0, ..., i − 1 do
3:         if t[j + 1] < t[j] then
4:             echanger(t, j + 1, j)
5:         end if
6:     end for
7: end for
```

**Théorème.** *La complexité de* `tri_à_bulles` *est en* $O(n^2)$.

*Preuve.* Calcul du nombre de comparaisons, soit $\mathcal{C}(n)$, de `tri_à_bulles` (ligne 4 de Algorithme 14).

$$\mathcal{C}(n) = \sum_{i=n-1}^{1} \sum_{j=0}^{i-1} 1 = \sum_{i=n-1}^{1} (i - 1 - 0 + 1) = \sum_{i=n-1}^{1} i$$

$$\mathcal{C}(n) = \frac{(n-1+1)(n-1)}{2} = \frac{n(n-1)}{2} = \frac{n^2}{2} - \frac{n}{2} = O(n^2)$$

□

# 6 Récursivité

## 6.1 Considérations sur la récursivité

La version itérative d'un traitement est souvent à préférer.
En effet :

- un dépassement de pile (*stack overflow*) peut se produire ;
- l'exécution d'une version récursive d'un algorithme est généralement un peu moins rapide que celle de la version itérative correspondante ; et ce même si le nombre d'instructions est le même (à cause de la gestion des appels de fonction) ;
- un algorithme récursif (naïf) peut conduire à exécuter bien plus d'instructions que la version itérative correspondante (cas du calcul de la suite de Fibonacci).

En revanche, la récursivité peut être adaptée dans certains cas.
En effet :

- sur des structures de données naturellement récursives, il est plus facile d'écrire des algorithmes récursifs qu'itératifs ;
- certains algorithmes sont, en outre, difficiles à écrire en itératif.

## 6.2 Exemple : la fonction factorielle

**Définition** (fonction factorielle)**.** La fonction factorielle est définie, sur $\mathbb{N}$, par :

$$\begin{cases} 0! = 1 \\ n! = \Pi_{i=1}^{n} i = n \times (n-1) \times ... \times 2 \times 1 \quad \text{si } n \geq 1 \end{cases}$$

**Définition** (définition récursive de la fonction factorielle)**.**

$$n! = \begin{cases} 1 & \text{si } n = 0 \\ n \times (n-1)! & \text{si } n \geq 1 \end{cases}$$

**Algorithm 15** `fact`$(n \in \mathbb{N})$

```
1: if n = 0 then
2:     return 1
3: else
4:     return n × fact(n − 1)
5: end if
```

Complexité : $O(n)$.

**Algorithm 16** `fact_it`$(n \in \mathbb{N})$

```
1: res ← 1
2: for i ← 1, ..., n do
3:     res ← res × i
4: end for
5: return res
```

Complexité : $O(n)$.

# 7 Calcul des termes de la suite Fibonacci

## 7.1 Définition

**Définition** (suite de Fibonacci)**.**

$$F_n = \begin{cases} 0 & \text{si } n = 0 \\ 1 & \text{si } n = 1 \\ F_{n-1} + F_{n-2} & \text{si } n \geq 2 \end{cases}$$

## 7.2 Approche récursive

**Algorithm 17** `fibo_rec`$(n \in \mathbb{N})$

```
1: if n = 0 then
2:     return 0
3: else if n = 1 then
4:     return 1
5: else
6:     return fibo_rec(n − 1) + fibo_rec(n − 2)
7: end if
```

**Théorème.** *La complexité de* `fibo_rec` *est en* $O(\phi^n)$*; où* $\phi = \frac{1+\sqrt{5}}{2}$ *est le nombre d'or.*

*Preuve.* Admis. □

## 7.3 Approche itérative

**Algorithm 18** `fibo_it`$(n \in \mathbb{N})$

```
1: F ← tableau de n + 1 éléments
2: F[0] ← 0
3: F[1] ← 1
4: for i ← 2, ..., n do
5:     F[i] ← F[i − 1] + F[i − 2]
6: end for
7: return F[n]
```

**Théorème.** *La complexité de* `fibo_it` *est en* $O(n)$*.*

*Preuve.* – Nombre d'affectations : $O(n)$.

– Nombre d'additions : $O(n)$.

□

## 8 Théorème maître

**Théorème.** *Soient* $a, b, d \in \mathbb{N},\ a \geq 1,\ b \geq 2$ *; soit la fonction* $f : \mathbb{N} \to \mathbb{N}$ *définie par :*

$$f(n) = \begin{cases} O(1) & si\ n \leq 1 \\ a.f(n/b) + O(n^d) & si\ n \geq b \end{cases}$$

*alors :*

$$f(n) = \begin{cases} O(n^d) & si\ a < b^d \\ O(n^d.\log n) & si\ a = b^d \\ O(n^{\log_b a}) & si\ a > b^d \end{cases}$$

*Preuve.*

$$f(n) = a.f(\frac{n}{b}) + O(n^d)$$

$$f(n) = a.\Big(a.f(\frac{n}{b^2}) + O(\frac{n^d}{b^d})\Big) + O(n^d)$$

$$f(n) = a^2.f(\frac{n}{b^2}) + a.O(\frac{n^d}{b^d}) + O(n^d)$$

$$f(n) = a^2.\Big(a.f(\frac{n}{b^3}) + O(\frac{n^d}{b^{2.d}})\Big) + a.O(\frac{n^d}{b^d}) + O(n^d)$$

$$f(n) = a^3.f(\frac{n}{b^3}) + a^2.O(\frac{n^d}{b^{2.d}}) + a.O(\frac{n^d}{b^d}) + O(n^d)$$

$$f(n) = a^3.\Big(a.f(\frac{n}{b^4}) + O(\frac{n^d}{b^{3.d}})\Big) + a^2.O(\frac{n^d}{b^{2.d}}) + a.O(\frac{n^d}{b^d}) + O(n^d)$$

$$f(n) = a^4.f(\frac{n}{b^4}) + a^3.O(\frac{n^d}{b^{3.d}}) + a^2.O(\frac{n^d}{b^{2.d}}) + a.O(\frac{n^d}{b^d}) + O(n^d)$$

C'est-à-dire

$$f(n) = a^4.f(\frac{n}{b^4}) + \sum_{i=0}^{3} a^i.O(\frac{n^d}{b^{i.d}})$$

Au rang $k$, on trouve

$$f(n) = a^k.f(\frac{n}{b^k}) + \sum_{i=0}^{k-1} a^i.O(\frac{n^d}{b^{i.d}})$$

On a

$$\frac{n}{b^k} = 1 \iff n = b^k \iff \log_b n = k$$

D'où

$$f(n) = a^{\log_b n}.f(1) + \sum_{i=0}^{\log_b n-1} a^i.O(\frac{n^d}{b^{i.d}})$$

$$f(n) = a^{\log_b n}.O(1) + \sum_{i=0}^{\log_b n-1} a^i.O(\frac{n^d}{b^{i.d}})$$

On a

$$a^{\log_b n} = e^{\ln a.\ln n.\frac{1}{\ln b}} = n^{\log_b a}$$

D'où

$$f(n) = n^{\log_b a}.O(1) + \sum_{i=0}^{\log_b n-1} a^i.O(\frac{n^d}{b^{i.d}})$$

On a

$$\sum_{i=0}^{\log_b n-1} a^i.O(\frac{n^d}{b^{i.d}}) = \sum_{i=0}^{\log_b n-1} n^d.O(\frac{a^i}{b^{i.d}}) = n^d.\sum_{i=0}^{\log_b n-1} O\Big((\frac{a}{b^d})^i\Big)$$

Soit

$$f(n) = n^{\log_b a}.O(1) + n^d.\sum_{i=0}^{\log_b n-1} O\Big((\frac{a}{b^d})^i\Big)$$

(i) <u>Cas : $a = b^d$</u>

$$f(n) = n^{\log_b b^d}.O(1) + n^d.\sum_{i=0}^{\log_b n-1} O(1)$$

$$f(n) = n^d.O(1) + n^d.O(\log_b n)$$

$$f(n) = O(n^d.\log n)$$

(ii) <u>Cas : $a < b^d$</u>

Alors $\frac{a}{b^d} < 1$, donc

$$\sum_{i=0}^{\log_b n-1} O\Big((\frac{a}{b^d})^i\Big) = O\Big(\frac{1 - (\frac{a}{b^d})^{\log_b n}}{1 - \frac{a}{b^d}}\Big)$$

$$\sum_{i=0}^{\log_b n-1} O\Big((\frac{a}{b^d})^i\Big) = O\Big(\frac{1}{1 - \frac{a}{b^d}}\Big)$$

$$\sum_{i=0}^{\log_b n-1} O\Big((\frac{a}{b^d})^i\Big) = O(1)$$

D'où

$$f(n) = n^{\log_b a}.O(1) + n^d.O(1)$$

En outre,

$$a < b^d \iff \log_b a < \log_b b^d = d$$

D'où

$$f(n) = O(n^d).O(1) + n^d.O(1)$$
$$f(n) = O(n^d)$$

(iii) <u>Cas : $a > b^d$</u>

On a

$$\sum_{i=0}^{\log_b n-1} O\Big((\frac{a}{b^d})^i\Big) = O\Big(\frac{1-(\frac{a}{b^d})^{\log_b n}}{1-\frac{a}{b^d}}\Big)$$

$$\sum_{i=0}^{\log_b n-1} O\Big((\frac{a}{b^d})^i\Big) = O\Big(\frac{(\frac{a}{b^d})^{\log_b n}-1}{\frac{a}{b^d}-1}\Big)$$

$$\sum_{i=0}^{\log_b n-1} O\Big((\frac{a}{b^d})^i\Big) = O\Big(\frac{1}{\frac{a}{b^d}-1}.\Big((\frac{a}{b^d})^{\log_b n}-1\Big)\Big)$$

$$\sum_{i=0}^{\log_b n-1} O\Big((\frac{a}{b^d})^i\Big) = O\Big((\frac{a}{b^d})^{\log_b n}\Big)$$

$$\sum_{i=0}^{\log_b n-1} O\Big((\frac{a}{b^d})^i\Big) = O\Big(\frac{a^{\log_b n}}{(b^{d.\log_b n})}\Big) = O\Big(\frac{a^{\log_b n}}{(b^{\log_b n})^d}\Big)$$

$$\sum_{i=0}^{\log_b n-1} O\Big((\frac{a}{b^d})^i\Big) = O\Big(\frac{a^{\log_b n}}{n^d}\Big)$$

On a donc

$$f(n) = n^{\log_b a}.O(1) + n^d.O\Big(\frac{a^{\log_b n}}{n^d}\Big)$$

$$f(n) = n^{\log_b a}.O(1) + O\Big(\frac{n^d.a^{\log_b n}}{n^d}\Big)$$

$$f(n) = n^{\log_b a}.O(1) + O(a^{\log_b n})$$

Comme $a^{\log_b n} = n^{\log_b a}$, on obtient

$$f(n) = n^{\log_b a}.O(1) + O(n^{\log_b a})$$

Soit

$$f(n) = O(n^{\log_b a})$$

□

# 9 Diviser pour régner – exponentiation rapide

## 9.1 Approche *Diviser pour régner*

Principe.

1. **Diviser** : découper le problème à résoudre en $a$ sous-problèmes (de taille $n/b$ chacun);
2. **Régner** : résoudre *récursivement* les $a$ sous-problèmes ;
3. **Combiner** : à partir des solutions des $a$ sous-problèmes, calculer en $O(n^d)$ une solution au problème à résoudre.

La complexité peut alors se traduire par l'équation :

$$t(n) = a.t(n/b) + O(n^d)$$

(plus généralement par l'équation $t(n) = a.t(n/b)+O(\tau(n))$ avec $\tau : \mathbb{N} \to \mathbb{N}$).

## 9.2 Exponentiation rapide

**Théorème.** $\forall x \in \mathbb{R}$,

$$x^n = \begin{cases} 1 & si\ n = 0 \\ (x^2)^{\frac{n}{2}} & si\ n > 0\ et\ n \equiv 0 \mod 2 \\ x.(x^2)^{\frac{n-1}{2}} & si\ n > 0\ et\ n \equiv 1 \mod 2 \end{cases}$$

*Preuve.* – Si $n > 0$ et $n \equiv 0 \mod 2$, alors $\exists k \in \mathbb{N}^*$ tel que $n = 2.k$. On a $k = \frac{n}{2}$ et

$$x^n = x^{2.k} = (x^2)^k = (x^2)^{\frac{n}{2}}$$

– Si $n > 0$ et $n \equiv 1 \mod 2$, alors $\exists k \in \mathbb{N}$ tel que $n = 2.k+1$. On a $k = \frac{n-1}{2}$ et

$$x^n = x^{2.k+1} = x^{2.k}.x^1 = x.(x^2)^k = x.(x^2)^{\frac{n-1}{2}}$$

□

**Algorithm 19** `exp_rapide`$(x, n)$

```
1: if n = 0 return 1 end if
2:
3: if n ≡ 0 mod 2 then
4:     return exp_rapide(x.x, n/2)
5: else
6:     return x × exp_rapide(x.x, (n−1)/2)
7: end if
```

**Théorème.** *La complexité de* `exp_rapide` *est en* $O(\log n)$.

*Preuve.* Soit $\mathcal{C}(n)$ le nombre de comparaisons pour une instance de taille $n$. On a,

$$\mathcal{C}(n) = \mathcal{C}(n/2) + O(1)$$

On invoque le théorème maître avec $a = 1$, $b = 2$ et $d = 0$. On a

$$a = 1 = 2^d$$

D'où

$$\mathcal{C}(n) = O(n^d . \log n)$$

Comme $n^0 = 1$, on trouve

$$\mathcal{C}(n) = O(\log n)$$

□

# 10 Recherche dichotomique

**Algorithm 20** `dicho_init`($t$ : tableau, $n$ : taille du tableau, $x$ : élément)

1: $d \leftarrow 0$
2: $f \leftarrow n - 1$
3: **return** `dicho`$(t, d, f, x)$

**Algorithm 21** `dicho`($t$ : tableau, $d, f$ : indices, $x$ : élément)

1: **if** $d > f$ **then return** $-1$ **end if** ▷ non trouvé
2: **if** $f = d$ **then**
3: **if** $t[d] = x$ **then return** $d$ **else return** $-1$ **end if**
4: **end if**
5: $m \leftarrow \lfloor \frac{d+f}{2} \rfloor$ ▷ partie entière
6: **if** $t[m] = x$ **then**
7: **return** $m$
8: **end if**
9: **if** $t[m] < x$ **then**
10: **return** `dicho`$(t, m + 1, f, x)$
11: **else**
12: **return** `dicho`$(t, d, m - 1, x)$
13: **end if**

**Théorème.** *La complexité de* `dicho` *est en* $O(\log n)$.

*Preuve.* Soit $\mathcal{C}(n)$ le nombre de comparaisons pour une instance de taille $n$. On a,

$$\mathcal{C}(n) = \mathcal{C}(n/2) + O(1)$$

On invoque le théorème maître avec $a = 1$, $b = 2$ et $d = 0$. On a

$$a = 1 = 2^d$$

D'où

$$\mathcal{C}(n) = O(n^d . \log n)$$

Comme $n^0 = 1$, on trouve

$$\mathcal{C}(n) = O(\log n)$$

□

*Deuxième preuve.* Soit $\mathcal{C}(n)$ le nombre de comparaisons pour une instance de taille $n$. On a,

$$\mathcal{C}(n) = \gamma + \mathcal{C}(\frac{n}{2}) \quad (\gamma \text{ constante})$$

La deuxième instance appelée vérifie :

$$\mathcal{C}(\frac{n}{2}) = \gamma + \mathcal{C}(\frac{n}{4})$$

D'où,

$$\mathcal{C}(n) = \gamma + \mathcal{C}(\frac{n}{2}) = \gamma + \Big(\gamma + \mathcal{C}(\frac{n}{4})\Big) = 2\gamma + \mathcal{C}(\frac{n}{4})$$

Soit,

$$\underline{\mathcal{C}(n) = 2\gamma + \mathcal{C}(\frac{n}{2^2})}$$

La troisième instance appelée vérifie :

$$\mathcal{C}(\frac{n}{2^2}) = \gamma + \mathcal{C}(\frac{n}{2^3})$$

D'où,

$$\underline{\mathcal{C}(n) = 3\gamma + \mathcal{C}(\frac{n}{2^3})}$$

Par suite, $\mathcal{C}(n)$ s'écrit :

$$\underline{\mathcal{C}(n) = k\gamma + \mathcal{C}(\frac{n}{2^k})}$$

On a :

$$\frac{n}{2^k} = 1 \Rightarrow \underline{\mathcal{C}(\frac{n}{2^k}) = O(1)}$$

Et :

$$\frac{n}{2^k} = 1 \iff n = 2^k \iff \underline{\log_2 n = k}$$

$\mathcal{C}(n)$ s'écrit alors :

$$\mathcal{C}(n) = k\gamma + \mathcal{C}(\frac{n}{2^k}) = \log_2(n)\gamma + O(1) \in O(\log n)$$

□

# 11 Tri fusion

**Algorithm 22** `tri_fusion`($lst$ : liste de taille $n$)

1: **if** $n = 1$ **return** $lst$ **end if**
2: $m = \lfloor n/2 \rfloor$ ▷ Partie entière.
3: $lst_1 \leftarrow$ `tri_fusion`($lst[0 \to m-1]$)
4: $lst_2 \leftarrow$ `tri_fusion`($lst[m \to n-1]$)
5: **return** `fusion`($lst_1, lst_2$)

**Algorithm 23** `fusion`($lst_1$ : liste de taille $n_1$, $lst_2$ : liste de taille $n_2$)

1: $res \leftarrow$ `[ ]`
2: **while not** $\Big($`est_vide`($lst_1$) **and** `est_vide`($lst_2$)$\Big)$ **do**
3: **if** `est_vide`($lst_1$) **then**
4: $res \leftarrow res + lst_2$
5: $lst_2 \leftarrow$ `[ ]`
6: **else if** `est_vide`($lst_2$) **then**
7: $res \leftarrow res + lst_1$
8: $lst_1 \leftarrow$ `[ ]`
9: **else if** `head`($lst_1$) $\leq$ `head`($lst_2$) **then**
10: $res \leftarrow res +$ `[ head`($lst_1$) `]`
11: $lst_1 \leftarrow$ `tail`($lst_1$)
12: **else**
13: $res \leftarrow res +$ `[ head`($lst_2$) `]`
14: $lst_2 \leftarrow$ `tail`($lst_2$)
15: **end if**
16: **end while**
17: **return** $res$

**Théorème.** *L'algorithme* `fusion` *termine.*

*Preuve.* L'algorithme termine lorsque les listes $lst_1$ et $lst_2$ sont vides.

Soit la prédicat $P$ quantifiant, pour chaque tour de boucle $i$, la somme des tailles des liste $lst_1$ et $lst_2$ ; formellement

$$P(i) := |lst_1^i| + |lst_2^i|$$

où $lst_1^i$ (resp. $lst_2^i$) correspond à la liste $lst_1$ (resp. $lst_2$) au $i$-ème tour de boucle.

Par l'absurde (descente infinie), on suppose que l'algorithme ne termine pas, i.e., $\forall i \in \mathbb{N},\ P(i) > 0$.

On vérifie que le prédicat $P$ assure : $\forall i \in \mathbb{N},\ P(i) > P(i+1)$.

On considère la suite $(P_i)_{i\in\mathbb{N}}$ définie par : $\forall i \in \mathbb{N},\ P_i = P(i)$.

Ainsi, la suite $(P_i)_{i\in\mathbb{N}}$ est

(i) à valeur entière ($P : \mathbb{N} \to \mathbb{N}$) ;

(ii) infinie ;

(iii) strictement décroissante.

D'où la fausseté de l'hypothèse de non terminaison de l'algorithme ($\forall i \in \mathbb{N},\ P(i) > 0$).

Conclusion : l'algorithme termine ($\exists i \in \mathbb{N},\ P(i) = 0$). □

**Théorème.** *La complexité de* `fusion` *est en* $O(n)$.

*Proof.* On a

$$t(n) = O(1) + boucle(0, lst_1^0, lst_2^0)$$

et

$$boucle(i, lst_1^i, lst_2^i) = \begin{cases} O(1) & \text{si } |lst_1^i| = |lst_2^i| = 0 \\ O(1) & \text{si } |lst_1^i| = 0 \text{ ou } |lst_2^i| = 0 \\ O(1) + boucle(i, lst_1^{i+1}, lst_2^{i+1}) & \text{sinon ; avec :} \\ & \quad |lst_1^{i+1}| + |lst_2^{i+1}| = |lst_1^i| + |lst_2^i| - 1 \end{cases}$$

On associe la suite $(u_j)_{j\in\mathbb{N}}$ ; $u_j$ quantifiant le temps en fonction de $|lst_1^j| + |lst_2^j|$ :

$$u_j = \begin{cases} O(1) & \text{si } j = 0 \\ O(1) + u_{j-1} & \text{sinon} \end{cases}$$

On a $n = |lst_1^0| + |lst_2^0|$ et

$$t(n) = O(1) + u_n$$

On calcule $u_n$.

$$u_n = O(1) + u_{n-1}$$

$$u_n = O(1) + \big(O(1) + u_{n-2}\big)$$

$$u_n = 2.O(1) + u_{n-2}$$

...

$$u_n = k.O(1) + u_{n-k}$$

Soit

$$u_n = n.O(1) + u_0$$
$$u_n = n.O(1) + O(1)$$
$$u_n = O(n)$$

D'où

$$t(n) = O(n)$$

□

**Théorème.** *La complexité de* `tri_fusion` *est en* $O(n \log n)$.

*Preuve.* Soit $\mathcal{C}(n)$ le nombre de comparaisons pour une instance de taille $n$. On a

$$\mathcal{C}(n) = 2.\mathcal{C}(n/2) + O(n)$$

On invoque le théorème maître avec $a = 2$, $b = 2$ et $d = 1$. On a

$$a = 2 = 2^d$$

D'où

$$\mathcal{C}(n) = O(n^d.\log n)$$

C'est-à-dire

$$\mathcal{C}(n) = O(n.\log n)$$

□

*Deuxième preuve.* Soit $\mathcal{C}(n)$ le nombre de comparaisons pour une instance de taille $n$.

$$\mathcal{C}(n) = 1 + 2.\mathcal{C}(\frac{n}{2}) + \gamma.n$$

Où $\gamma$ est une constante. On a de même :

$$\mathcal{C}(\frac{n}{2}) = 1 + 2.\mathcal{C}(\frac{n}{4}) + \gamma.\frac{n}{2}$$

Soit :

$$\mathcal{C}(n) = 1 + 2.\Big(1 + 2.\mathcal{C}(\frac{n}{4}) + \gamma.\frac{n}{2}\Big) + \gamma.n$$
$$\mathcal{C}(n) = 1 + 2 + 4.\mathcal{C}(\frac{n}{4}) + \gamma.n + \gamma.n$$

$$\mathcal{C}(n) = 4.\mathcal{C}(\frac{n}{4}) + 2\gamma.n + 3$$

$$\underline{\mathcal{C}(n) = 2^2.\mathcal{C}(\frac{n}{2^2}) + 2\gamma.n + (2+1)}$$

$$\mathcal{C}(\frac{n}{2^2}) = 1 + 2.\mathcal{C}(\frac{n}{2^3}) + \gamma.\frac{n}{2^2}$$

$$\mathcal{C}(n) = 2^2.\Big(1 + 2.\mathcal{C}(\frac{n}{2^3}) + \gamma.\frac{n}{2^2}\Big) + 2\gamma.n + (2+1)$$

$$\mathcal{C}(n) = 2^2 + 2^2.2.\mathcal{C}(\frac{n}{2^3}) + \gamma.n + 2\gamma.n + (2+1)$$

$$\underline{\mathcal{C}(n) = 2^3.\mathcal{C}(\frac{n}{2^3}) + 3\gamma.n + (2^2+2+1)}$$

$$\mathcal{C}(\frac{n}{2^3}) = 1 + 2.\mathcal{C}(\frac{n}{2^4}) + \gamma.\frac{n}{2^3}$$

$$\mathcal{C}(n) = 2^3.\Big(1 + 2.\mathcal{C}(\frac{n}{2^4}) + \gamma.\frac{n}{2^3}\Big) + 3\gamma.n + (2^2+2+1)$$

$$\mathcal{C}(n) = 2^3 + 2^3.2.\mathcal{C}(\frac{n}{2^4}) + 2^3.\gamma.\frac{n}{2^3} + 3\gamma.n + (2^2+2+1)$$

$$\mathcal{C}(n) = 2^3 + 2^4.\mathcal{C}(\frac{n}{2^4}) + \gamma.n + 3\gamma.n + (2^2+2+1)$$

$$\underline{\mathcal{C}(n) = 2^4.\mathcal{C}(\frac{n}{2^4}) + 4\gamma.n + (2^3+2^2+2+1)}$$

Par suite,

$$\underline{\mathcal{C}(n) = 2^t.\mathcal{C}(\frac{n}{2^t}) + t.\gamma.n + (2^t + ... + 2^2 + 2 + 1)}$$

On a :

$$2^t + ... + 2^2 + 2 + 1 = \frac{2^{t+1}-1}{2-1} = 2^{t+1} - 1$$

D'où,

$$\underline{\mathcal{C}(n) = 2^t.\mathcal{C}(\frac{n}{2^t}) + t.\gamma.n + 2^{t+1} - 1}$$

On a :

$$\frac{n}{2^t} = 1 \iff n = 2^t \iff t = \log_2 n$$

Et $\mathcal{C}(1) = 1$ ; d'où :

$$\mathcal{C}(n) = 2^{\log_2 n}.1 + \log_2(n)\gamma.n + 2^{\log_2(n)+1} - 1$$

$$\mathcal{C}(n) = n + n.\log_2(n).\gamma + 2^{\log_2(n)}.2 + -1$$

$$\mathcal{C}(n) = n + n.\log_2(n).\gamma + 2.n - 1$$

$$\underline{\mathcal{C}(n) = n.\log_2(n).\gamma + 3.n - 1} \in O(n \log n)$$

□

# 12 Tri comptage

## 12.1 Algorithme

Le tri comptage (ou tri casier, ou encore *counting sort* en anglais) est un algorithme de tri par dénombrement, qui s'applique sur des valeurs entières. Le principe repose sur :

(i) la construction d'un histogramme des données ;

(ii) le balayage de cet histogramme, afin de reconstruire les données triées.

Nous donnons, ci-dessous, l'algorithme. Noter que : *tab* est un tableau de $n$ éléments (indexés à partir de 0), et $M$ est la valeur maximale de *tab* (connue *a priori*).

**Algorithm 24** `tri_comptage`(*tab* : tableau de $n$ éléments, $M$ : entier)

```
1:  histo ← tableau de M + 1 éléments
2:                                  ▷ Initialisation de l'histogramme à 0
3:  for i ← 0, ..., M do
4:      histo[ tab[i] ] ← 0
5:  end for
6:                                  ▷ Remplissage de l'histogramme
7:  for i ← 0, ..., n − 1 do
8:      histo[ tab[i] ] ← histo[ tab[i] ] + 1
9:  end for
10:                                 ▷ Remplissage du tableau trié
11: cpt ← 0
12: for i ← 0, ..., M do
13:     for j ← 0, ..., histo[i] − 1 do
14:         tab[cpt] ← i
15:         cpt ← cpt + 1
16:     end for
17: end for
```

## 12.2 Complexité

**Théorème.** `tri_comptage` *est en* $O(n + M)$.

*Preuve.* (i) Première boucle : $O(M)$ affectations.

(ii) Deuxième boucle : $O(n)$ affectations.

(iii) Calcul du nombre d'affectations, soit $\mathcal{C}$, de la troisième boucle. On a :

$$\mathcal{C} = \sum_{i=0}^{M} \sum_{j=0}^{histo[i]-1} 2$$

$$\mathcal{C} = 2\sum_{i=0}^{M} \sum_{j=0}^{histo[i]-1} 1$$

$$\mathcal{C} = 2\sum_{i=0}^{M} \max\{|histo[i]|,\ 1\}$$

On pose :

$$K := \{k \in \{0, ..., M\} : histo[k] \neq 0\}$$

On a :

$$\mathcal{C} = 2\Big(\sum_{i\in K} |histo[i]| + \sum_{i\in\{0,...,M\}-K} 1\Big)$$

On remarque que :

$$\sum_{i\in K} |histo[i]| = n$$

D'où,

$$\mathcal{C} = 2\Big(n + \sum_{i\in\{0,...,M\}-K} 1\Big)$$

Mais,

$$\sum_{i\in\{0,...,M\}-K} 1 \leq M$$

Soit,

$$\mathcal{C} \leq 2\Big(n + M\Big)$$

D'où,

$$\mathcal{C} = O(n + M)$$

□

### 12.3 Exemple

Pour trier le tableau suivant par le tri comptage.

| Indice | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Valeur | 3 | 0 | 0 | 2 | 5 | 1 | 3 | 1 | 1 | 5 | 5 | 2 | 2 | 0 | 1 | 2 | 2 | 2 | 7 | 1 |

On passe par la construction de l'histogramme :

| Indice | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| Valeur | 3 | 5 | 6 | 2 | 0 | 3 | 0 | 1 |

# 13 Représentation des graphes

## 13.1 Considérations préliminaires

Soit un graphe $G = (S, A)$ tel que : $|S| = n$ et $|A| = m$ (avec $n, m \in \mathbb{N}$).
Les sommets de $G$ sont numérotés de 0 à $n - 1$.

## 13.2 Représentation par matrice d'adjacence

**Définition.** La matrice d'adjacence du graphe $G$, soit $M$, est une matrice booléenne de type $n \times n$ vérifiant :

$$M_{i,j} = \begin{cases} 1 & \text{si } i \text{ et } j \text{ sont adjacents} \\ 0 & \text{sinon} \end{cases}$$

Pour $i, j \in \{0, ..., n-1\}$, si $s_i$ est le $i$-ième sommet, et si $s_j$ est le $j$-ième sommet, alors :

$$M_{i,j} = 1 \iff (s_i, s_j) \in A$$

## 13.3 Représentation par liste d'adjacence

**Définition.** La liste d'adjacence du graphe $G$, soit `succ`, est une liste indexée par les sommets de $G$, et telle que :

$$\forall s \in S,\ \texttt{succ}(s) = \{s' : (s, s') \in A\}$$

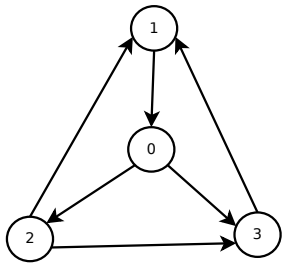
Autrement dit, $\forall s \in S$, $\texttt{succ}(s)$ est l'ensemble des sommets adjacents à $s$.

## 13.4 Exemple

Soit le graphe $G = (S, A)$, défini par :

$$\begin{cases} S = \{0, 1, 2, 3\} \\ A = \{(0, 2), (0, 3), (1, 0), (2, 1), (2, 3), (3, 1)\} \end{cases}$$

Un tel graphe peut être représenté comme suit :




Les représentations par matrice et liste d'adjacence sont données ci-après.

| Matrice d'adjacence | Liste d'adjacence |
|---|---|
| $\begin{pmatrix} 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 \end{pmatrix}$ | $\begin{array}{\|c\|c\|} \hline 0 & \{2,3\} \\ \hline 1 & \{0\} \\ \hline 2 & \{1,3\} \\ \hline 3 & \{1\} \\ \hline \end{array}$ |

## 13.5 Espace mémoire

| Matrice d'adjacence | Liste d'adjacence |
|---|---|
| $O(n^2)$ | $O(n+m)$ |

## 13.6 Complexité de quelques opérations

| Opérations | Matrice d'adjacence | Liste d'adjacence |
|---|---|---|
| Tester l'existence d'un arc $s \to s'$ | $O(1)$ | $O(\|\texttt{succ}(s)\|)$ |
| Retourner les sommets adjacents à un sommet | $O(n)$ | $O(1)$ |
| Parcourir l'ensemble des arcs | $O(n^2)$ | $O(m)$ |

## 13.7 Choix d'utilisation

- D'une manière générale, on considère que si le graphe a "peu" d'arêtes, il est plus intéressant d'utiliser une représentation par liste d'adjacence, plutôt que par matrice d'adjacence (qui contiendrait alors beaucoup de 0).
- Mais si le graphe a "beaucoup" d'arêtes, il est plus intéressant d'utiliser une matrice d'adjacence.

## 13.8 Relation entre sommets adjacents et arêtes

**Définition.** Soit la fonction $\widetilde{\texttt{succ}}$, définie par

$$\widetilde{\texttt{succ}} : S \to A$$

$$s \mapsto \widetilde{\texttt{succ}}(s) := \{(s, s') : s' \in \texttt{succ}(s)\}$$

qui associe, à chaque sommet, l'ensemble des arêtes qui lui sont adjacentes.

**Lemme.** *Si le graphe $G$ est orienté,*

$$\sum_{s \in S} |\widetilde{\texttt{succ}}(s)| = |A|$$

*Preuve.* $\{\widetilde{\texttt{succ}}(s) : s \in S\}$ est une partition de $A$ ; i.e.,

$$\bigcup_{s \in S} \widetilde{\texttt{succ}}(s) = A$$

et

$$\bigcap_{s \in S} \widetilde{\texttt{succ}}(s) = \emptyset$$

□

**Lemme.** *Si le graphe $G$ est non orienté,*

$$\sum_{s \in S} |\widetilde{\texttt{succ}}(s)| = 2.|A|$$

*Preuve.* D'une part,

$$\bigcup_{s \in S} \widetilde{\texttt{succ}}(s) = A$$

D'autre part, quel que soient $s$ et $s'$ de $S$, et $a \in A$, tels que $a = (s, s') = (s', s)$ ; alors

$$a \in \widetilde{\texttt{succ}}(s) \cap \widetilde{\texttt{succ}}(s')$$

(chaque arête est comptée exactement 2 fois). □

On note que, $\forall s \in S$, $|\widetilde{\texttt{succ}}(s)| = |\texttt{succ}(s)|$. On en déduit les théorèmes suivants.

**Théorème.** *Si le graphe $G$ est orienté,*

$$\sum_{s \in S} |\texttt{succ}(s)| = |A|$$

*Preuve.* On a :

$$\sum_{s \in S} |\widetilde{\texttt{succ}}(s)| = \sum_{s \in S} |\texttt{succ}(s)|$$

Et on a précédemment démontré que :

$$\sum_{s \in S} |\widetilde{\texttt{succ}}(s)| = |A|$$

□

**Théorème.** *Si le graphe $G$ est non orienté,*

$$\sum_{s \in S} |\texttt{succ}(s)| = 2.|A|$$

*Preuve.* On a :

$$\sum_{s \in S} |\widetilde{\texttt{succ}}(s)| = \sum_{s \in S} |\texttt{succ}(s)|$$

Et on a précédemment démontré que :

$$\sum_{s \in S} |\widetilde{\texttt{succ}}(s)| = 2.|A|$$

□

# 14 Arbres

## 14.1 Arbres – arbres binaires

### 14.1.1 Arbres binaires et profondeur

**Définition** (graphe connexe)**.** Un graphe est connexe si : pour tous sommets $s$ et $s'$, il existe une chaîne reliant $s$ à $s'$.

**Définition** (arbre)**.** Un arbre est un graphe connexe sans cycle, dont on distingue un sommet appelé racine.

**Définition** (arbre binaire)**.** Un arbre binaire est un arbre, dont tout nœud possède, au plus, deux successeurs.

**Définition** (profondeur d'un nœud)**.** Soit $\mathcal{A}$ un arbre binaire, la profondeur d'un nœud $s \in \mathcal{A}$, notée $\mathfrak{p}(s)$, est définie par :

(i) $\mathfrak{p}(s) = 1$, si $s$ est racine de $\mathcal{A}$ ;

(ii) $\mathfrak{p}(s) = \mathfrak{p}(\text{parent de } s) + 1$, sinon.

**Définition** (profondeur d'un arbre)**.**

$$\mathfrak{p}(\mathcal{A}) := \max\{\mathfrak{p}(s) \mid s \text{ est une feuille de } \mathcal{A}\}$$

### 14.1.2 Arbres binaires parfaits

**Définition** (arbre binaire parfait)**.** Un arbre binaire parfait est un arbre binaire, tel que

(i) tout nœud interne (i.e. non feuille), possède exactement deux successeurs ;

(ii) toutes les feuilles sont à la même profondeur de la racine.

**Lemme.** $\forall n \in \mathbb{N}$,

$$\sum_{i=0}^{n} 2^i = 2^{n+1} - 1$$

*Preuve.* Par récurrence sur $n$.

(i) $n = 0$ ; $2^0 = 1 = 2^1 - 1$.

(ii) Hypothèse de récurrence : soit $k \in \mathbb{N}$ tel que

$$\sum_{i=0}^{k} 2^i = 2^{k+1} - 1$$

(iii) $n = k + 1$.

$$\sum_{i=0}^{k+1} 2^i = \sum_{i=0}^{k} 2^i + 2^{k+1}$$

Par l'hypothèse de récurrence,

$$\sum_{i=0}^{k+1} 2^i = 2^{k+1} - 1 + 2^{k+1}$$

$$\sum_{i=0}^{k+1} 2^i = 2.2^{k+1} - 1$$

$$\sum_{i=0}^{k+1} 2^i = 2^{k+2} - 1$$

□

**Théorème.** *Un arbre binaire parfait, de $n$ nœuds, a une profondeur en $O(\log n)$.*

*Preuve.* Soit $\mathcal{A}$ un arbre binaire parfait de $n$ nœuds.

$$\mathfrak{p}(A) = 1 \quad \text{pour} \quad n = 1 = 2^2 - 1$$

$$\mathfrak{p}(A) = 2 \quad \text{pour} \quad n = 1 + 2 = 3 = 2^2 - 1$$

$$\mathfrak{p}(A) = 3 \quad \text{pour} \quad n = 1 + 2 + 2^2 = 7 = 2^3 - 1$$

Soit $q \in \mathbb{N}^*$ tel que

$$\mathfrak{p}(A) = q \quad \text{pour} \quad n = \sum_{i=0,\ldots,q-1} 2^i = 2^q - 1$$

On a

$$\mathfrak{p}(A) = q + 1 \quad \text{pour} \quad n = \sum_{i=0,\ldots,q-1} 2^i + 2.2^{q-1} = \sum_{i=0,\ldots,q} 2^i = 2^{q+1} - 1$$

La récurrence établie, on a ainsi, $\forall k \in \mathbb{N}^*$,

$$\mathfrak{p}(A) = k \quad \text{pour} \quad n = 2^k - 1$$

Soit

$$2^k = n + 1 \iff k = \log_2(n+1) \in O(\log n)$$

□

### 14.1.3 Parcours préfixe et infixe d'un arbre binaire

– On suppose qu'un noeud peut-être "nul" (sa profondeur est alors 0 par convention).

– Initialement, la racine de l'arbre est passée en argument de l'algorithme.

**Algorithm 25** `prefixe` ($s \in \mathcal{A}$)

```
1: if est_nul(s) then
2:     return
3: end if
4: print(s)
5: prefixe(s_g)
6: prefixe(s_d)
```

**Algorithm 26** `infixe` ($s \in \mathcal{A}$)

```
1: if est_nul(s) then
2:     return
3: end if
4: infixe(s_g)
5: print(s)
6: infixe(s_d)
```

**Théorème.** *La complexité du parcours préfixe (ou infixe), dans un arbre binaire, est en $O(n)$.*

## 14.2 Arbre binaire de recherche

### 14.2.1 Définition

**Définition** (ABR)**.** Un arbre binaire de recherche (ABR) est un arbre binaire valué, qui est soit un arbre vide ; soit un arbre vérifiant, pour tout noeud $s$ :

– $\forall s' \in G(s), s'.val \leq s.val$ ;

– $\forall s' \in D(s), s.val < s'.val$ ;

où $G(s)$ (resp. $D(s)$) est le sous-arbre gauche (resp. droit) du noeud $s$.

### 14.2.2 Recherche dans un ABR

**Algorithm 27** `recherche_ABR` ($s \in \mathcal{A}$, $x \in V$ : valeur recherchée)

```
1: if est_nul(s) then
2:     return False
3: else if s.val = x then
4:     return True
5: else if s.val > x then
6:     return recherche_ABR(s.f_g, x)
7: else
8:     return recherche_ABR(s.f_d, x)
9: end if
```

**Théorème.** *La complexité de la recherche dans un ABR est, en moyenne, en* $O(\log n)$.

*Preuve.* Admis. □

## 14.3 Parcours infixe dans un ABR

**Théorème.** *Le parcours infixe d'un ABR donne une séquence des noeuds triés, selon l'ordre croissant des valeurs.*

*Preuve.* (Par récurrence sur la taille de l'ABR). Si $|\mathcal{A}| = 1$, alors la proposition est trivialement vraie.

Supposons que, pour tout ABR de taille $m \leq k$, la proposition soit vraie. Considérons un ABR de taille $k + 1$ ; alors la séquence affichée est de la forme :

séquence affichée par `infixe`$(s.f_g)$ . $s$ . séquence affichée par `infixe`$(s.f_d)$.

Par définition d'un ABR,

– $\forall s' \in G(s), s'.val \leq s.val$ ;

– $\forall s' \in D(s), s.val < s'.val$.

D'autre part, l'hypothèse de récurrence nous assure que la séquence affichée par `infixe`$(s.f_g)$ (resp. `infixe`$(s.f_d)$) est conforme à la proposition. □

# 15 Parcours de graphe en largeur et applications

## 15.1 Parcours en largeur (*Breadth First Search*)

**Algorithm 28** `BFS`$(G = (S, \texttt{succ}),\ s_0)$

```
1:  done ← [ s_0 ]
2:  todo ← File_Vide
3:  todo.enfiler(s_0)
4:  while todo n'est pas vide do
5:      s ← todo.defiler()
6:      for s' ∈ succ(s) do
7:          if s' ∉ done then
8:              todo.enfiler(s')
9:              done ← done +  [ s' ]
10:         end if
11:     end for
12: end while
13: return done
```

## 15.2 Applications : graphe d'accessibilité et composantes connexes

### 15.2.1 Graphe d'accessibilité

Le parcours en largeur **permet générer le graphe d'accessibilité** des sommets accessibles depuis le sommet "racine" donné en argument ($s_0$) ; i.e. les sommets pour lesquels il existe un chemin depuis $s_0$.

### 15.2.2 Composantes connexes d'un graphe non orienté

D'une façon générale, le parcours en largeur **permet de déterminer les composantes connexes d'un graphe non orienté**. Pour cela, il suffit d'appliquer le sur-algorithme suivant :

**Algorithm 29** `Composantes_Connexes`$(G = (S, \texttt{succ}))$

```
1: done ← [ ]                                              ▷ Liste vide
2: todo ← S
3: while todo n'est pas vide do
4:     s ← todo.pop()
5:     new ← BFS(G, s_0)
6:     todo ← todo − new
7:     done.append(new)
8: end while
9: return done
```

Le nombre de tours de boucle est égal à la taille de la liste *done*, et correspond au nombre de composantes connexes.

## 15.3 Application : plus court chemin

Parcours en largeur peut aussi être utilisé pour **chercher chacun des plus courts chemins (en nombre d'arcs ou arêtes) entre la "racine"** $s_0$ **et chacun des autres sommets du graphe d'accessibilité depuis** $s_0$. Pour ce faire, il convient de stocker le prédécesseur de chaque sommet "généré".

L'algorithme suivant est une amélioration de `BFS`, donnant en outre : le tableau $\pi$ des prédécesseurs, associant à chaque sommet son prédécesseur, i.e. le sommet qui l'a fait entrer dans la file *done*.

**Algorithm 30** `BFS2`$(G = (S, \texttt{succ}),\ s_0)$

```
1:  π ← tableau de taille |S|
2:  ∀s ∈ S, π[s] ← None
3:  done ← [ s_0 ]
4:  todo ← File_Vide
5:  todo.enfiler(s_0)
6:  while todo n'est pas vide do
7:      s ← todo.defiler()
8:      for s' ∈ succ(s) do
9:          if s' ∉ done then
10:             todo.enfiler(s')
11:             done ← done + [ s' ]
12:             π[s'] ← s
13:         end if
14:     end for
15: end while
16: return (done, π)
```

L'algorithme suivant trouve un plus court chemin, s'il existe, pour aller de $s_0$ à $s_j$ (préalablement `BFS2` a été exécuté).

**Algorithm 31** `Plus_Cout_Chemin`$(G = (S, \texttt{succ}),\ s_0,\ s_j)$

```
1:  res ← [ ]                                   ▷ Liste vide
2:  s ← s_j
3:  if π[s] = None then
4:      return [ ]                              ▷ Pas de plus court chemin
5:  end if
6:  while True do
7:      if s = s_0 then
8:          res.append(s)
9:          return res
10:     end if
11:     res.append(s)
12:     s ← π[s]
13: end while
```

# 16 Problème de l'arrêt

**Définition** (Arrêt)**.**
Entrées :

1. `<Prog>` : le code source d'un programme `Prog` ;
2. `x` : une entrée pour `Prog`.

Sortie : `Prog(x)` s'arrête-t-il ?

**Théorème** (Turing)**.** Arrêt *est indécidable.*

*Preuve.* (Par l'absurde). On suppose qu'Arrêt est décidable ; i.e. il existe un programme, soit `Halt`, qui décide le problème de l'arrêt ; i.e., pour tout programme `Prog` de code source `<Prog>`, pour toute entrée `x` de `Prog` :

- `Prog(x)` s'arrête $\iff$ `Halt(<Prog>, x)` répond `Vrai` ;
- `Prog(x)` ne s'arrête pas $\iff$ `Halt(<Prog>, x)` répond `Faux`.

On considère le programme `Diagonale` ci-après :

```
Diagonale(y):
    Si Halt(y, y) = Vrai :
        effectuer une boucle infinie
    Sinon
        retourner "toto"
    Fin Si
```

Nous considérons l'exécution : `Diagonale(<Diagonale>)`.

(a) Cas 1 : `Halt(<Diagonale>, <Diagonale>) = Vrai`

D'après le code de `Diagonale`, il suit que `Diagonale(<Diagonale>)` ne s'arrête pas, donc que `Halt(<Diagonale>, <Diagonale>)` répond `Faux`.

(b) Cas 2 : `Halt(<Diagonale>, <Diagonale>) = Faux`

D'après le code de `Diagonale`, il suit que `Diagonale(<Diagonale>)` s'arrête, donc que `Halt(<Diagonale>, <Diagonale>)` répond `Vrai`.

□