

Chapitre 3 : Analyse de Programmes

20 septembre 2023

Introduction

Pour un algorithme donné, plusieurs questions peuvent se poser :

- Est-on sûr que l'algorithme **termine** ?
- Est-on sûr que l'algorithme est **correct**, c'est-à-dire, que la sortie calculée par la séquence d'instruction est bien celle attendue pour l'entrée donnée selon la spécification du problème ?
- Combien de **temps** et combien de ressources matérielles vont être nécessaires pour l'exécution de l'algorithme sur une entrée donnée ?

Introduction

Problématique : Nous allons voir dans ce chapitre des méthodes qui permettent, par des raisonnements, d'observer les propriétés des algorithmes, concernant leur **terminaison**, leur **correction** et leur **complexité**.

Terminaison

Définition

On dit qu'un algorithme **termine** lorsque pour toute entrée de l'algorithme, le nombre d'étapes de calcul en fonction de l'entrée est fini.

Terminaison

Exemple

Le programme ci-dessous ne termine pas, car il utilise une boucle infinie :

```
#include <stdio.h>
```

```
int main(){
```

```
    int x = 1;
```

```
    while (x>0){
```

```
        printf("%d \n", x);
```

```
        x++;
```

```
    }
```

```
}
```

Terminaison

Exemple

Le programme ci-dessous termine, car la condition de la boucle cesse d'être vérifiée au bout de 20 itérations :

```
#include <stdio.h>
```

```
int main(){
```

```
    int x = 1;
```

```
    while (x<=20){
```

```
        printf("%d \n", x);
```

```
        x++;
```

```
    }
```

```
}
```

Terminaison

Remarque

On peut ne pas savoir si un programme termine (exemple : conjecture de Syracuse)

Terminaison

Nombre d'opération non-borné : possible avec des **boucles**

Prouver qu'un algorithme contenant des boucles termine =
prouver que ces boucles terminent : technique du **variant**

Terminaison

Définition

Soit un algorithme contenant une boucle. Un **variant** est une valeur s'exprimant en fonction des variables de l'algorithme qui :

- ne prend que des valeurs entières positives ou nulles lors d'une exécution de la boucle ;
- décroît strictement à chaque tour de boucle.

Terminaison

Théorème

Si une boucle admet un variant, alors cette boucle termine.

Application : dans le second programme présenté, on considère la quantité $y = 20 - x$:

- La condition $x \leq 20$ est équivalente au fait que le variant $y = 20 - x$ soit positif ou nul, et y est entier ;
- x est incrémenté à chaque itération de la boucle : x croît, donc y décroît.

Terminaison


Exemple

Algorithme d'exponentiation naïve :

```
int exp_naive(int a, int p)
{
    int r = 1;
    for (int i = 0; i < p; i++){
        r = r*a;
    }
    return r;
}
```

Terminaison

Attention



Il faut bien que le variant soit entier pour assurer la terminaison : c'est parce qu'il est entier et décroissant que sa valeur baisse d'au moins une unité à chaque itération de la boucle, et qu'il y a au plus n itérations si la valeur initiale du variant est n .

Si le variant n'est pas entier, il peut décroître une infinité de fois sans devenir inférieur à 0 (exemple : suite des $\frac{1}{n}$).

Terminaison

Problème : peut-on détecter de façon systématique pour un algorithme s'il termine ou non (comme on peut détecter s'il est syntaxiquement correct) ?

Définition

Le **Problème de l'Arrêt** est un problème de décision dont la spécification est la suivante :

- **Entrée** : Code source d'un programme dans un langage donné
- **Sortie** : *Vrai* si le programme termine, *Faux* sinon.

Terminaison

Théorème

Le problème de l'Arrêt est **indécidable** : il n'existe pas d'algorithme capable de déterminer, pour un code source de programme donné, si le programme termine ou non.

Les analyses de terminaison de programme doivent se faire au cas par cas, et lorsque cela est possible

Correction

- Terminaison : est-ce qu'un algorithme effectue un nombre fini d'opérations sur une entrée donnée ?
- **Correction** : est-ce qu'un algorithme renvoie la sortie attendue pour sa spécification sur une entrée donnée ?

Correction

Définition

On dit qu'un algorithme est **partiellement correct** s'il correspond bien à sa spécification pour les entrées sur lesquelles il termine.

On dit qu'un algorithme est **totalement correct** s'il est partiellement correct et s'il termine pour toute entrée.

Correction

Exemple

Un algorithme qui prend en entrée un entier naturel non-nul et calcule l'ensemble des successeurs dans la suite de Syracuse jusqu'à la première occurrence de 1 sera partiellement correct (terminaison pas assurée)

Correction

Définition

Soit un algorithme contenant une boucle. Un **invariant de boucle** est une propriété qui :

- est vraie avant le début de la boucle ;
- si elle est vraie au début d'un tour de boucle, est vraie également à la fin.

Correction

Théorème

Soit un algorithme comprenant une boucle, et un invariant de cette boucle. Alors pour toute entrée pour laquelle l'algorithme termine, l'invariant est vérifié à la fin de l'exécution de la boucle.

Exemple

Dans l'algorithme d'exponentiation naïve, on a $r = a^i$ à chaque étape de calcul ; c'est un variant de la boucle `for`.

Complexité

Pour un même problème, plusieurs algorithmes différents sont possibles, avec des comportements différents

On peut donc chercher à mesurer la **performance** de ces algorithmes, en mesurant :

- Leur **complexité temporelle**
- Leur **complexité spatiale**

Complexité

La complexité temporelle (ou *coût*) est définie de la façon suivante :

Définition

Pour un programme et une entrée donnée, la **complexité temporelle** est le nombre d'opérations élémentaires réalisées pendant l'exécution du programme sur l'entrée.

Complexité

Les opérations élémentaires (ou opérations atomiques) sont les opérations correspondant à une unique instruction assembleur. Les opérations considérées comme élémentaires sont les suivantes :

- Accès mémoire pour la lecture ou l'écriture d'une valeur de variable (ou de case de tableau)
- Opération d'addition, soustraction, multiplication, et opérations de comparaisons sur des valeurs numériques.

Complexité

Nombre d'opération en fonction d'une entrée e : $Op(e)$

Définition

Soit un programme, soit Op la fonction associant à chaque entrée e du programme la complexité temporelle de celui-ci sur cette entrée. Soit $n \in \mathbb{N}$ un entier naturel et soit E_n l'ensemble des entrées de taille n .

La **complexité dans le pire cas** en fonction de n est la complexité temporelle la plus élevée pour une entrée de taille n :

$$C(n) = \max_{e \in E_n} Op(e)$$

Complexité

Nombre d'opération en fonction d'une entrée e : $Op(e)$

Définition

Soit un programme, soit Op la fonction associant à chaque entrée e du programme la complexité temporelle de celui-ci sur cette entrée. Soit $n \in \mathbb{N}$ un entier naturel et soit E_n l'ensemble des entrées de taille n .

La **complexité en moyenne** en fonction de n est la moyenne des complexités temporelle de toutes les entrées de taille n .

Complexité

Nombre d'opération en fonction d'une entrée e : $Op(e)$

Définition

Soit un programme, soit O_p la fonction associant à chaque entrée e du programme la complexité temporelle de celui-ci sur cette entrée. Soit $n \in \mathbb{N}$ un entier naturel et soit E_n l'ensemble des entrées de taille n .

La **complexité en moyenne** en fonction de n est la moyenne des complexités temporelle de toutes les entrées de taille n .

Complexité

Définition de la taille de l'entrée = variable :

- Longueur pour un tableau, une liste, une chaîne de caractères...
- Pour un entier : possible d'utiliser la valeur de l'entrée, ou la taille de son écriture, selon les cas

Complexité

Remarque

On peut également (en remplaçant max par min) définir une complexité dans le meilleur cas.

Remarque

Pour la complexité en moyenne, elle peut être définie en fonction de la probabilité p_e de tomber sur chacune des entrées e :

$$C_m(n) = \sum_{e \in E_n} p_e \times Op(e).$$

On considérera le plus souvent une répartition uniforme

$$\text{discrète, d'où } C_m(n) = \frac{1}{\text{Card}(E_n)} \times \sum_{e \in E_n} Op(e).$$

Complexité

Remarque

La complexité en moyenne ne peut pas toujours s'exprimer, notamment lorsqu'il existe une infinité de cas possibles.

Exemple

On considère un algorithme, qui prend deux listes de longueurs N_1 et n_2 en entrées et vérifie l'existence d'un élément en commun entre les deux listes (parcours d'une boucle dans une boucle) :

- Pire cas : $N_1 \times N_2$
- Cas moyen : dépend de l'ensemble des valeurs possibles

Ordres de Grandeur

Plutôt qu'à des complexités exactes, on s'intéressera à des **ordres de grandeur**

On souhaite savoir comment se comporte la complexité temporelle de l'algorithme lorsque ces entrées sont de grande taille, c'est-à-dire lorsque leur taille tend vers $+\infty$. On parle de **complexité asymptotique**

Ordres de Grandeur

Plutôt qu'à des complexités exactes, on s'intéressera à des **ordres de grandeur**

On souhaite savoir comment se comporte la complexité temporelle de l'algorithme lorsque ces entrées sont de grande taille, c'est-à-dire lorsque leur taille tend vers $+\infty$. On parle de **complexité asymptotique**

Pour définir la complexité asymptotique, on aura besoin de définir rigoureusement les ordres de grandeur. Pour cela, on utilisera les **notations de Landau** :

Ordres de Grandeur

Définition

Soient f et g deux fonctions de \mathbb{N} dans \mathbb{N} .

- On dit que g est **majorée par f** à un facteur constant près s'il existe un facteur $k \in \mathbb{R}_+^*$ et un rang $n_0 \in \mathbb{N}$ tels que, pour tout entier $n \geq n_0$, on a $g(n) \leq kf(n)$. L'ensemble des fonctions majorées par f à un facteur constant près se note $\mathcal{O}(f(n))$, et on note :

$$g(n) = \mathcal{O}(f(n)) \text{ ou } g(n) \in \mathcal{O}(f(n))$$

Ordres de Grandeur

Définition

Soient f et g deux fonctions de \mathbb{N} dans \mathbb{N} .

- On dit que g est **minorée par** f à un facteur constant près s'il existe un facteur $k \in \mathbb{R}_+^*$ et un rang $n_0 \in \mathbb{N}$ tels que, pour tout entier $n \geq n_0$, on a $kf(n) \leq g(n)$. L'ensemble des fonctions minorées par f à un facteur constant près se note $\Omega(f(n))$, et on note :

$$g(n) = \Omega(f(n)) \text{ ou } g(n) \in \Omega(f(n))$$

Ordres de Grandeur

Définition

Soient f et g deux fonctions de \mathbb{N} dans \mathbb{N} .

- On dit que g est **de même ordre de grandeur que** f si g est à la fois majorée par f à un facteur constant près et minorée par f à un autre facteur constant près. L'ensemble des fonctions de même ordre de grandeur que f se note $\Theta(f(n))$, et on note :

$$g(n) = \Theta(f(n)) \text{ ou } g(n) \in \Theta(f(n))$$

Ordres de Grandeur

Définition

Soient f et g deux fonctions de \mathbb{N} dans \mathbb{N} .

- On dit que g est **équivalente** à f si

$$\lim_{n \rightarrow +\infty} \frac{g(n)}{f(n)} = 1 \text{ (en supposant que } f \text{ ne s'annule pas). On note :}$$

$$g(n) \sim f(n)$$

Ordres de Grandeur

On s'intéressera **très principalement** à la notation O , pour majorer les complexités asymptotiques d'algorithmes.

Remarque

On a $g(n) \in \mathcal{O}(f(n)) \Leftrightarrow f(n) \in \Omega(g(n))$: g est majorée par f si et seulement si f est minorée par g .

Ordres de Grandeur

Exemple

Soit un algorithme dont la complexité dans le pire cas est $C(n) = 4n^3 + 3n^2 + 2n + 1$. On a :

- ❶ $C(n) = \mathcal{O}(n^3)$ (facteur $k = 4$)
- ❷ $C(n) = \Omega(n^3)$ (facteur $k = 1$)
- ❸ $C(n) = \Theta(n^3)$ (conséquence des deux affirmations précédentes)
- ❹ $C(n) \sim 4n^3$

Propriétés

On a les propriétés suivantes sur les relations $g(n) = \mathcal{O}(f(n))$, $g(n) = \Omega(f(n))$ et $g(n) = \Theta(f(n))$:

Proposition

La relation $g(n) = \mathcal{O}(f(n))$ est :

- **réflexive** : pour toute fonction f de \mathbb{N} dans \mathbb{N} ,
 $f(n) = \mathcal{O}(f(n))$
- **transitive** : pour toutes fonctions f , g et h de \mathbb{N}
dans \mathbb{N} , si $g(n) = \mathcal{O}(f(n))$ et $h(n) = \mathcal{O}(g(n))$
alors $h(n) = \mathcal{O}(f(n))$

Propriétés

Proposition

La relation $g(n) = \Omega(f(n))$ est réflexive et transitive.

Proposition

- La relation $g(n) = \Theta(f(n))$ est réflexive et transitive.
- La relation $g(n) = \Theta(f(n))$ est **symétrique** : pour toutes fonctions f et g de \mathbb{N} dans \mathbb{N} , si $g(n) = \Theta(f(n))$ alors $f(n) = \Theta(g(n))$.

Propriétés

Proposition

- Si $C(n) = \Theta(f(n))$ alors $kC(n) = \Theta(f(n))$ pour $k > 0$.
- Si $C(n) = \Theta(f(n))$ et $D(n) = \Theta(f(n))$ alors $C(n) + D(n) = \Theta(f(n))$.
- Si $C(n) = \Theta(f(n))$ et $D(n) = \Theta(g(n))$ alors $C(n)D(n) = \Theta(f(n)g(n))$.

Propriétés

Proposition

- Si $C(n) = \mathcal{O}(f(n))$ et $D(n) = \mathcal{O}(f(n))$ alors $C(n) + D(n) = \mathcal{O}(f(n))$.
- Si $C(n) = \mathcal{O}(f(n))$ et $D(n) = \mathcal{O}(g(n))$ alors $C(n)D(n) = \mathcal{O}(f(n)g(n))$.
- Si $C(n) = \mathcal{O}(D(n))$ alors $C(n) + D(n) = \Theta(D(n))$.
- Si $C(n) = \Theta(f(n))$ et $D(n) = \mathcal{O}(f(n))$ alors $C(n) + D(n) = \Theta(f(n))$.

Complexités Usuelles

- $\mathcal{O}(1)$: complexité **constante**
- $\mathcal{O}(\log_a(n))$, $a > 1$: complexité **logarithmique**
- $\mathcal{O}(n)$: complexité **linéaire**
- $\mathcal{O}(n \log(n))$: complexité **linéarithmique**
- $\mathcal{O}(n^2)$: complexité **quadratique**
- $\mathcal{O}(n^3)$: complexité **cubique**
- $\mathcal{O}(n^k)$, $k \in \mathbb{N}^*$: complexité **polynomiale**
- $\mathcal{O}(a^n)$, $a > 1$: complexité **exponentielle**

Complexités Usuelles

Remarque

Le choix du a dans la complexité logarithmique ou linéarithmique n'a donc pas d'importance.

Proposition

Soient $k, k' \in \mathbb{R}$ avec $1 < k < k'$, soient $a, a' \in \mathbb{R}$ avec $1 < a < a'$. On a :

$$\begin{aligned} \mathcal{O}(1) &\subset \mathcal{O}(\log(n)) \subset \mathcal{O}(n) \subset \mathcal{O}(n \log(n)) \\ &\subset \mathcal{O}(n^k) \subset \mathcal{O}(n^{k'}) \subset \mathcal{O}(a^n) \subset \mathcal{O}(a'^n) \end{aligned}$$

Sommes Usuelles

- $\sum_{0 \leq k \leq n} k = \frac{n(n+1)}{2} = \Theta(n^2)$
- $\sum_{0 \leq k \leq n} k^p = \Theta(n^{p+1})$ pour $p \in \mathbb{N}^*$
- $\sum_{0 \leq k \leq n} 2^k = 2^{n+1} - 1 = \Theta(2^n)$

Sommes Usuelles

- $\sum_{1 \leq k \leq n} \frac{1}{k} = H_n = \Theta(\log(n))$
- $\sum_{0 \leq k \leq n} \frac{1}{2^k} = 2 - \frac{1}{2^n} = \Theta(1)$
- $\sum_{1 \leq k \leq n} \log(k) = \log(n!) = \Theta(n \log(n))$

Analyse de Complexité

Voici la façon de calculer les coûts pour chacun des outils de programmation qui ont été vus pour l'instant :

- **Opérations élémentaires** (opérations sur les valeurs, numériques, comparaisons) : complexité constante $\mathcal{O}(1)$
- **Instructions conditionnelles** : on considère les complexités de chacun des blocs conditionnels et on majore par le maximum.


Analyse de Complexité

Voici la façon de calculer les coûts pour chacun des outils de programmation qui ont été vus pour l'instant :

- **Fonctions** : coût de la fonction pour la valeur sur laquelle elle est appelée, chaque fois qu'elle est appelée sur une valeur donnée dans le programme.
- **Boucle** : la complexité de la boucle est égale à la somme des complexités de chaque itération. Si on a un majorant pour chaque itération, la complexité totale de la boucle est donc majorée par la somme des majorants.

Analyse de Complexité

Attention



Une façon de majorer la complexité d'une exécution de boucle est d'avoir un majorant global de toutes les itérations, indépendant de celles-ci et de le multiplier par le nombre de tours de boucle ; cependant, cela peut amener à avoir un majorant qui n'est pas optimal.

Plusieurs cas particuliers peuvent apparaître dans le calcul de complexité de boucles. On peut notamment trouver les cas suivants :

Analyse de Complexité

Lors de l'utilisation d'une boucle `for`, dans laquelle le coût de chaque itération est constant ou majoré par une constante indépendante du nombre n d'itérations, on peut majorer la complexité totale de l'exécution de la boucle par n .

Si on a une incrémentation ou une décrémentation de k unités au lieu d'une seule, on va avoir $\lfloor \frac{n}{k} \rfloor$ itérations de boucle

Exemple

Pour l'exponentiation naïve : un appel à la fonction fait p tours de boucles, où p est la puissance indiquée. On fait donc p multiplications au total.

Analyse de Complexité

Dans le cas des boucles `while`, on pourra chercher à borner le nombre d'itérations de la boucle si c'est possible, en fonction de l'évolution des variables intervenant dans la condition d'arrêt.

Exemple

On considère l'algorithme qui calcule les termes de la suite de Fibonacci jusqu'à un seuil n donné :

Analyse de Complexité

```
int fibo(int n)
{
    int a = 0;
    int b = 1;
    while (b < n){
        b = a + b;
        a = b - a;
    }
    return a;
}
```

on peut estimer la complexité algorithmique à l'aide de raisonnements mathématiques non-triviaux.

Analyse de Complexité

Dans le cas de deux boucles consécutives, les complexités s'ajoutent. L'ordre de grandeur de l'exécution successive des deux boucles est le plus grand des deux ordres de grandeur de boucles.

Analyse de Complexité

Lorsque l'on a une boucle à l'intérieur d'une autre boucle :

- Si on a n_1 itérations de la boucle externe, et si le nombre n_2 de tours de la boucle interne ne dépend pas du numéro d'itération $i < n_1$ de la boucle externe, on a $n_1 \times n_2$ itérations du bloc à l'intérieur de la boucle interne ;
- Sinon, si le nombre n_2 d'itérations de la boucle interne varie selon l'étape d'itération i , on calcule la valeur de $n_2(i)$ pour chaque i puis la somme $\sum_{1 \leq i \leq n_1} n_2(i)$.