Chapitre 3 : Introduction à l'Analyse des Algorithmes

1. Introduction

Le but en analyse des algorithmes est d'étudier le fonctionnement d'un algorithme. Il s'agit, pour les problèmes solubles, de démontrer que l'algorithme est correct (il résout correctement le problème et il termine toujours) et aussi d'estimer les ressources nécessaires au déroulement de la solution considérée (calculer sa complexité).

Dans ce chapitre, nous introduisons les bases du calcul de la complexité d'un algorithme. La *complexité* d'un algorithme consiste à évaluer la quantité de ses ressources. Les principales ressources qui nous intéressent sont les fonctions temps d'exécution et espace mémoire nécessaire. Pour formaliser ces notions on doit distinguer la notion de problème de celle d'instance.

1.1 Notion de problème

Un problème est une question qui a les propriétés suivantes:

- Elle est générique (s'applique à un ensemble d'éléments);
- Toute question posée pour chaque élément admet une réponse;
- Un problème peut avoir plusieurs solutions (plusieurs algorithmes différents peuvent résoudre le même problème).

Exemple 1 :

1) Déterminer si un entier donné est pair ou impair;

2) Déterminer le maximum d'un ensemble d'entiers donnés

3) Trier une suite d'entiers donnés, etc.

1.2 Notion d'instance de problème

L'instance d'un problème c'est la question générique posée ou appliquée à un élément.

Exemple 2 :

1) L'entier 15 est-il pair ou impair ?

2) Déterminer le maximum de l'ensemble {1, 8, -14, 0, 5}

3) Trier la suite précédente.

2. La Complexité d'un Algorithme

Le calcul de la complexité d'un algorithme a pour but de déterminer la quantité de ressources nécessaires à l'exécution de cet algorithme en fonction de la taille des données.

Ces ressources peuvent être la quantité de mémoire utilisée, la largeur d'une bande passante, le temps de calcul etc. Nous nous intéressons plus souvent au temps et à la mémoire. Le but est de pouvoir identifier, face à plusieurs algorithmes qui résolvent un même problème, celui qui est le plus efficace (le plus rapide et/ou qui consomme le moins d'espace mémoire).

Il s'agit donc de trouver une équation qui relie le temps d'exécution à la taille des données. Nous pourrons ainsi comparer deux algorithmes qui résolvent le même problème en comparant le rapport de leur équation.

Exemple 3 :

1) Calculer le nombre de comparaisons d'éléments dans un algorithme de tri d'une liste d'entiers donnée.
2) Calculer le nombre d'opérations exécutées dans un produit de deux matrices.
3) Calculer le nombre d'appels récursifs dans le cadre de la récursivité.

Plusieurs algorithmes peuvent résoudre un même problème. L'analyse des algorithmes permet de choisir la meilleure solution qui résout un problème donné. La complexité algorithmique est un concept fondamental pour tout informaticien, elle permet de déterminer si un algorithme *a* et meilleur qu'un algorithme *b* et s'il est optimal ou s'il ne doit pas être utilisé. . .

Exemple 4 : Calculer la somme des n premiers nombres entiers.

Solution 1 :

```
Fonction Somme (E/ n: entier) : entier
Variable som, i : entier ;
Début
   som←0 ;
   pour  i←1 à n
    faire
        som←som+i ;
    fait
   retourner som ;
Fin ;
```

Solution 2 :

```
Fonction Somme (E/ n: entier) : entier
Début
    retourner n*(n+1) / 2 ;
Fin ;
```

Pour le même problème, il existe deux solutions justes. En déroulant l'algorithme pour n=1 milliard, la première solution exécute 1 milliard d'instructions pour trouver le résultat tandis que la deuxième solution renvoie le résultat en une seule instruction.

Intuitivement, la complexité d'un algorithme est le nombre d'opérations nécessaires à l'algorithme pour effectuer son travail. Cette complexité donne une idée du temps nécessaire à l'algorithme pour s'exécuter. Elle est mesurée en fonction de la taille des données du problème (pour lequel l'algorithme a été conçu).

L'évaluation de l'efficacité d'un algorithme en termes d'espace occupé et la comparaison entre algorithmes en termes de temps d'exécution, se fait toujours pour des grandes tailles de données.

Mesurer la complexité d'un algorithme, avant son exécution permet "de donner une idée, à l'avance" sur le temps que cet algorithme va mettre pour terminer et retourner le résultat.

2.1 Comment déterminer le temps d'exécution d'un algorithme ?

Il n'est pas nécessaire, à priori, de connaître la valeur *exacte* du temps d'exécution d'un algorithme mais il suffit d'une *approximation* de cette valeur. On pourra mesurer le temps exact une fois le programme lancé sur machine.

La complexité temporelle (ou temps d'exécution d'un algorithme) est une approximation du nombre d'opérations que cet algorithme exécute en fonction de la taille des données.

La valeur approchée de ce temps s'obtient à l'aide d'une fonction mathématique que l'on doit déterminer et qui borne la croissance du temps en fonction des données.

Exemple 5 : *Calcul du maximum d'une suite.*

```
Fonction Maximum (E/ T : tableau [20] entier, E/ n : entier) : entier
Variable max, n : entier
```

```
Debut
    max←T[1] ;
    pour  i←2 à n
      faire
        si (T[i] >max) alors max←T[i] ;
      fait
    retourner max;
Fin ;
```

L'analyse :

1. Si la liste est triée en ordre décroissant on ne fera qu'une seule affectation, celle qui est avant la boucle.
2. Si la liste est triée en ordre décroissant, on fera *n* affectations, une avant la boucle et *n-1* dans la boucle.
3. Si *n* = 10, il y a 10! façons d'arranger ces nombres. Si le max est en première position, on fera une affectation, s'il est en 2ème position, on en fera deux, s'il est en 3ème position, on en fera au plus 3 et ainsi de suite. S'il est en dernière position on fera au plus *n* affectations.

Nous venons de voir que la valeur du temps que met un algorithme à s'exécuter dépend aussi de la répartition des données (structure triée en ordre croissant ou décroissant ou bien non triée). L'exemple suivant montre le nombre de décalages dans un tri-selection où on compare les éléments 2 à 2 et on décale à droite les éléments mal placés:

Exemple 6 : *Soit la suite d'entiers {* 8, 2, 4, 9, 3, 6 }

L'élément coloré donne la position du début des décalages à droite à faire dans chaque ligne, à partir de la suite donnée. Le traitement s'arrête (ligne 8) avec une suite triée.

1)	*8*	2	4	9	3	6
2)	2	*8*	4	9	3	6
3)	2	4	8	*9*	3	6
4)	2	4	*8*	3	9	6
5)	2	*4*	3	8	9	6
6)	2	3	4	8	*9*	6
7)	2	3	4	*8*	6	9
8)	2	3	4	6	8	9

Pour *n* = 6 nous avons 8 lignes de traitement. Nous pouvons admettre, intuitivement, que:

1) pour *n* > 6 le nombre de lignes de traitement augmente, et que les séquences plus courtes soient plus faciles à traiter.

2) un tableau déjà trié "est rapidement trié" alors qu'un tableau trié dans le sens inverse sera trié en temps plus long. Cependant la distribution des données est une information difficile à obtenir car elle est estimée expérimentalement. Elle permet de calculer "la complexité en moyenne".

3) La complexité ne s'évalue pas sur les instances mais sur l'algorithme.

L'analyse de performances d'un algorithme est indépendante du type de la machine sur laquelle il s'exécute. Elle ne fournit pas le nombre exact d'opérations mais un ordre de grandeur. En analyse d'algorithme, il n'y a pas de différences entre une solution qui fait N opérations et une autre qui fait $N + 300$ opérations, car N et $N + 300$ tendent vers la même limite quand N tend vers l'infini.

2.2 Types de Complexité

Il existe trois types de complexité :

1) La complexité du meilleur cas

C'est, par exemple pour l'algorithme du tri d'un tableau, le cas où le tableau donné est déjà trié. Cette complexité n'est pas très intéressante car tous les algorithmes réagissent de la même manière pour ce cas, elle ne permet pas de distinguer deux solutions. Cependant elle peut permettre d'éliminer des solutions très coûteuses même pour le meilleur cas.

2) La complexité en moyenne

Elle nécessite la connaissance de la distribution des données, c'est-à-dire les fonctions de probabilités sur les données qui peuvent être obtenues après avoir effectué plusieurs tests expérimentaux.

3) La complexité du pire cas

Elle fournit une borne supérieure de la ressource (par exemple, le temps d'exécution) qui indique que dans toutes les situations l'algorithme ne peut pas dépasser la valeur de cette borne. C'est donc une garantie et c'est ce que nous cherchons toujours à obtenir.

Il s'agit donc de trouver une équation qui relie le temps d'exécution à la taille des données. Nous pourrons ainsi comparer deux algorithmes qui résolvent le même problème en comparant le rapport de leur équation et choisir le plus rapide.

2.3 La Complexité Asymptotique

En pratique, il est difficile de calculer de manière exacte la complexité d'un algorithme (le nombre d'opérations pour la complexité temporelle).

La complexité *asymptotique* est une approximation du nombre d'opérations que l'algorithme exécute en fonction de la donnée d'entrée. Elle est "asymptotique" parce qu'elle prend en compte une donnée de grande taille et ne retient que le terme de poids fort dans la formule et ignore le coefficient multiplicateur.

Exemple 7 : Soit la donnée n de grande taille et T(n) la complexité exprimée par la formule suivante :

$$T(n) = 10n^3 + 3n^2 + 5n + 1$$

Le terme de poids fort est $10n^3$. En ignorant le coefficient, nous dirons que *T (n)* est bornée par une fonction polynomiale cubique et on note cela par :

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

3. Comparaison de Fonctions Asymptotiques

De la même manière que pour les nombres, les fonctions peuvent aussi être comparées entre elles. On parle de comparaison asymptotique de fonctions.

3.1 Les notations de Landau

Les notations de Landau sont des formules mathématiques qui décrivent les comparaisons de fonctions asymptotiques.

1. La notation O

Elle donne une borne supérieure de la complexité.

f(n) = O(g(n)) ssi il existe des constantes positives c et n_0 telles que

$$f(n) \leq c * g(n) \quad \forall\ n \geq n_0$$

Par exemple: $f(n) = 3*n + 2 = O(n)$ car $3*n + 2 \leq 4*n \quad \forall\ n \geq 2$

2. La notation Ω

Elle donne une borne inf de la complexité.

f(n) = Ω(g(n)) ssi il existe des constantes positives c et n_0 telles que

$$f(n) \geq c * g(n) \quad \forall\ n \geq n_0$$

Par exemple: $f(n) = 3*n + 2 = \Omega(n)$ car $3*n + 2 \geq 3*n \quad \forall\ n \geq 2$

3. La notation Θ

Elle encadre la complexité entre deux bornes.

f (n) =Θ(g(n)) ssi il existe des constantes positives c_1, c_2 et n_0 telles que

$$c_1 * g(n) \leq f(n) \leq c_2 * g(n) \quad \forall\ n \geq n_0$$

Par exemple: f (n) = 3*n + 2 =Θ(n) car on a à la fois : f (n) = O(n) et f (n) = Ω(n)

Vocabulaire

Il existe un vocabulaire courant des fonctions de complexité. Supposons que T(n) est une complexité temps, alors nous dirons :

1) si T(n) = O(1)

On a une complexité constante, indépendante de la taille des données. C'est le cas optimal d'une complexité donc les algorithmes les plus rapides.

2) si T (n) = $O(n^k)$

Cette complexité est dite polynomiale :

si $k = 0$, c'est la complexité constante : $O(n^0) = O(1)$ vue précédemment.

si $k = 1$ elle est dite *linéaire*; c'est le cas de la recherche séquentielle dans une liste;

si $k = 2$ elle est dite *quadratique*. C'est le cas par exemple du tri par permutations;

si $k = 3$, elle est dite *cubique* (par exemple: le produit de deux matrices);

3) si T (n) = O(log n) ou bien O(n log n), … elle est dite *logarithmique*;

4) si T(n) = $O(2^n)$ ou bien O(n!), … elle est dite exponentielle; c'est le cas des algorithmes les plus lents.

Par exemple si T (n) = 2^n, pour n = 60 on a $T(n) = 1; 15\ 10^{12}$ secondes et

$$1{,}15*10^{12} \text{ s} \approx 36558 \text{ ans}$$

4. Exemples

4.1 La recherche séquentielle dans une suite

On peut supposer que la suite est stockée dans un tableau ou bien dans une liste. La recherche doit faire un parcours séquentiel et tester les cases l'une après l'autre, depuis la première case jusqu'à trouver la valeur ou bien arriver à la fin de la suite sans la trouver. Si n est le nombre d'éléments de la suite, nous ferons au plus n tests. Donc on a une complexité linéaire, soit T(n) = O(n)

4.2 La recherche dichotomique dans un tableau trié

L'algorithme calcule le milieu du tableau de taille n et teste si la valeur se trouve en cette position du tableau. Si oui la recherche termine avec une seule comparaison, sinon on recherchera la valeur dans un tableau de taille $n/2$.

Dans le tableau de taille $n/2$ on calcule son milieu et on teste si la valeur se trouve en cette position. Si oui la recherche termine et on aura au total 2 comparaisons, sinon on continue à rechercher la valeur dans un tableau de taille $(n/2)/2$, c'est-à-dire, dans un tableau de taille $n/2^2$.

On procède de la même manière dans ce sous-tableau de taille $n/2^2$: on calcule son milieu et on teste si la valeur se trouve en cette position. Si oui la recherche termine après 3 comparaisons, sinon on continue la recherche dans un tableau de taille $(n/2^2)/2 = n=2^3$ et ainsi de suite.

Pour savoir au bout de combien de fois nous pourrons continuer à diviser la suite, nous supposons (pour simplifier) que la taille initiale du tableau est n= 2^k.

Si à chaque étape on doit diviser le tableau en deux, sa taille se réduit de moitié à chaque étape et nous pourrons faire cela au plus k fois.

Le nombre maximum de tests que l'on fait correspond à k et nous déterminons sa valeur ainsi :

$n = 2^k$ alors $\log n = k*\log 2$, soit $k = \log n / \log 2$ d'où $k= \log_2 n$

Ainsi la recherche dichotomique a une complexité logarithmique.

Puisque la fonction $log\ n \leq n \quad \forall\ n \geq 1$ alors la recherche dichotomique est plus rapide que la recherche séquentielle.

5. Quelques Règles Pratiques

Le nombre d'opérations qu'un algorithme séquentiel effectue peut être es-timé en composant les règles suivantes selon la séquence des instructions (instruction conditionnelles, les itérations, les appels de fonction etc.) de cet algorithme. Nous donnons ci-dessous quelques règles pratiques de calcul de la complexité temporelle.

Soit f (…) une fonction qui résout un problème donné :

5.1 Lorsque *f* est une conditionnelle

Supposons l'algorithme suivant:

```
fontion f (......) : type
  ...
debut
      si (condition) alors   g1 (.....) ;
                     sinon  g2 (.....) ;
fin;
```

$$T\ (f(\ldots)) = Max\ (T1(g1(\ldots)\ ,\ \ T2(g2(\ldots)))$$

5.2 Lorsque *f* est une itération

Supposons l'algorithme suivant:

```
fontion f (….) : type
  ...
debut
    pour  i←1 à n
       faire
          g(.....);
       fait
fin;
```

Si *g* est indépendante de *i* alors $T(f(\ldots)) = n * T1(g(\ldots))$

Si *g* est dépendante de *i* alors $T(f(\ldots)) = \sum_{i=1}^{i=n} (T1(g(i, \ldots))$

a) Cas de deux itérations séquentielles :

Exemple 8 : Ecrire un algorithme qui réalise la somme les éléments d'un tableau.

```
Algorithme BouclesSequentielles
constante max=100;
variable    som , i, n : entier ;
            T : tableau[max] de entier;
Debut
   1. lire (n) ;
   2. som ←0 ;
   3. pour  i←1 à n faire
                        lire (T[i]);
                      fait
   4. pour  i←1 à n faire
                        som←som+ T[i];
                      fait
   5. ecrire (som);
Fin
```

T(n) = O(n) car dans les lignes 1 et 2 et 5 nous avons une seule opération dans chaque ligne, puis les lignes 3 et 4 comportent respectivement n lectures et n additions. Donc au total 2n + 3 opérations ce qui est en O(n) puisque $2n + 3 \leq 3n$ dès que $n > 3$.

b) Cas de deux itérations imbriquées (dépendantes) :

Exemple 9 : Ecrire une fonction qui vérifie si une certaine valeur val est la somme de deux éléments d'un tableau donné.

```
Fonction Verification (E/ T : Tableau, E/ n : entier, E/val : entier) : booleen
variable     i, j : entier ;
debut
    1. pour  i←1 à n-1 faire
                          pour  j← i+1 à n faire
                                                si (T[i]+T[j]=val) alors retourner VRAI;
                                              fait
                        fait
    2. retourner (FAUX);
fin
```

Le nombre d'opérations (comparaisons) que réalise la fonction est estimé comme suit :
pour i = 1 on réalise (n-1) comparaisons (j = 2 à n)
pour i = 2 on réalise (n-2) comparaisons (j = 3 à n)
pour i = 3 on réalise (n-3) comparaisons (j = 4 à n)
….
pour i = k on réalise (n-k) comparaisons (j = k+1 à n)
….
pour i = n-1 on réalise 1 comparaison (j = n à n)

La somme de ces calculs (pour i = 1 à n-1 vaut:

$$(n-1) + (n-2) + (n-3) + \ldots + 2 + 1 = (n * (n-1)) / 2$$

Ainsi, $T(n) = O(n^2)$

Exemple 10 : Soit un tableau d'entiers T et soit une valeur donnée **val**. Ecrire une fonction qui vérifie si val est égale à un certain élément de T si la dimension est impaire et qui vérifie si val est égale à la somme de deux éléments du tableau si sa dimension est paire.

```
Fonction DoubleVerification (E/ T : Tableau, E/ n : entier, E/val : entier) : booleen
variable     i, j : entier ;
debut
    si (n%2=0) alors
                    pour  i←1 à n-1
                        faire
                           pour  j← i+1 à n
                                  faire
                                      si (T[i]+T[j]=val) alors retourner VRAI;
                                   fait
                            fait
                        retourner Faux ;
                sinon
                        pour  i←1 à n
                            faire
                                si (T[i]=val) alors retourner VRAI;
```

```
                    fait
                retourner FAUX;
    fsi ;
fin
```

Si la condition est vraie la vérification nécessite deux boucles imbriquées (dépendantes). La complexité dans ce cas est $O(n^2)$. Si la condition est fausse, le traitement se ramène à la recherche d'une valeur dans un tableau qui a une complexité O(n). Puisqu'il s'agit d'une instruction conditionnelle, la complexité de la fonction étant le maximum entre les deux complexités.

Ainsi, $T(n)= Maximum(O(n),O(n^2))$ donc $T(n)= O(n^2)$