Chapitre I

Concepts préliminaires

I.1. Mesure des algorithmes : O-notation

I.1.1 Introduction

L'efficacité d'un programme se mesure principalement à travers sa consommation des ressources Temps (d'exécution) et Espace (mémoire).

Nous nous intéressons ici à l'analyse du temps d'exécution des programmes en utilisant la O-notation.

Pour un problème donné, il peut exister plusieurs algorithmes différents de résolution. L'implémentation d'un des algorithmes à l'aide d'un langage de programmation donne un programme. En exécutant ce programme sur une machine donnée avec différents exemplaires du problème initial, on peut mesurer l'évolution de son temps d'exécution en fonction de la taille de l'exemplaire.

Ainsi l'implémentation de deux algorithmes (A1 et A2) différents pour résoudre un même problème, donne lieux à deux programmes (P1 et P2) pouvant avoir des temps d'exécution très différents pour un même exemplaire.

A travers l'exemple ci-dessous, nous allons montrer l'intérêt d'un algorithme efficace.

Soient deux algorithmes différents pour calculer le terme U_n de la suite de Fibonacci:

A1:
```
i ← 1; j ← 0;
Pour k←1,n
    j ← i + j;
    i ← j - i;
Fp;
Retourner j
```

A2 :
```
Fib(n:entier) : entier
Si ( n < 2 )
    Retourner 1;
Sinon
    Retourner  Fib(n-1) + Fib(n-2)
Fsi
```

L'implémentation, en langage C (Compilateur gcc, version 4.6.3), de ces deux algorithmes sur un PC doté d'un processeur i3 et de 4GO de Ram, donne lieux à deux programmes P1 et P2, dont les temps d'exécution ont été mesurés pour différentes valeurs de n.
Le tableau suivant résume les temps d'exécution de P1 et P2 en fonction de n :

n	***40***	***42***	***44***	***45***	***48***	***50***	***52***	***100***	***2 . 10^5***	***4 . 10^5***	***10^6***
P1	<1 ms	<1 ms	<1 ms	<1 ms	<1 ms	<1 ms	<1 ms	< 1 ms	3 ms	6 ms	13 ms
P2	1,4 s	3,6 s	9,5 s	15,3 s	65,4 s	169 s	443 s	> 160 000 années (estimation)	-	-	-

Le temps d'exécution de P1 est très faible (de l'ordre des millisecondes) et augmente linéairement avec n (on le remarque à partir de n = 200 000) alors que celui de P2 augmente beaucoup plus rapidement (croissance exponentielle). P2 a une complexité beaucoup plus forte que celle de P1. Le temps d'exécution du programme P2 pour la valeur n=100, est une estimation en utilisant la méthode hybride d'analyse (voir plus loin dans ce chapitre). C'est-à-dire, on trouve la forme analytique du temps d'exécution (O-notation), puis on fixe les valeurs des constantes cachées, en réalisant des tests sur quelques valeurs de n.

Clairement, A2 est un mauvais algorithme, car son implémentation donne des programmes que l'on ne peut utiliser pratiquement que sur de petites valeurs de n.

I.1.2 Définition

O(f(n)) est un ensemble de fonctions, dont la croissance est inférieure ou égale à celle de f(n).
Plus formellement, on a :
$O(f(n)) = \{ t:\mathbb{N}\rightarrow\mathbb{R}^* / \exists c \in \mathbb{R}^+, \exists n_0 \in \mathbb{N}, \forall n \geq n_0 : t(n) \leq c\, f(n) \}$

- $\mathbb{N}$: l'ensemble des entiers naturels
- $\mathbb{R}^*$: l'ensemble des réels positifs ou nuls
- $\mathbb{R}^+$: l'ensemble des réels strictement positifs

O(f(n)) est l'ensemble de toutes les applications t inférieures à c*f(n)

La croissance de f(n) est un majorant de l'ensemble O(f(n)).

Exemples :

- $n^2 \in O(n^2)$
- $n^2 \in O(n^3)$ car $n^2 \leq c*n^3$ pour $c=1$ et $n \geq 0$.
- $2^{n+1} \in O(2^n)$ car $2^{n+1} = 2*2^n \leq c\ 2^n$ pour $c \geq 2$ et $n \geq 0$.
- $3^n \notin O(2^n)$ car $(3/2)^n$ est croissante. Donc il n'existe pas de constante supérieure ou égale à $(3/2)^n$ pour toute valeur de n.
- $(n+1)! \notin O(n!)$ car $(n+1)!/n! = n+1$ qui est aussi croissante.
- $\log_b n \in O(\log_a n)$ car $\log_b n=(1/\log_a b)*\log_a n$ et $(1/\log_a b)$ est une constante.

Comportement asymptotique :

Si le temps d'exécution d'un algorithme A est une fonction appartenant à O(f(n)), on dit que A est de complexité O(f(n)).
Donc, pour des valeurs de n tendant vers l'infini, on s'attend à ce que le temps d'exécution de A, ne croit pas plus rapidement que f(n).

Principe d'invariance :

La complexité d'un algorithme est indépendante de son implémentation (choix de la machine, du langage, ...). En d'autres termes, deux implémentations différentes d'un même algorithme ne peuvent différer en efficacité que par une constante multiplicative près. c1*f(n) pour l'un et c2*f(n) pour l'autre.

Propriétés de la O-notation

a) $O(f(n)) = O(g(n))$ ssi $f(n) \in O(g(n))$ et $g(n) \in O(f(n))$
b) $O(f(n)) \subset O(g(n))$ ssi $f(n) \in O(g(n))$ et $g(n) \notin O(f(n))$
c) $O(f(n)+g(n)) = O(\max(f(n),g(n)))$
d) Si $\lim (f(n)/g(n)) = k$ $(k>0)$ Alors $O(f(n)) = O(g(n))$
e) Si $\lim (f(n)/g(n)) = 0$ Alors $O(f(n)) \subset O(g(n))$

Quelques complexités usuelles

Voici quelques complexités usuelles que l'on peut rencontrer dans l'analyse des algorithmes. Elles sont classées de la plus faible à la plus forte :

$O(\log n)$	→	logarithmique
$O(n)$	→	linéaire
$O(n^2)$	→	quadratique
$O(n^k)$	→	polynomiale (avec k une constante)
$O(a^n)$, $O(n!)$, $O(n^n)$, ...	→	exponentiel (avec a une constante)

Autres notations

$\Omega(f(n))$: l'ensemble de toutes les applications $t \geq c*f(n)$
f(n) est un minorant de l'ensemble. La croissance du temps d'exécution de l'algorithme est alors minorée par celle de f(n).

$\Theta(f(n)) = O(f(n)) \cap \Omega(f(n))$
La croissance du temps d'exécution de l'algorithme est alors exactement celle de f(n).

Opérations élémentaires

Une opération élémentaire est une opération (instruction ou bloc d'instructions) du programme dont le temps d'exécution est indépendant de la taille (n) de l'exemplaire du problème à résoudre.
Généralement, toutes les opérations simples (affectation, E/S d'une unité de donnée, évaluation d'expression simples, ...) ainsi que les comparaisons (<, >, ...) ont un temps d'exécution indépendant de la taille du problème à résoudre.

I.1.3 Règles de calcul de la complexité

a) Toute opération élémentaire a une complexité O(1) car son temps d'exécution est constant.

b) La complexité d'une séquence de deux blocs d'instructions : M1 (de complexité O(f(n))) et M2 (de complexité O(g(n))), est égale à la plus grande des complexités des deux blocs : $O(\max(f(n), g(n)))$

c) La complexité d'une conditionnelle (Si cond Alors M1 Sinon M2 Fsi) est le maximum entre les complexités de la condition du SI (cond), de la partie Alors (M1) et celle de la partie Sinon (M2). Cette simplification s'effectue dans une analyse en pire cas. C'est-à-dire que l'on supposera que le bloc le plus complexe (entre la partie Alors et la partie Sinon) sera

toujours exécuté, d'où une séquentielle entre l'évaluation de la condition (cond) et le bloc le plus complexe (M1 ou M2).

d) La complexité d'une boucle est égale à la somme sur toutes les itérations de la complexité du corps de la boucle

Exemple :

Donnons la complexité de l'algorithme du tri par sélection.

```
1       Pour i ← 1 , n-1
2           Pour j ← i+1 , n
3               Si T[i] > T[j]
4                   Tmp ← T[i];
5                   T[i] ← T[j];
6                   T[j] ← Tmp;
7               Fsi
8           Fpour
9       Fpour
```

Chaque itération de la boucle interne (Pour j ← i+1 , n … Fpour) prend au maximum un temps égal à une constante a. Cette valeur inclue le temps du test de la ligne 3, le temps des trois affectations des lignes 4, 5 et 6 ainsi que celui des instructions de contrôle de la boucle: (test, incrémentation de j, …)

La boucle interne fait (n-i) itérations, donc son temps d'exécution est borné par :

$b + (n-i)\ a$

où b désigne le temps d'initialisation de la boucle.

Ainsi chaque itération de la boucle externe (Pour i ← 1 , n-1... Fpour), prend un temps borné par :

$c + b + (n-i)\ a$

avec c une constante englobant le temps des instructions de contrôle de la boucle.

La boucle externe fait n-1 itérations, donc son temps d'exécution est borné par :

$$d + \sum_{i=1,n-1} [\ c + b + (n-i)\ a\ ] \quad = \quad d + (n-1)\ (c+b) + (n-1)\ n\ a\ /\ 2$$

d étant une constante représentant le temps de l'initialisation de la boucle externe.

Le temps d'exécution total est donc borné par :

$(a/2)\ n^2 + (c+b-a/2)\ n + (d-c-b)$

Donc le programme est en $O(n^2)$ car a/2 est une valeur positive.

I.1.4 Complexité des algorithmes récursifs

Dans le corps d'une fonction ou procédure récursive, existe toujours un ou plusieurs appels à cette même fonction ou procédure. Ainsi pour évaluer le temps d'exécution (T(n)) d'un appel à cette fonction ou procédure sur un exemplaire de taille n, on est amené à résoudre une équation de

récurrence.

Les trois approches suivantes peuvent être utilisées pour la résolution d'une telle équation :

a) éliminer la récurrence par substitution de proche en proche
b) deviner une solution et la démontrer par récurrence
c) utiliser la solution de certaines équations connues

Dans ce qui suit, nous allons présenter l'application de ces trois approches sur des exemples.

a) Application de la première approche « par substitution »

Soit la fonction récursive suivante :

```
Fact(n:entier) : entier
Si n ≤ 1
        Fact ← 1
Sinon
        Fact ← n * Fact(n-1)
Fsi
```

Posons T(n) le temps d'exécution nécessaire pour un appel à Fact(n).

Puisque le corps de la fonction Fact est une instruction conditionnelle, T(n) est le max entre :

- la complexité du test (n ≤ 1) qui est en O(1)
- la complexité de la partie Alors: (Fact ← 1) qui est aussi en O(1)
- la complexité de la partie Sinon: (Fact ← n * Fact(n-1)) dont le temps d'exécution dépend de T(n-1) (pour le calcul de Fact(n-1))

On peut donc écrire :

T(n) = a	si n ≤ 1	et,
T(n) = b+T(n-1)	sinon	(n > 1).

La constante 'a' englobe le temps du test (n ≤ 1) et le temps de l'instruction : Fact ← 1
La constante 'b' englobe :

- le temps du test (n ≤ 1)
- le temps de l'opération * entre n et le résultat de Fact(n-1)
- et le temps de l'instruction d'affectation

T(n-1) (le temps nécessaire pour le calcul de Fac(n-1)) sera calculé (récursivement) avec la même décomposition.

Pour calculer la solution générale de cette équation, on peut procéder par substitution (la première approche) :

```
T(n) = b + T(n-1)       = b + [b+T(n-2)]
     = 2b + T(n-2)      = 2b + [b+T(n-3)]
     = ...
     = ib + T(n-i)      = ib + [b+T(n-i+1)]
     = ...
     = (n-1)b + T(n-n+1) = nb - b + T(1) = nb - b + a
T(n) = nb - b + a
```

Donc Fact est en O(n) car b est une constante positive.

b) Application de la deuxième approche « par vérification »

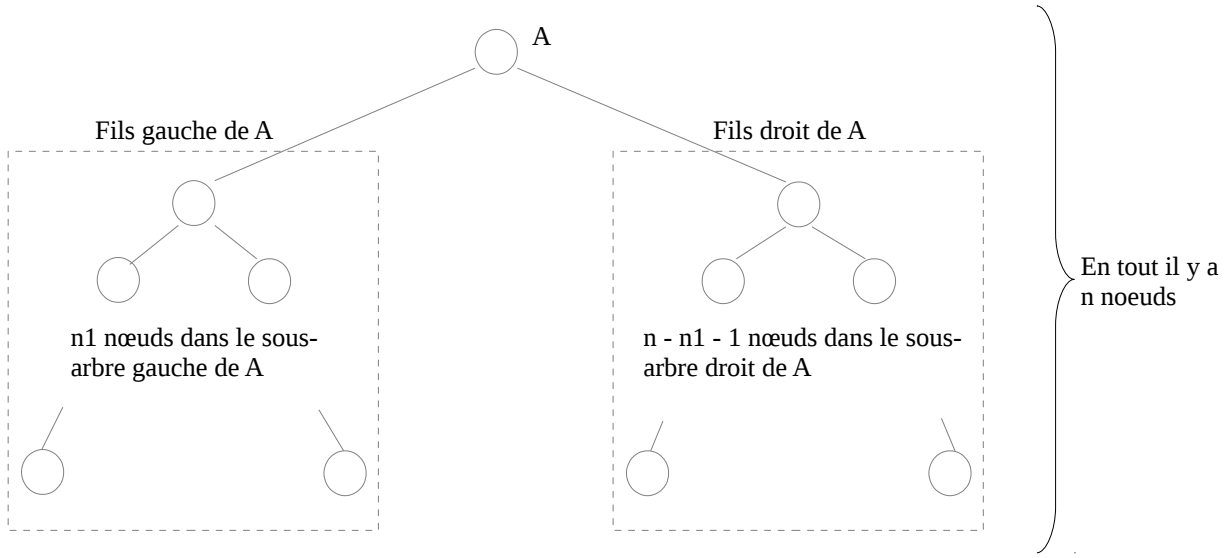
Soit la procédure récursive du parcours *Inordre* dans un arbre binaire :

```
Inordre( A:arbre )
Si A <> NIL
        Inordre( fg(A) );
        ecrire( info(A) );
        Inordre( fd(A) );
Fsi
```

Les fonctions fg(), fd(), et info() sont des accesseurs en temps constant, aux différents champs d'un nœud (le fils gauche, le fils droit et la valeur du nœud). Elles ont donc une complexité de O(1).

La taille du problème (n) est le nombre de nœuds dans l'arbre de racine A.

Posons T(n) le temps nécessaire pour parcourir un arbre contenant n nœuds.



Supposons que le sous arbre gauche de A contienne n1 nœuds, avec n1 < n (donc le sous arbre droit doit renfermer n – n1 – 1 nœuds), nous pouvons alors écrire l'équation de récurrence suivante :

T(n) = a si n=0 (le cas où A = NIL) et

T(n) = b + T(n1) + T(n-n1-1) si n>0 (le cas où A # NIL, avec n1 < n)

la constante 'a' représente le temps d'exécution du test (A#NIL),

la constante 'b' représente le temps nécessaire pour le test et l'exécution de la partie Sinon

T(n1) représente le temps d'exécution de l'appel Inordre(fg(A)), avec n1 < n.

T(n - n1 - 1) représente le temps d'exécution de l'appel Inordre(fd(A)), avec n-n1-1 < n.

Pour résoudre cette équation de récurrence, supposons que cette procédure soit en O(n) , c'est à dire T(n) est de la forme : dn + c (avec d et c des constantes à déterminer).
Vérifions cette hypothèse par récurrence :

T(0) = a d'après l'équation de récurrence et
T(0) = c d'après l'hypothèse de récurrence, donc il faut que c = a
T(1) = b + T(0) + T(0) = b+2a d'après l'équation de récurrence (car pour n=1 les sous-arbres gauche et droit sont vides, c'est à dire n1=0 et n-n1-1=0)
T(1) = d + a d'après l'hypothèse de récurrence, donc d = b+a

L'hypothèse de récurrence est donc T(n) = (b+a)n +a

Supposons cette hypothèse est vraie jusqu'à m-1 et vérifions qu'elle reste vraie pour m:
T(m) = b + T(m1) + T(m-m1-1) d'après l'équation de récurrence.

Comme m1 < m et (m-m1-1) < m on a alors :
T(m1) = (b+a)m1 + a et T(m-m1-1) = (b+a) (m-m1-1) + a d'après l'hypothèse de récurrence.

Donc T(m) = b + [(b+a)m1 + a] + [(b+a) (m-m1-1) + a]
= b + (b+a)m1 + a + (b+a)m – (b+a)m1 - (b+a) + a
= (b+a)m + a donc l'hypothèse de récurrence est vraie quelque soit n.

c) Application de la troisième approche « par identification à des équations connues »

Nous présentons, dans cette approche, l'utilisation des équations homogènes et non homogènes.

- Cas des équations homogènes

Une équation de récurrence est dite homogène si elle a la forme suivante:

$$a_0 t_n + a_1 t_{n-1} + ... + a_k t_{n-k} = 0 \qquad (1)$$

Pour trouver sa solution générale, on procède comme suit:
(i) Etablir son équation caractéristique (un polynôme de degré k) :
$a_0 x^k + a_1 x^{k-1} + ... a_k = 0$ (soient r_1, r_2, ... r_k ses racines)

(ii) Si toutes les racines r_i sont distinctes, alors la solution générale de (1) est donnée par :

$$t_n = C_1 r_1^n + C_2 r_2^n + ... C_k r_k^n \qquad (2)$$

où les C_i sont des constantes que l'on peut déterminer avec les conditions initiales de l'équation de récurrence.

(iii) Pour chaque solution multiple r_j de multiplicité m (apparaît m fois comme solution de l'équation caractéristique), on remplace dans (2) le terme $C_j r_j^n$ par la somme :
$C_{j1} r_j^n + C_{j2} n r_j^n + C_{j3} n^2 r_j^n ... C_{jm} n^{m-1} r_j^n$ pour former la solution générale de (1).

Exemples:
1) Soit : $t_n - 3\, t_{n-1} - 4\, t_{n-2} = 0$ (forme générale)
$t_0 = 0$, $t_1 = 1$ (conditions initiales)
Son équation caractéristique est : $x^2 – 3\, x – 4 = 0$
L'équation a 2 racines distinctes : $r_1 = -1$ et $r_2 = 4$
Donc la solution générale est :

$t_n = C_1(-1)^n + C_2\ 4^n$ qui est $O(4^n)$
En appliquant les conditions initiales on trouve C_1=-1/5 et C_2=1/5

2) Soit : $t_n - 5\ t_{n-1} + 8\ t_{n-2} - 4\ t_{n-3} = 0$ (forme générale)
$t_0 = 0$, $t_1 = 1$, $t_2 = 2$ (conditions initiales)
Son équation caractéristique est : $x^3 - 5\ x^2 + 8\ x - 4 = 0$ ou alors : $(x-1)\ (x-2)^2 = 0$
L'équation a 3 racines : 1, 2 et 2 (2 est une solution multiple de multiplicité 2)
Donc la solution générale est :
$t_n = C_1 1^n + C_2 2^n + C_3 n 2^n$
Les conditions initiales donnent : $C_1 = -2$, $C_2 = 2$ et $C_3 = -1/2$

- Cas des équations non homogènes:

Une équation de récurrence est dite non homogène si elle a la forme suivante:

$$a_0 t_n + a_1 t_{n-1} + ... + a_k t_{n-k} = b_1^n P_1(n) + b_2^n P_2(n) + b_3^n P_3(n) + ... \qquad (3)$$

où les b_i sont des constantes distinctes et les $P_i(n)$ des polynômes en n de degré d_i

La résolution d'une telle équation suit le même schéma que celui des équations homogènes en partant de l'équation caractéristique suivante :

$$(a_0 x^k + a_1 x^{k-1} + ... a_k)\ (x - b_1)^{d1+1}\ (x - b_2)^{d2+1}\ (x - b_3)^{d3+1} ... = 0$$

Exemple:

Soit : $t_n - 2t_{n-1} = n + 2^n$ et $t_0 = 0$

C'est une équation non homogène avec :
b1 = 1 , P1(n) = n , d1 = 1
b2 = 2 , P2(n) = 1 , d2 = 0

Son équation caractéristique est donc : $(x-2)\ (x-1)^2\ (x-2) = 0$ dont les racines sont : 2, 1, 1, 2
Donc les racines 1 et 2 sont, chacune, de multiplicité 2.

La solution générale est alors donnée par : $t_n = C_1 1^n + C_2 n 1^n + C_3 2^n + C_4 n 2^n$

I.1.5 Analyse hybride

L'analyse de la complexité d'un algorithme, telle que présentée ci-dessus, permet de déterminer la forme générale de la fonction T(n) du temps d'exécution de toute implémentation possible de cet algorithme.

T(n) nous donne des informations sur la vitesse avec laquelle le temps d'exécution augmente lorsque la taille de l'exemplaire (n) évolue.

L'analyse hybride permet de trouver le temps d'exécution d'un programme implémentant un

algorithme donné.

Par exemple la complexité de l'algorithme du tri par sélection (voir section 1.3 ci-dessus) est $O(n^2)$.
$T(n) = (a/2)\ n^2 + (c+b-a/2)\ n + (d-c-b) = C1\ n^2 + C2\ n + C3$

Nous pouvons donc conclure que l'accroissement du temps d'exécution de n'importe quelle implémentation de cet algorithme, sera toujours proportionnel à n^2 (croissance quadratique).

Les constantes cachées C1, C2 et C3 changent d'une implémentation à une autre. Elles dépendent principalement de la qualité du compilateur (optimisation du code), et de la puissance de la machine (processeur, mémoire, …) sur laquelle le programme sera exécuté.

Si on veut connaître le temps d'exécution d'une implémentation donnée de cet algorithme, quelque soit la valeur de n, on doit trouver la valeur de chacune de ces trois constantes en procédant par des tests pratiques. Il faut par exemple, au moins trois essais pour obtenir trois temps d'exécution en fonction de trois valeurs de n différentes. On pourra alors résoudre un système de trois équations à trois inconnues (C1, C2 et C3). Une autre méthode plus précise, pour trouver les constantes cachées, serait d'utiliser les techniques d'approximation de courbe à travers un nuage de points.

Sur la même machine et le même compilateur que ceux utilisés dans l'exemple d'introduction, nous avons obtenu les résultats suivants :

Pour n = 50 000, le temps est de 12.36 s
Pour n = 100 000, le temps est de 50.57 s
Pour n = 200 000, le temps est de 198.01 s

Le système d'équations à résoudre est alors :

$C1\ (50\ 000)^2 + C2\ (50\ 000) + C3 = 12.36$
$C1\ (100\ 000)^2 + C2\ (\ 100\ 000) + C3 = 50.57$
$C1\ (200\ 000)^2 + C2\ (\ 200\ 000) + C3 = 198.01$

Les solutions sont : $C1 = 4.8\ 10^{-9}$ $C2 = 4\ 10^{-5}$ $C3 = -2$

Donc pour cette implémentation, le temps d'exécution (en secondes) est estimé par :

$$T(n) = 4.8\ 10^{-9}\ n^2 + 4\ 10^{-5}\ n - 2$$

Nous pouvons alors estimer le temps d'exécution pour de grandes valeurs de n :

Pour n = 10 000 000, le temps est estimé à 480 398 s (> 5 jours)
Pour n = 100 000 000, le temps est estimé à 48 003 998 s (> 18 mois)

Cet algorithme de tri (tri par sélection) est l'un des plus mauvais.

Nous l'avons comparé avec d'autres algorithmes de tri (tri par insertion, tri par fusion et tri de Hoare ou Quicksort) que nous avons aussi implémenté sur la même machine. Le tableau ci-dessous résume la comparaison :

n	Tri par sélection $O(n^2)$	Tri par insertion $O(n^2)$	Tri par fusion O(n*log n)	Tri de Hoare (qsort) O(n*log n)
50 000	12.36 s	3.5 s	21 ms	17 ms
100 000	50.57 s	14.3 s	32 ms	35 ms
200 000	198.01 s	56.8 s	65 ms	53 ms
1 000 000		1436.5 s	0.38 s	0.31 s
2 000 000			1.24 s	1.38 s
4 000 000			2.42 s	2.71 s
8 000 000			3.50 s	3.26 s
10 000 000	**> 5 jours** (estimation)		**4.23 s**	**4.56 s**
20 000 000			9.18 s	7.11 s
40 000 000			18.31 s	15.4 s
80 000 000			38.10 s	31.36 s
100 000 000	**> 18 mois** (estimation)		**48.51 s**	**39.45 s**

Pour l'implémentation du dernier algorithme (Tri de Hoare) nous avons utilisé la fonction 'qsort' de la librairie standard du langage C.

Pour le tri par fusion, nous avons implémenté la version récursive utilisant un tableau temporaire pour réaliser la fusion des deux moitiés du tableau initial. L'espace occupé par cette implémentation est donc de 2n (deux tableaux de taille n chacun)[1].

Pour ce qui est de l'estimation du temps d'exécution du programme récursif (P2) pour le calcul du terme U_n de la suite de Fibonacci (voir au début du chapitre l'exemple d'introduction), commençant d'abord par trouver la forme analytique du temps d'exécution (la O-notation) en calculant la complexité de l'algorithme A2 :

```
A2 :    Fib( n:entier ) : entier
        Si ( n < 2 )
                Retourner 1
        Sinon
                Retourner Fib(n-1) + Fib(n-2)
        Fsi
```

Posons T(n) le temps nécessaire pour évaluer un appel à Fib(n).

On peut alors écrire l'équation de récurrence suivante :

$T(n) = a$ (si $n<2$)

$T(n) = b + T(n-1) + T(n-2)$ (sinon)

1 Voir le chapitre suivant (Diviser pour Résoudre) pour le code de cette procédure.

avec :

a : le temps d'exécution nécessaire au test (n<2) et à l'exécution de l'instruction 'retourner 1'.
b : le temps nécessaire au test (n<2), à l'opération + et à l'instruction 'Retourner …'
T(n-1) : le temps nécessaire au premier appel Fib(n-1)
T(n-2) : le temps nécessaire au deuxième appel Fib(n-2)

C'est une équation non homogène : $t_n - t_{n-1} - t_{n-2} = b$

Son équation caractéristique est (x2 - x - 1) (x - 1) = 0
Les trois racines sont : (1+√5)/2 , (1-√5)/2 et 1
où (1+√5)/2 ≈ 1.62 et (1-√5)/2 ≈ -0.62

Donc la solution générale est $T(n) = C1 * 1.62^n - C2 * 0.62^n + C3$, donc $O(1.62^n)$.

A travers trois tests, on peut trouver les valeurs des constantes C1, C2 et C3.

Les valeurs de C2 et C3 n'étant pas très importantes pour de grandes valeurs de n car la quantité 1.62^n croît très rapidement. Donc pour simplifier le travail, on peut juste chercher la valeur de C1 pour avoir une estimation assez correcte du temps d'exécution.

Un seul test suffit donc :

$T(50) = C1 * 1.62^{50} = 169$ s

La valeur de C1 est donc = $5.65\ 10^{-9}$

Vérifions avec les autres mesures effectuées :

n	**Temps mesuré par les tests sur PC**	**Temps estimé par la formule ($5.65\ 10^{-9} * 1.62^n$)**
40	1.4 s	1.4 s
42	3.6 s	3.6 s
50	169 s	169 s
52	443 s	443.4 s
53	715.3 s	748.4 s
55	35 min	32 min
60	Non effectué	5 h 50 min
70	Non effectué	Plus de 30 jours
80	Non effectué	Plus de 10 années
100	Non effectué	Plus de 160 000 années
1000	Non effectué	**Plus de 10^{184} milliard d'années**

Il est à noter que pour de grandes valeurs de n, le résultat U_n dépasse largement le plus grand nombre manipulable par les opérations élémentaires. Dans notre cas, nous avons contourné ce problème en travaillant avec le modulo dans les additions. Le résultat retourné (U_n) est bien entendu incorrect, mais cela permet quand même d'évaluer le temps d'exécution induit uniquement par les appels récursifs. Normalement, le temps d'exécution total est encore plus grand et doit

prendre aussi en considération le temps nécessaire pour les opérations d'addition et de comparaison sur de tels grands nombres (généralement en O(n) pour des représentations en tableaux ou en listes de chiffres).

La complexité de l'algorithme itératif (A1) pour le calcul du terme U_n de la suite de Fibonacci, est linéaire (O(n)), car la boucle 'pour' fait exactement n itérations.

I.2. Graphes et algorithmes de parcours

I.2.1 Définitions et terminologie

Les graphes sont des modèles pour représenter des relations entre objets (graphe orienté) ou des relations symétriques entre eux (graphe non orienté)

Graphe orienté :

Un graphe orienté G est un ensemble de sommets pouvant être connectés par des arcs.
G = (S,A) où S est l'ensemble des sommets et A l'ensemble d'arcs.

Les sommets d'un graphe peuvent par exemple représenter des objets et les arcs des relations entre objets.

S'il existe un arc entre les sommets u et v (u → v) alors v est un sommet adjacent (ou voisin) au sommet u.

Dans la plupart des applications, on associe de l'information aux nœuds et/ou aux arcs. On parlera alors d'un graphe étiqueté.

Un chemin dans le graphe G=(S,A) est la séquence $\{v_1, v_2, ...v_n\}$ de sommets appartenant à S tels que les arcs ($v_1 \rightarrow v_2$, $v_2 \rightarrow v_3$, ..., $v_{n-1} \rightarrow v_n$) existent dans A. Sa longueur est le nombre d'arcs qui le composent. Dans le cas d'un graphe orienté étiqueté, le poids du chemin et la somme des poids des arcs qui le composent.

Un chemin est dit simple si tous ses sommets sont distincts.

Un circuit est un chemin de longueur > 0 tel qu'il commence et finit au même sommet.

Une composante fortement connexe d'un graphe orienté G est un sous ensemble de ce graphe composé de l'ensemble maximal de nœuds dans lequel il existe un chemin de tout nœud de l'ensemble vers chaque autre nœud de l'ensemble.
Formellement, soit G=(S,A) un graphe. On peut partitionner S en classes d'équivalence : S1, S2, ...Sr
v et w appartiennent au même ensemble S_i si et seulement si, il existe un chemin de v à w et un chemin de w à v.
Soit T_i l'ensemble des arcs avec les têtes et les queues dans S_i.
Les graphes $G_i=(S_i, T_i)$ sont appelés les composantes fortement connexes de G.

Un graphe est dit fortement connexe s'il a une seule composante fortement connexe.

Une arborescence est un graphe orienté ou il existe un sommet particulier 'r' relié à tout autre sommet du graphe par un chemin unique commençant par le sommet 'r'.

Graphe non orienté :

Un graphe non orienté G est un ensemble de sommets pouvant être connectés par des arêtes.
G = (S,A) où S est l'ensemble des sommets et A l'ensemble d'arêtes.
Chaque arête (u, v) représente en fait les deux arcs u→v et v→u.

On parlera de cycle au lieu de circuit, et de chaîne au lieu de chemin.

Un graphe non orienté est connexe si tous ses sommets sont connectés. Sinon chaque groupe de sommets connectés forme une composante connexe du graphe.

Un arbre est un graphe non orienté connexe sans cycle.

I.2.2 Représentation mémoire

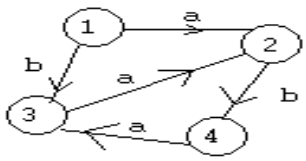
a) Par matrice associée

Si S = {v1, v2, ..., vn} est l'ensemble des sommets, le graphe est représenté par une matrice A[n, n] de booléens tels que A[i, j] est vrai s'il existe un arc de vi vers vj. Si le graphe est étiqueté, A[i, j] représentera la valeur de l'arc.

b) Tableaux de listes linéaires chaînées (Liste d'adjacence)

Le graphe est représenté par un tableau Tete[1..n] où Tete[i] est un pointeur vers la liste des sommets adjacents à vi.

Exemple :



Graphe étiqueté

	1	2	3	4
1		a	b	
2				b
3		a		
4			a	

Matrice

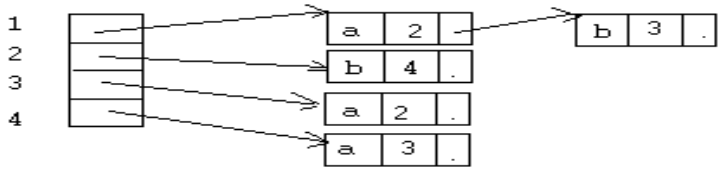


Tableau de listes

I.2.3 Parcours d'un graphe

a) Depht First Search : Parcours en Profondeur

C'est un type de parcours très utilisé sur les graphes : " Depht First Search " que l'on peut traduire par " Recherche en profondeur d'abord".
C'est la généralisation du parcours Préordre sur les structures arborescentes.
Beaucoup d'algorithmes sur les graphes utilisent ce type de parcours que nous notons DFS par la suite.

Le principe est le suivant :

Initialement tous les nœuds sont marqués "non visités".

Choisir un nœud v de départ et le marquer "visité". Chaque nœud adjacent à v, non visité, est à son tour visité en utilisant DFS récursivement. Une fois tous les nœuds accessibles à partir de v ont été visités, la recherche initiée au niveau du sommet v (DFS(v)) est complète.

Si certains nœuds du graphe restent encore "non visités", sélectionner un comme nouveau nœud de départ et répéter le processus jusqu'à ce que tous les nœuds soient visités.

Nous obtenons ainsi l'algorithme suivant :

```
DFS(v):
  Marq[v] ← "visité"
  Pour chaque nœud w adjacent à v:
        Si Marq[w]= "non visité"
              DFS(w)
        Fsi
  Fpour
```

Avec l'initialisation :

```
Pour chaque sommet v
        Marq[v] ← "non visité"
Fpour
```

Et comme programme appelant :

```
Pour chaque sommet v
        Si Marq[v]= "non visité": DFS(v) Fsi
Fpour
```

L'algorithme construit une forêt (implicite) de recouvrement des recherches.

On peut donner une version itérative du parcours en profondeur utilisant une pile:

```
DFS_iter( v )
 creer_pile( P );
 empiler( P , v );
 Tantque Non Pilevide( P )
        Depiler( P, v );
        Si Marq[v]= "non visité" :
                Marq[v] ← "visité";
                Pour chaque w adjacent à v  // les prendre en ordre inverse
                        Si Marq[w] = "non visité" :
                                Empiler( P, w )
                        Fsi
                Fpour
        Fsi
 FTQ
```

b) Breadth First Search: Parcours en Largeur

C'est la généralisation pour les graphes du parcours niveau par niveau d'une structure arborescente.
Le principe est de visiter en premier les sommets les plus proches avant de s'éloigner du sommet initial.

Ce type de parcours est nécessaire quand on recherche un ou plusieurs nœuds dans des graphes infinis.

L'algorithme 'BFS(v)' est non récursif et utilise une file d'attente.

```
BFS(v)
  Marq[v] ← "visité"
  CreerFile(F)
  Enfiler(F, v)
  Tantque Non Filevide(F) :
        Défiler(F, v)
        Pour chaque w adjacent à v
              Si Marq[w] = "non visité"
                    Marq[w] ← "visité"
                    Enfiler(F, w)
              Fsi
        Fpour
  Ftantque
```

Avec l'initialisation :

```
   Pour chaque sommet v
        Marq[v] ←  "non visité"
   Fpour
```

Et comme appelant :

```
   Pour v =1, n
        Si Mark(v)= "non visité"
              BFS(v)
        Fsi
   Fpour
```

Dans le cas où le degré moyen par nœud est constant (indépendant de n), ces deux types de parcours (DFS et BFS), ont une complexité linéaire (O(n) si le graphe est formé de n nœuds). Car chaque nœud est visité une seule fois. De plus pour chaque nœud visité, la boucle de parcours des voisins fait un nombre constant d'itérations (le degré moyen est constant). Donc le nombre d'itérations total est proportionnel à n.

Dans le cas où le degré moyen par nœud dépend de n, comme dans le cas d'un graphe complet par exemple où chaque nœud possède n-1 voisins, la complexité est quadratique $O(n^2)$ car pour chaque nœud visité, la boucle des voisins fait n-1 itérations.

Malgré que la complexité des deux parcours soit polynomiale (O(n) ou $O(n^2)$) en fonction de la taille du graphe, beaucoup de problèmes (comme celui du voyageur de commerce PVC) utilisant l'une de ces deux procédure (DFS ou BFS) ont quand même, une complexité exponentielle car l'espace de recherche engendré pour ces problèmes, contient un nombre de nœuds dans l'ordre de $O(a^n)$ ou plus. Par exemple, pour le PVC où il est question de trouver le plus petit cycle Hamiltonien dans un graphe complet de n nœuds, la procédure de parcours (DFS ou BFS) explorera un espace de recherche arborescent renfermant (n-1) ! nœuds feuilles.