Algorithmique avancée

Version 2.6

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)

```
if n = 0 then
    return True
else
    return False
end if
```

Complexité : $O(1)$.

2.2 Afficher les éléments d'un tableau

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

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

Complexité : $O(n)$.

2.3 Afficher les éléments positifs d'un tableau

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

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

Complexité : $O(n)$.

2.4 Retourner l'éléments maximum d'un tableau

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

$max \leftarrow t[0]$ ▷ on suppose $n > 0$
for $i \leftarrow 1, ..., n-1$ **do**
    if $t[i] \geq max$ **then**
        $max \leftarrow t[i]$
    end if
end for
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)

$max \leftarrow t[0]$ ▷ on suppose $n > 0$
$iMax \leftarrow 0$
for $i \leftarrow 1, ..., n-1$ **do**
    if $t[i] \geq max$ **then**
        $max \leftarrow t[i]$
        $iMax \leftarrow i$
    end if
end for
return $iMax$

Complexité : $O(n)$.

2.6 Retourner la somme des éléments d'un tableau

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

$res \leftarrow 0$
for $i \leftarrow 0, ..., n-1$ **do**
    $res \leftarrow res + t[i]$
end for
return res

Complexité : $O(n)$.

2.7 Rechercher un élément dans un tableau

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

```
for i ← 0, ..., n − 1 do
    if t[i] = x then
        return True
    end if
end for
return False
```

Complexité : $O(n)$.

3 Algorithmes sur matrices

3.1 Afficher les éléments d'une matrice

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

for $i \leftarrow 0, ..., n-1$ **do** ▷ parcours des lignes
    for $j \leftarrow 0, ..., m-1$ **do** ▷ parcours des colonnes
        `print`$(A_{i,j})$
    end for
    `print("\n")` ▷ échappement pour une nouvelle ligne
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 $n \times m)$

$C \leftarrow$ matrice $n \times m$
for $i \leftarrow 0, ..., n-1$ **do**
    for $j \leftarrow 0, ..., m-1$ **do**
        $C_{i,j} \leftarrow A_{i,j} + B_{i,j}$
    end for
end for
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 $n \times n$)

```
for i ← 0, ..., n − 1 do
    for j ← 0, ..., n − 1 do
        if i ≠ j and M_{i,j} ≠ 0 then
            return False
        end if
    end for
end for
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. 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)

$tmp \leftarrow t[i]$
$t[i] \leftarrow t[j]$
$t[j] \leftarrow tmp$

5.2 Tri par sélection

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

for $i \leftarrow 0, ..., n-2$ **do**
    $i_{min} \leftarrow$ `indice_min_sous_tab`$(t, i, n-1)$
    `echanger`(t, i, i_{min})
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:     **if** $t[i] < t[i_{min}]$ **then**
4:         $i_{min} \leftarrow i$
5:     **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).

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

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

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

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

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

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

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

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

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

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

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

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

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

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

$\square$

5.3 Tri à bulles

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

1: **for** $i \leftarrow n-1, ..., 1$ **do**
2:     **for** $j \leftarrow 0, ..., i-1$ **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 :

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

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

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

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

if $n = 0$ **then**
    return 1
else
    return $n \times$ `fact`$(n-1)$
end if

Complexité : $O(n)$.

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

1: $res \leftarrow 1$
2: **for** $i \leftarrow 1, ..., n$ **do**
3: $\quad res \leftarrow res \times i$
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`$(n \in \mathbb{N})$

if $n = 0$ **then**
    return 0
else if $n = 1$ **then**
    return 1
else
    return `fibo_rec`$(n-1)$ + `fibo_rec`$(n-2)$
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`$(n \in \mathbb{N})$

$F \leftarrow$ tableau de $n+1$ éléments
$F[0] \leftarrow 0$
$F[1] \leftarrow 1$
for $i \leftarrow 2, ..., n$ **do**
    $F[i] \leftarrow F[i-1] + F[i-2]$
end for
return $F[n]$

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

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

– Nombre d'additions : $O(n)$.

$\square$

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

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

$\square$

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 :

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

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)

if $n = 0$ **return** 1 **end if**

if $n \equiv 0 \mod 2$ **then**
    return `exp_rapide`($x.x$, $\frac{n}{2}$)
else
    return $x \times$ `exp_rapide`($x.x$, $\frac{n-1}{2}$)
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)

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

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

if $d > f$ **then return** -1 **end if** ▷ non trouvé
if $f = d$ **then**
    if $t[d] = x$ **then return** d **else return** -1 **end if**
end if
$m \leftarrow \lfloor \frac{d+f}{2} \rfloor$ ▷ partie entière
if $t[m] = x$ **then**
    return m
end if
if $t[m] < x$ **then**
    return `dicho`$(t, m+1, f, x)$
else
    return `dicho`$(t, d, m-1, x)$
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,

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

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 :

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

$\square$

11 Tri fusion

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

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

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

$res \leftarrow$ `[ ]`
while **not** $\Big($`est_vide`(lst_1) **and** `est_vide`(lst_2)$\Big)$ **do**
    if `est_vide`(lst_1) **then**
        $res \leftarrow res + lst_2$
        $lst_2 \leftarrow$ `[ ]`
    else if `est_vide`(lst_2) **then**
        $res \leftarrow res + lst_1$
        $lst_1 \leftarrow$ `[ ]`
    else if `head`(lst_1) $\leq$ `head`(lst_2) **then**
        $res \leftarrow res +$ `[ head(`lst_1`) ]`
        $lst_1 \leftarrow$ `tail`(lst_1)
    else
        $res \leftarrow res +$ `[ head(`lst_2`) ]`
        $lst_2 \leftarrow$ `tail`(lst_2)
    end if
end while
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 que l'algorithme ne termine pas ($\forall i \in \mathbb{N}$, $P(i) > 0$).

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

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

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 γ 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 :

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

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

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

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

$\square$

12 Représentation des graphes

12.1 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 $n - 1$.

12.2 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 :

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

12.3 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 :

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

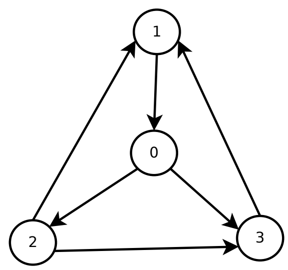
Autrement dit, $\forall s \in S$, $\texttt{succ}(s)$ est l'ensemble des sommets adjacents à s.

12.4 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 représentatons par matrice et liste d'adjacence sont données ci-après.

Matrice d'adjacence	**Liste d'adjacence**
$\begin{pmatrix} 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 \end{pmatrix}$	0 $\{2,3\}$ 1 $\{0\}$ 2 $\{1,3\}$ 3 $\{1\}$

12.5 Espace mémoire

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

12.6 Complexité de quelques opérations

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

12.7 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.

12.8 Relation entre sommets adjacents et arêtes

Théorème. *Si le graphe G est orienté,*

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

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

$$\bigcup_{s \in S} \mathtt{succ}(s) = S$$

et

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

□

Théorème. *Si le graphe G est orienté,*

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

Preuve. D'une part,

$$\bigcup_{s \in S} \mathtt{succ}(s) = S$$

D'autre part, quel que soient s et s' de S, et $a \in A$,

$$a = (s, s') \iff a \in \mathtt{succ}(s) \cap \mathtt{succ}(s')$$

(chaque arête est comptée exactement 2 fois). □

13 Arbres

13.1 Arbres – arbres binaires

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) $\mathfrak{p}(s) = 0$, si s est nœud nul ;

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

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

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

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

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 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 de $O(\log n)$.*

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

$$\mathfrak{p}(A) = 1 \iff n = 1 = 2^2 - 1$$

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

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

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

$$\mathfrak{p}(A) = q \iff n = \sum_{i=0,\dots,q-1} 2^i = 2^q - 1$$

On a

$$\mathfrak{p}(A) = q+1 \iff n = \sum_{i=0,\dots,q-1} 2^i + 2.2^{q-1}$$

D'où,

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

On a ainsi, $\forall k \in \mathbb{N}^*$,

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

Soit

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

□

Algorithm 24 `infixe` ($s \in \mathcal{A}$)

```
if est_nul(s) then
    return
else
    print(s)
end if
infixe(s_g)
infixe(s_d)
```

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

13.2 Arbre binaire de recherche

13.2.1 Définition

Définition (ABR)**.** Un arbre binaire de recherche (ABR) est un arbre binaire valué, qui est soit arbre vide ; soit 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.

13.2.2 Recherche dans un ABR

Algorithm 25 `recherche_ABR` ($s \in \mathcal{A}$, $x \in V$: valeur recherchée)

```
if est_nul(s) then
    return False
else if s.val = x then
    return True
else if s.val > x then
    return recherche_ABR(s.f_g, x)
else
    return recherche_ABR(s.f_d, x)
end if
```

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

Preuve. Admis. □

13.3 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 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. □

14 Parcours de graphes

14.1 Parcours en largeur (*Breadth First Search*)

Algorithm 26 BFS($G = (S, \texttt{succ}),\ s_0$)

```
done ← [ s0 ]
todo ← File_Vide
todo.enfiler(s0)
while todo n'est pas vide do
    s ← todo.defiler()
    for s' ∈ succ(s) do
        if s' ∉ done then
            todo.enfiler(s')
            done ← done + [ s' ]
        end if
    end for
end while
return done
```

Permet de trouver le plus court chemin entre deux sommets (dans un graphe non pondéré).