Chapitre 3

Complexité des algorithmes

3.1 Introduction

Le temps d'exécution d'un algorithme dépend des facteurs suivants :

- Les données utilisées par le programme,
- La qualité du compilateur (langage utilisé),
- La machine utilisée (vitesse, mémoire,...),
- La complexité de l'algorithme lui-même,

On cherche à mesurer la complexité d'un algorithme indépendamment de la machine et du langage utilisés, c-à-d uniquement en fonction de la taille des données n que l'algorithme doit traiter. Par exemple, dans le cas de tri d'un tableau, n est le nombre d'éléments du tableau, et dans le cas de calcul d'un terme d'une suite n est l'indice du terme, ...etc.

3.2 O-notation

Soit la fonction $T(n)$ qui représente l'évolution du temps d'exécution d'un programme P en fonction du nombre de données n.
Par exemple :

$$T(n) = cn^2$$

Où c est une constante non spécifiée qui représente la vitesse de la machine, les performances du langage, ...etc. On dit dans l'exemple précédent que la complexité de P est $O(n^2)$. Cela veut dire qu'il existe une constante c positive tel que pour n suffisamment grand on a :

$$T(n) \leq cn^2$$

Cette notation donne une majoration du nombre d'opérations exécutées (temps d'exécution) par le programme P. Un programme dont la complexité est $O(f(n))$ est un programme qui a $f(n)$ comme fonction de croissance du temps d'exécution.

3.3 Règles de calcul de la complexité d'un algorithme

3.3.1 La complexité d'une instruction élémentaire

Une opération élémentaire est une opération dont le temps d'exécution est indépendant de la taille n des données tel que l'affectation, la lecture, l'écriture, la comparaison ...etc.

La complexité d'une instruction élémentaire est $O(1)$

3.3.2 La multiplication par une constante

$$O(c * f(n)) = O(f(n))$$

Exemple :

$$O(\frac{n^3}{4}) = O(n^3)$$

3.3.3 La complexité d'une séquence de deux modules

La complexité d'une séquence de deux modules M_1 de complexité $O(f(n))$ et M_2 de complexité $O(g(n))$ est égale à la plus grande des complexité des deux modules : $O(max(f(n), g(n)))$.

$$O(f(n)) + O(g(n)) = O(max(f(n), (g(n))$$

3.3.4 La complexité d'une conditionnelle

La complexité d'une conditionnelle

Si (Cond) **Alors**
  M_1 ;
Sinon
  M_2 ;
Fin Si;

est le max entre les complexités de l'évaluation de <Cond>, M_1 et M_2.

Si (<Cond> $[O(h(n))]$) **Alors**
  M_1 $[O(f(n))]$
Sinon
  M_2 $[O(g(n))]$
Fin Si;

La complexité de la conditionnelle est : $Max\{O(h(n)), O(f(n)), O(g(n))\}$

3.3.5 La complexité d'une boucle

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

Tant que (<Condition> $[O(h(n))]$) **faire**
  [m fois]
  P ; $[O(f(n))]$
Fin TQ;

La complexité de la boucle est : $\sum^{m} Max(h(n), f(n))$ où m est le nombre d'itérations exécutées par la boucle.

Exemple 1 :

Algorithme Recherche;
Var T : **tableau**[1..n] **de entier ;**
x,i : **entier ;**
trouv : **booleen ;**
Début
Pour i **de** 1 **à** n **faire**
Lire (T[i]) ; $O(1)$ $O(\sum_1^n 1) = O(n)$
Fin Pour;

Lire(x) ; O(1)
Trouv ← faux ; $O(1)$ $O(1)$
i ← 1 ; $O(1)$ $O(n)$
Tant que (trouv=faux et i<=n [$O(1)$]) **faire**
Si (T[i]=x [$O(1)$]) **Alors**
Trouv ← vrai ; [$O(1)$] O(1) O(n)
Fin Si;
i ← i + 1 ; [$O(1)$]
Fin TQ;

Si (trouv=vrai [$O(1)$]) **Alors**
Ecrire (x,'existe') [$O(1)$] O(1)
Sinon
Ecrire (x, 'n"existe pas')[$O(1)$]
Fin Si;
Fin.

La complexité de l'algorithme est de $O(n) + O(1) + O(n) + O(1) = O(n)$.

Exemple 2 : Soit le module suivant :

Pour i **de** 1 **à** n **faire**
  [n fois $O(\sum_{i=1}^{n} n-i))]$
  Pour j **de** i+1 **à** n **faire**
    [n − i fois $O(\sum_{1}^{n-i} 1) = O(n-i)]$
    Si (T[i] > T[j] $[O(1)]$) **Alors**
      tmp ← T[i] ; $[O(1)]$
      T[i] ← T[j] ; $[O(1)]$ O(1)
      T[j] ← tmp ;$[O(1)]$
    Fin Si;
  Fin Pour;
Fin Pour;

La complexité $= O(\sum^{n} (n-i) = O((n-1)+(n-2)\ldots+1)$
$= O(\frac{n(n-1)}{2}) = O(\frac{n^2}{2} - \frac{n}{2}) = O(n^2)$

Exemple 3 : Recherche dichotomique

```
Algorithme RechercheDecho;
Var  T : tableau[1..n] de entier ;
     x,sup,inf,m : entier ;
     trouv : booleen ;
Début
 Lire(x) ;            [O(1)]
 Trouv ← faux ;  [O(1)]
 inf ← 1 ;            [O(1)                          O(1)]
 sup ← n ;             [O(1)]
 Tant que (trouv=faux et inf<sup) faire
                                                    [log2(n) fois]
     m ← (inf + Sup ) div 2 ;  [O(1)]
     Si (T[m]=x [O(1)]) Alors
         Trouv ← vrai [O(1)]
     Sinon
         Si (T[m]<x [O(1)]) Alors
             inf ← m +1 ; [ O(1)                  O(1)                  O(log2(n))]
         Sinon
             Sup ← m - 1 ; [O(1)]
         Fin Si;
     Fin Si;
 Fin TQ;
 Si (trouv=vrai [O(1)]) Alors
     Ecrire (x,'existe') ; [O(1)]                  O(1)
 Sinon
     Ecrire (x, 'n'existe pas') ; [O(1)]
 Fin Si;
Fin.
```

La complexité de l'algorithme est de :

$$O(1) + O(log_2(n)) + O(1) = O(log_2(n)) = O(log(n)).$$

3.4 Complexité des algorithmes récursifs

La complexité d'un algorithme récursif se fait par la résolution d'une équation de récurrence en éliminant la récurrence par substitution de proche en proche.

Exemple

Soit la fonction récursive suivante :

```
Fonction Fact( n : entier) : entier;
Début
    Si (n <= 1 [O(1)] ) Alors
        Fact ← 1                          [O(1)]
    Sinon
        Fact ← n * Fact(n - 1)            [O(1)]
    Fin Si;
Fin;
```

Posons $T(n)$ le temps d'exécution nécessaire pour un appel à $Fact(n)$, $T(n)$ est le maximum entre :

- la complexité du test $n \leq 1$ qui est en $O(1)$,
- la complexité de : Fact ← 1 qui est aussi en O(1),
- la complexité de : 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 que :

$$T(n) = \begin{cases} a & si \quad n <= 1 \\ b + T(n-1) & sinon \quad (si\ n > 1) \end{cases}$$

- La constante a englobe le temps du test ($n <= 1$) et le temps de l'affectation (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'affectation finale (Fact ← ...)

$T(n-1)$ (le temps nécessaire pour le calcul de $Fact(n-1)$ sera calculé (récursivement) avec la même décomposition. Pour calculer la solution générale de cette équation, on peut procéder par substitution :

$$\begin{aligned} T(n) &= b + T(n-1) \\ &= b + [b + T(n-2)] \\ &= 2b + T(n-2) \\ &= 2b + [b + T(n-3)] \\ &= \dots \\ &= ib + T(n-i) \\ &= ib + [b + T(n-i+1)] \\ &= \dots \\ &= (n-1)b + T(n-n+1) = nb - b + T(1) = nb - b + a \\ T(n) &= nb - b + a \end{aligned}$$

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

3.5 Types de complexité algorithmique

1. $T(n) = O(1)$, temps constant : temps d'exécution indépendant de la taille des données à traiter.
2. $T(n) = O(log(n))$, temps logarithmique : on rencontre généralement une telle complexité lorsque l'algorithme casse un gros problème en plusieurs petits, de sorte que la résolution d'un seul de ces problèmes conduit à la solution du problème initial.
3. $T(n) = O(n)$, temps linéaire : cette complexité est généralement obtenue lorsqu'un travail en temps constant est effectué sur chaque donnée en entrée.
4. $T(n) = O(n.log(n))$: l'algorithme scinde le problème en plusieurs sous-problèmes plus petits qui sont résolus de manière indépendante. La résolution de l'ensemble de ces problèmes plus petits apporte la solution du problème initial.
5. $T(n) = O(n^2)$, temps quadratique : apparaît notamment lorsque l'algorithme envisage toutes les paires de données parmi les n entrées (ex. deux boucles imbriquées) Remarque : $O(n^3)$ temps cubique
6. $T(n) = O(2^n)$, temps exponentiel : souvent le résultat de recherche brutale d'une solution.