

Analyse et complexité des algorithmes

Table des matières

1 Rappels : Les bases de l'analyse d'un algorithme	3
1.1 Introduction : Problèmes et programmes	3
1.1.1 Notion de problème	3
1.1.2 Notion d'instance de problème	4
1.1.3 La solution d'un problème	4
1.2 La complexité d'un algorithme	4
1.2.1 Comment déterminer le temps d'exécution d'un algorithme ?	6
1.2.2 Types de complexité	8
1.2.3 La complexité asymptotique	8
1.3 Comparaison de fonctions asymptotiques	9
1.3.1 Les notations de Landau	9
1.3.2 Vocabulaire	10
1.4 Exemples	11
1.4.1 La recherche séquentielle dans une suite	11
1.4.2 La recherche dichotomique dans un tableau trié	11
1.5 Quelques règles pratiques	12
1.5.1 Lorsque f est une conditionnelle	12
1.5.2 Lorsque f est une itération	12

Chapitre 1

Rappels : Les bases de l'analyse d'un algorithme

1.1 Introduction : Problèmes et programmes

Le but en analyse d'algorithmes est d'étudier le fonctionnement d'un algorithme. Il s'agit, pour les problèmes solubles, de démontrer que l'algorithme est correct (il termine toujours et il résout correctement le problème) 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 nécessaires.

Pour formaliser ces notions nous devons distinguer la notion de *problème* de celle d'*instance de problème*.

1.1.1 Notion de problème

Un problème est une question qui a les propriétés suivantes :

1. elle est générique (s'applique à un ensemble d'éléments) ;
2. toute question posée pour chaque élément admet une réponse ;

Exemple 1.1.1 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.*

1.1.2 Notion d'instance de problème

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

Exemple 1.1.2 1. *L'entier 15 est-il pair ou impair ?*

2. *déterminer le maximum de l'ensemble $\{1, 5, -14, 0, -5\}$*
3. *trier la suite précédente.*

1.1.3 La solution d'un problème

Cette notion est formellement définie par la notion de *procédure effective*.

Définition 1.1.3 *Une procédure effective est définie en terme d'un langage accepté par une machine de Turing.*

1.2 La complexité d'un algorithme

La complexité d'un algorithme a pour but de déterminer la quantité des 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 1.2.1 1. *Calculer le nombre de comparaisons d'éléments dans un algorithme de tri d'une liste d'entiers donnée en ordre croissant.*

2. *Calculer le nombre d'opérations exécutées dans un produit de deux matrices $n \times n$.*

Plusieurs algorithmes peuvent résoudre un même problème. L'analyse des algorithmes permet de choisir la meilleure solution qui résout un problème donné.

Par exemple, observons les deux solutions (écrites en C) du problème suivant :

Exemple 1.2.2 Calcul du maximum de quatre valeurs a, b, c, d :

Solution 1 :

```
int maximum (int a, int b, int c, int d) {
    int max = a;

    if (b > max) max = b;
    if (c > max) max = c;
    if (d > max) max = d;

    return max;
}
```

Solution 2 :

```
int maximum (int a, int b, int c, int d) {

    if (a > b)
        if (a > c)
            if (a > d) return a;
            else return d;
        else
            if (c > d) return c;
            else return d;
    else
        if (b > c)
            if (b > d) return b;
            else return d;
        else
            if (c > d) return c;
            else return d;
}
```

Les deux solutions exigent exactement 3 comparaisons pour déterminer la réponse bien que la solution 1 soit plus simple à comprendre. Elles sont de même complexité temporelle pour une exécution sur machine. Mais en termes d'espace, la solution 1 exige un espace supplémentaire pour stocker le max. La quantité d'espace supplémentaire étant insignifiante, ces deux solutions sont aussi équivalentes en complexité spatiale. L'évaluation de l'efficacité d'un algorithme en termes d'espace occupé et la comparaison entre algorithmes 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.

1.2.1 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.

Par exemple, considérons le programme C de l'algorithme de recherche du maximum d'une suite d'entiers :

Exemple 1.2.3 *Calcul du maximum d'une suite.*

```
int maximum (tab t, int n) {
    int max = t[0];
    for (i=1; i<n; i++)
        if (t[i] > max) max = t[i];
    return max;
}
```

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 croissant, 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 1.2.4 Soit la suite d'entiers $\{8, 2, 4, 9, 3, 6\}$

L'élément en gras 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.

1.2.2 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 cas pire

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.

1.2.3 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 1.2.5 Soit la donnée n de grande taille et $T(n)$ la complexité exprimée par la formule suivante :

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

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

$$T(n) = \mathcal{O}(n^3)$$

1.3 Comparaison de fonctions asymptotiques

Comme nous comparons les nombres, il est aussi possible de comparer les fonctions asymptotiques.

1.3.1 Les notations de Landau

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

1. La notation \mathcal{O}

Elle donne une borne sup de la complexité.

$$f(n) = \mathcal{O}(g(n)) \text{ ssi } \exists \text{ des constantes positives } c \text{ et } n_0 \text{ telles que}$$

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

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

2. La notation Ω

Elle donne une borne inf de la complexité.

$$f(n) = \Omega(g(n)) \text{ ssi } \exists \text{ des constantes positives } c \text{ et } n_0 \text{ telles que}$$

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

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.

$$f(n) = \Theta(g(n)) \text{ ssi } \exists \text{ des constantes positives } c_1, c_2 \text{ et } n_0 \text{ telles que}$$

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

Par exemple : $f(n) = 3 * n + 2 = \Theta(n)$ car on a à la fois :

$$f(n) = \mathcal{O}(n) \text{ et } f(n) = \Omega(n)$$

1.3.2 Vocabulaire

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

1. si $T(n) = \mathcal{O}(1)$

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 $T(n) = \mathcal{O}(n^k)$

Cette complexité est dite **polynomiale** :

- 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) ;
- si $k = 0$, c'est la complexité constante : $\mathcal{O}(n^0) = \mathcal{O}(1)$ vue précédemment.

3. si $T(n) = \mathcal{O}(\log n)$ ou bien $\mathcal{O}(n * \log n), \dots$: elle est dite *logarithmique* ;

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

Par exemple si $T(n) = 2^n$ microsecondes (μs)¹, pour $n = 60$ on a $T(n) = 1,15 * 10^{12} \mu s$. Sachant que dans une année il y a $31536 * 10^3$ s, cela donne :

$$1,15 * 10^{12} \text{ s} \approx 36.558 \text{ ans soit plus de 360 siècles.}$$

1. $1\mu s = 10^{-6} s$

1.4 Exemples

1.4.1 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 $T(n) = \mathcal{O}(n)$

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

1.5 Quelques règles pratiques

Le nombre d'opérations qu'un algorithme séquentiel effectue peut être estimé 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 $f(\dots)$ une fonction qui résout un problème donné :

1.5.1 Lorsque f est une conditionnelle

Supposons l'algorithme suivant :

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

Alors

$$T(f(\dots)) = \text{Max}(T1(g1(\dots)), T2(g2(\dots)))$$

1.5.2 Lorsque f est une itération

Supposons l'algorithme suivant :

```
fonction f(.....): type
...
debut
  pour i<-1 à n faire
    g(.....);
    fait
fin;
```

et g est indépendante de i , alors

$$T(f(\dots)) = n * T1(g(\dots))$$

Si g est dépendante de i alors

$$T(f(\dots)) = \sum_{i=1}^{i=n} T1(g(i, \dots))$$

Par exemple :

Exemple 1.5.1 *cas de deux itérations séquentielles*

Action Exo1Complexite;

```

constante max=100;
entier S, i;
T: tableau[max]de entier;
//      Traitement1
debut
1: lire (x, taille);
2: S<- 0;

3: pour i<-1 à n faire lire (T(i));
4: pour i<-1 à n faire
    s<- S+i;
    fait
5: ecrire (S),
fin
```

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

Exemple 1.5.2 *cas de deux itérations imbriquées*

Action Exo2Complexite;

```

constante max=100;
entier S, i;
T: tableau[max]de entier;
```

```

//      Traitement2
debut
1: lire (x, taille);
2: S<- 0;

3: pour i<-1 à n-1 faire
   pour j<-i+1 à n faire
      s<- S+1;
   fait
4: ecrire (S),
fin

```

Alors le nombre de fois que l'instruction $S <- S + 1$ s'exécute est ainsi estimé :

- pour $i = 1$ on la calcule $n - 1$ fois ($j = 2$ à n)
- pour $i = 2$ on la calcule $n - 2$ fois ($j = 3$ à n)
- pour $i = 3$ on la calcule $n - 3$ fois ($j = 4$ à n), etc.
- ...
- pour $i = k$ on la calcule $n - k$ fois ($j = k + 1$ à n), etc
- ...
- pour $i = n - 1$ on la calcule 1 fois ($j = n$ à n)

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

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

$$T(n) = \mathcal{O}(n^2)$$

Exemple 1.5.3 cas d'une conditionnelle

Si les traitements des deux exemples précédents se trouvent dans une instruction de type :

```

Si (condition) alors Traitement1
                     sinon Traitement2
fsi

```

et **Traitement1** est en $\mathcal{O}(n)$ et **Traitement2** est en $\mathcal{O}(n^2)$ alors cette conditionnelle est en $\mathcal{O}(\text{Max}(n, n^2))$ donc en $\mathcal{O}(n^2)$.