Mathématiques pour l'informatique

Gwendal Le Bouffant

ENSSAT

Plan du cours

Introduction

Le temps d'exécution d'un programme dépend de plusieurs données :

1. du type d'ordinateur utilisé ;
2. du langage utilisé (représentation des données) ;
3. de la complexité abstraite de l'algorithme sous-jacent.

On distingue deux types de complexité :

1. La complexité en temps : c'est le temps mis par un algorithme pour traiter une donnée.
2. La complexité en espace : c'est l'espace mémoire utilisé par un algorithme pour traiter une donnée en plus de la taille de la donnée.

Dans ce cours nous ne nous intéresserons qu'à la **complexité en temps**.

Complexité en temps d'un algorithme

L'objectif est de :

- Pouvoir comparer entre eux différents algorithmes résolvant le même problème.
- Déterminer si un algorithme donné est exécutable en un temps acceptable sur des données d'une certaine taille.
- Fournir une bonne supérieure de la taille des données pour lesquelles l'algorithme est utilisable.

Il ne faut pas confondre la complexité d'un algorithme et la complexité d'un problème qui est la complexité du meilleur algorithme connu pour le résoudre (cf. LSI 2).

Complexité en temps d'un algorithme

Pour chaque algorithme, il s'agit de mettre en évidence :

- Un ou plusieurs paramètres reflétant la taille des données d'entrée.
- Une ou plusieurs opérations significatives, dites opérations élémentaires.

Exemples

- Algorithme de calcul du produit de deux matrices à coefficients réels, carrées d'ordre n.
- Décomposition d'un entier naturel en facteurs premiers.

Complexité en temps d'un algorithme

On distingue généralement trois cas :

- Le pire des cas : correspond à une donnée ou un ensemble de données pour lesquelles l'algorithme s'exécute en temps maximal.
- Le meilleur des cas : correspond à une donnée ou un ensemble de données pour lesquelles l'algorithme s'exécute en temps minimal.
- En moyenne : correspond à la complexité moyenne sur tous les types de données possibles.

Remarques

- La complexité dans le meilleur et dans le pire des cas donnent des indications sur les temps d'exécution minimaux et maximaux de l'algorithme.
- La complexité en moyenne permet d'estimer le comportement en général de l'algorithme, mais elle nécessite de connaître les probabilités p(d), ce qui peut s'avérer délicat.

Complexité en temps d'un algorithme

Exercices

1. Écrire un algorithme de recherche séquentielle d'un élément a dans une liste et calculer sa complexité.
2. Calculer le nombre d'opérations de l'algorithme de tri le plus simple :

```
Procédure slowsort
        Pour i de 1 à n-1
                Pour j de i+1 à n
                        Si x(j)< x(i) alors permuter(x(j),x(i))
                Fin
        Fin
Fin
```

Calcul pratique des coûts

Deux cas se présentent :

- Si l'algorithme est écrit sous une forme **itérative**, il s'agit de compter les opérations élémentaires dans la boucle la plus interne, puis de tenir compte des boucles les contenant.
- Si l'algorithme est écrit sous une forme **récursive**, il se prête particulièrement bien à un calcul du coût à l'aide d'une **équation récurrente**.

Exercice

Calculer la complexité du calcul de la somme des n premiers entiers :

1. Par un algorithme itératif.
2. Par un algorithme récursif.
3. Par la formule $\sum_{i=0}^{n} i = \frac{n(n-1)}{2}$.

Ordre de grandeur

Afin de simplifer l'étude, on regarde uniquement des complexités asymptotiques (c'est-à-dire quand la taille de l'entrée tend vers l'infini). Pour cela on introduit 3 notations :

- $\mathscr{O}$: borne supérieure asymptotique d'une fonction.
- Ω : borne inférieure asymptotique d'une fonction.
- Θ : borne asymptotiquement approchée d'une fonction : à la fois une borne inférieure et supérieure.

Ces notations vont être utilisées pour décrire le comportement asymptotique des algorithmes.

Ordre de grandeur

Définition 1.1

Une suite $f(n)$ positive est dominée asymptotiquement par une suite $g(n)$ si et seulement si :
$\exists c > 0,\ \exists n_0 \in \mathbb{N}$ *tels que* $\forall n > n_0,\ f(n) \leq cg(n)$
On note cette situation $f = \mathscr{O}(g)$.

Exemples

1. $n = \mathscr{O}(n)$,
2. $3n + 103700 = \mathscr{O}(n)$,
3. $3n^3 + 2n + 1 = \mathscr{O}(n^3)$.

Ordre de grandeur

Définition 1.2

Une suite $f(n)$ positive est minorée asymptotiquement par une suite $g(n)$ si et seulement si :
$\exists c > 0,\ \exists n_0 \in \mathbb{N}$ *tels que* $\forall n > n_0,\ cg(n) \leq f(n)$
On note cette situation $f = \Omega(g)$.

Exemple

$3n^3 + 2n + 1 = \Omega(n)$.

Ordre de grandeur

Définition 1.3

Deux fonctions f et g ont même ordre de grandeur asymptotique si et seulement si :
$\exists c,\, d > 0,\ \exists n_0 \in \mathbb{N}$ *tels que* $\forall n > n_0,\ cg(n) \leq f(n) \leq dg(n)$
On note cette situation $f = \Theta(g)$.

Exemples

1. $n = \Theta(n)$,
2. $3n + 103700 = \Theta(n)$,
3. $3n^3 + 2n + 1 = \Theta(n^3)$.

Exercice

À l'aide des théorèmes de croissance comparée, donner les ordres de grandeur de : $f(n) = 9n^2 + 17n \ln n + 26n + 4 \ln n + 2$.

Lien avec la complexité

- Les notations $\mathscr{O}$ et Θ servent à décrire le comportement asymptotique des algorithmes.
- Si deux algorithmes ont la même borne asymptotique supérieure, alors pour des grandes données ils mettront le même temps (à une constante multiplicative près) au maximum.
- Si deux algorithmes ont la même borne asymptotique inférieure, alors pour des grandes données ils mettront le même temps (à une constante multiplicative près) au minimum.

Lien avec la complexité

- Si on dispose d'un algorithme A dont on connaît la fonction de complexité en temps f dans le pire des cas, on dira qu'il s'exécute en $\mathscr{O}(f)$.
- Si on dispose d'un algorithme A dont on connaît la fonction de complexité en temps f dans le meilleur des cas, on dira qu'il s'exécute en $\Omega(f)$.
- Si on dispose d'un algorithme A dont la fonction de complexité en temps f est la même dans le meilleur des cas et dans le pire des cas, on dira qu'il s'exécute en $\Theta(f)$.
- Si on dispose d'un algorithme A, pour lequel on ne sait pas calculer exactement la complexité, mais seulement un encadrement tel que $3n^2 \leq c(n) \leq 4n^2 + 10$, alors on pourra dire que la complexité est en $\Theta(n^2)$.

Conclusion

Les algorithmes utilisables de façon efficace sont ceux qui s'exécutent en temps :

- Constants : $\Theta(1)$.
- Logarithmique en la taille n de la donnée d'entrée : $\Theta(\ln n)$.
- Linéaire en n : $\Theta(n)$.
- Quasi-linéaire : $\Theta(n \ln n)$.
- Quadratique : $\Theta(n^2)$.

Les algorithmes en temps polynômiale : $\Theta(n^k)$, avec $k > 3$ ou en temps exponentiel : $\Theta(k^n)$ $(k > 0)$ sont à peu près inutilisables pour des grandes valeurs de n.

Attention toutefois aux limites d'un résultat asymptotique : pour une plage de données fixées il se peut qu'un algorithme en $\mathscr{O}(Cn^2)$ soit plus rapide qu'un algorithme en $\mathscr{O}(C_0 n)$ si la constante C est beaucoup plus petite que la constante C_0. C'est par exemple le cas de la multiplication par l'algorithme FFT.

Équations récurrentes

Définition 2.1

Une **équation récurrente** *est une égalité reliant les termes successifs d'une suite.*

Exemples

- $u_{n+1} = \ln(1 + u_n)$ est une équation récurrente **non-linéaire d'ordre 1**.
- $u_{n+2} = u_{n+1} + 2u_n - 3$ est une équation récurrente **linéaire d'ordre 2**, à **coefficients constants**, **non-homogène**.
  Si on enlève le terme -3, on obtient l'équation **homogène associée** : $u_{n+2} = u_{n+1} + 2u_n$.

Les exemples précédents portent sur de suites définies dans $\mathbb{R}$ ou $\mathbb{C}$: ce sont des **équations scalaires**.

Équations récurrentes

Si on considère des suites de $\mathbb{R}^k$ ou $\mathbb{C}^k$, on obtient des **systèmes d'équations récurrentes vectorielles**.

Exemple

$U_{n+1} = AU_n + B$ avec $A = \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix}$, $U_n = \begin{pmatrix} x_n \\ y_n \end{pmatrix}$ et $B = \begin{pmatrix} 2 \\ 1 \end{pmatrix}$

Remarque

Une équation scalaire linéaire d'ordre k peut toujours s'écrire comme un système vectoriel d'ordre 1.

En posant $U_n = \begin{pmatrix} u_n \\ u_{n+1} \end{pmatrix}$ réécrire l'équation $u_{n+2} = u_{n+1} + 2u_n - 3$ dans ce sens.

Résolution des Équations récurrentes linéaires

Définition 2.2

Résoudre une équation récurrente (ou un système d'équations récurrentes) c'est déterminer toutes les suites vérifiant cette équation.
La donnée des k premiers termes, pour une équation d'ordre k, mène à une solution unique.

Théorème 2.3

Du fait de la linéarité, on obtient toutes les solutions en ajoutant à l'une d'elles toutes les solutions de l'équation homogène.

Résolution de l'équation homogène

- Il est clair que l'**ensemble des solutions** de l'équation homogène est stable par combinaison linéaire et qu'il est non vide. Donc il forme un **espace vectoriel**.
- On peut donc montrer que **la dimension de cet e.v** est égale à l'ordre de l'équation.
  On est donc amené à chercher k solutions indépendantes.
- **Si l'équation est à coefficients constants** il est facile de déterminer s'il existe des solutions du type (r^n).

Exemple

$u_{n+2} = u_{n+1} + 2u_n$.

Résolution de l'équation homogène

Théorème 2.4

- *Plus généralement, si l'équation caractéristique d'un problème d'ordre k possède k solutions distinctes $r_1, r_2, \ldots r_k$ alors les (r_i^n) sont des solutions indépendantes formant une base de l'espace vectoriel des solutions de l'équation homogène.*
- *Dans le cas des racines multiples à l'équation caractéristique, par exemple si r est une solution d'ordre m, on peut montrer que (r^n), (nr^n), $\ldots$, $(n^{m-1}r^n)$ forme une famille libre de solutions qui, associée aux autres, donne une base.*

Recherche d'une solution particulière

Pas de méthode générale. Il faut deviner, en s'inspirant de l'allure du second membre.

Exemples

1. $u_{n+2} = u_{n+1} + 2u_n - 3$
2. $u_{n+2} = u_{n+1} + 2u_n - 3.2^n$

Plus généralement si le second membre est du type $P(n)r^n$ où $P(n)$ est un polynôme et r est une valeur quelconque, il faut chercher une solution du type $Q(n)r^n$ où $Q(n)$ est un polynôme de degré égal à celui de P si r n'est pas solution de l'équation caractéristique, et augmenté de l'ordre de multiplicité de r s'il est solution de l'équation caractéristique.

Exercice

Résoudre $U_{n+1} = AU_n + B$ avec $A = \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix}$, $U_n = \begin{pmatrix} x_n \\ y_n \end{pmatrix}$ et $B = \begin{pmatrix} 2 \\ 1 \end{pmatrix}$

Fonctions génératrices

Définition 2.5

La **fonction génératrice** *de la suite* (u_n) *réelle ou complexe est la fonction*

$$G(z) = \sum_{n=0}^{+\infty} u_n z^n.$$

Elle est définie sur un disque $\{|z| < R\}$, R *étant le* **rayon de convergence** *de la série entière.*

Outre les équations linéaires cela permet de résoudre les équations comportant des **produits de convolution**.

Fonctions génératrices

Remarque

En traitement du signal, la coutume est plutôt l'usage de la **transformée en z** définie sur $\left\{|z| > \frac{1}{R}\right\}$:

$$X(z) = \sum_{n=0}^{+\infty} u_n z^{-n}.$$

Exemple

Soit $u_n = a^n,\ a \neq 0$.
Donner sa fonction génératrice puis sa transformée en z.

Proposition 2.6

- *Ces transformations sont* **linéaires** *:*
  Si $(u_n) \mapsto U$ *et* $(v_n) \mapsto V$
  alors $(\alpha u_n + \beta v_n) \mapsto \alpha U + \beta V$.
- *L'effet d'une* **translation** *est facile à exprimer :*
  Si $(u_n) \mapsto U(z) = \sum u_n z^n$ *alors* $(u_{n+1}) \mapsto \sum u_{n+1} z^n = \dots$
- *Le* **produit par n** *aussi :*
  Si $U = \sum u_n z^n$ *alors* $U' = \sum n u_n z^{n-1}$
  donc $zU' = \sum n u_n z^n$.
- *Le* **produit de convolution** *de deux suites est transformé en produit (simple) de leurs transformées :*

$$\text{Si } (u_n) \mapsto U(z) \text{ et } (v_n) \mapsto V(z) \text{ alors } (u_n * v_n) \mapsto U(z)V(z).$$

Fonctions génératrices

Remarque

Les formules sont analogues pour la transformation en z, en remplaçant z par $\frac{1}{z}$.

Exemple

Résoudre par **fonction génératrice** puis **transformée en** z :

$$u_{n+2} = u_{n+1} + 2u_n + 3$$