Chapitre 1

Complexité algorithmique

Djaouida Zaouche-Dahmani, EISTI
Tahar Gherbi, EISTI
Stefan Bornhofen, EISTI
Bartholomew George, EISTI

Ce premier chapitre est une introduction à la notion de complexité algorithmique, sorte de quantification de la performance d'un algorithme.

1.1 Généralités sur la complexité

Une mesure de complexité d'un algorithme est généralement une estimation de son temps de calcul ou l'espace mémoire utilisée. Par ailleurs, on distinguera deux sortes de complexité, selon que l'on s'intéresse au temps d'exécution ou à l'espace mémoire occupé [1].

1. *Complexité temporelle :* la complexité temporelle d'un algorithme est le nombre d'opérations élémentaires qui le composent. On appelle opération élémentaire une opération dont le coût est constant. Toutes les opérations élémentaires sont considérées à égalité de coût.
   Exemples :
   — affectation,
   — addition,
   — multiplication,
   — comparaison.
2. *Complexité spatiale :* la complexité en espace est la taille de la mémoire nécessaire pour stocker les différentes structures de données utilisées lors de l'exécution de l'algorithme.

Il est à noter que le nombre d'opérations ou la taille de la mémoire est calculé(e) en fonction de la taille du problème à résoudre, notée n. Par exemple, dans le cas du tri d'un tableau, la taille du problème est égale au nombre d'éléments. De plus, le calcul de complexité se fait *indépendamment* [1] :
— de la machine sur laquelle l'algorithme s'exécute.
— du langage de programmation utilisé.

Notations :
— A est un algorithme.
— $D(n)$ est l'ensemble des données de taille n.
— $P(d)$ probabilité de la donnée d.
— $Coût_A(d)$ est le coût de l'algorithme pour la donnée d.

1.2 Calcul de complexité : algorithmes non récursifs

Dans cette partie, nous effectuons nos premiers calculs de complexité. Nous traiterons ici le cas des algorithmes non récursifs, ceux récursifs seront étudiés dans le prochain chapitre.

1.2.1 Algorithmes sans structures de contrôle

Le calcul de la complexité consiste tout simplement à dénombrer le nombre d'opérations successives que possède un algorithme donné.

Exemple :
Considérons la fonction de conversion de température celsius en fahrenheit :

Def $celsius\_farenheight(C)$:
$F = C * 1.8 + 32$
$return\ F$

Nous pouvons dénombrer deux opérations arithmétiques et une affectation pour la fonction $celsius\_farenheight(C)$*. Il s'ensuit que sa complexité, notée* $T(n)$*, est égale à 3.*

1.2.2 Le cas des structures conditionnelles

Contrairement aux algorithmes sans structures de contrôle, où la complexité est constante, dans ce cas, la complexité dépend des données. Pour cela, nous distinguons trois formes de complexité en temps :

1. *La complexité dans le meilleur des cas* : c'est la situation la plus favorable, qui correspond, par exemple, à la recherche d'un élément situé à la première position d'un tableau. Formellement, la complexité dans le meilleur des cas est décrite par :
$$T(n) = min_{d \in D(n)}\{Co\hat{u}t_A(d)\}$$
2. *La complexité dans le pire des cas* : c'est la situation la plus défavorable, qui correspond, par exemple, à la recherche d'un élément dans un tableau alors qu'il n'y figure pas. Formellement, la complexité dans le pire des cas est décrite par :
$$T(n) = max_{d \in D(n)}\{Co\hat{u}t_A(d)\}$$
3. *La complexité en moyenne :* on suppose que les données sont réparties selon une certaine loi de probabilités p. Formellement, la complexité en moyenne est décrite par :
$$T(n) = \Sigma_{d \in D(n)}\{p(d) * Co\hat{u}t_A(d)\}$$

On calculera le plus souvent la complexité dans le pire des cas, car elle est la plus pertinente. Il vaut mieux en effet toujours envisager le pire.

Exemple :
Considérons la fonction f définie comme suit :

```
def f(x, y) :
    if x > y :
        h = x * x
    else :
        h = y * y + 1
    return h
```

Le test de la conditionnelle comporte une comparaison. La première alternative possède une opération arithmétique et une affectation, alors que la seconde deux opérations arithmétiques et une affectation. Ainsi le maximum des coûts des différentes alternatives est 3. Donc on a $T(n) = 4$ pour le pire des cas, $T(n) = 3$ pour le meilleur des cas et $T(n) = 3.5$ pour le cas moyen.

1.2.3 Le cas des structures itératives

Le temps d'exécution d'une boucle est égal à la somme du coût du test, corps de la boucle et la gestion du compteur. Le calcul de la complexité d'une boucle doit tenir compte du nombre de ses itérations qui peut être connu ou non. Dans ce qui suit, nous étudions la complexité de quelques exemples itératifs non récursifs.

Exemples :

1. La fonction *somme_1* calcule la somme des n premiers nombres.
```
def somme_1(n) :
    S = 0
    for i in range(n) :
        S = S + i
    return S
```
2. La fonction *somme_2* calcule aussi la somme des n premiers nombres.
```
def somme_2(n) :
    S = n * (n + 1)//2
    return S
```

Comparer les complexités de *somme_1* et *somme_2*.

1.3 Comportement asymptotique des fonctions de réference

Les fonctions exprimant une complexité sont définies sur N et à valeurs dans R^+. Les définitions suivantes permettent de comparer le comportement à l'infini de deux fonctions définies

sur N. Plus précisément, il s'agit de critères pour affirmer qu'une fonction en domine une autre, ou au contraire est du même ordre de grandeur, voir même équivalente [2].

1.3.1 Notations asymptotiques

Dans ce qui suit, nous introduisons les notations aux ensembles $O(g)$, $\Omega(g)$ et $\Theta(g)$, les 3 notations de mesure de complexité.

Borne supérieure aymptotique

$O(g(n)) = \{f : N \to R^+ | \exists k > 0 \; et \; n_0 \geq 0 \;\; tels \; que \; \forall n \geq n_0, \; 0 \leq f(n) \leq k * g(n)\}$

Si une fonction $f(n) \in O(g(n))$, on dit que $g(n)$ est une borne supérieure aymptotique pour $f(n)$, ou encore $g(n)$ domine $f(n)$. On note abusivement $f(n) = O(g(n))$.

Borne inférieure aymptotique

$\Omega(g(n)) = \{f : N \to R^+ | \exists k > 0 \; et \; n_0 \geq 0 \;\; tels \; que \; \forall n \geq n_0, \; 0 \leq k * g(n) \leq f(n)\}$

Si une fonction $f(n) \in \Omega(g(n))$, on dit que $g(n)$ est une borne inférieure aymptotique pour $f(n)$. On note abusivement $f(n) = \Omega(g(n))$.

Borne aymptotique

$\Theta(g(n)) = \{f : N \to R^+ | \exists k_1 > 0, k_2 > 0 \; et \; n_0 \geq 0 \; tels \; que \; \forall n \geq n_0, \; 0 \leq k_1 * g(n) \leq f(n) \leq k_2 * g(n)\}$

Si une fonction $f(n) \in \Theta(g(n))$, on dit que $g(n)$ est une borne aymptotique pour $f(n)$. On note abusivement $f(n) = \Theta(g(n))$.

Exercices :

1. Donner une interprétation graphique du grand O.
2. Donner une interprétation graphique du grand $Theta$.
3. Donner une interprétation graphique du grand $Omega$.
4. Montrer que $n^2 = O(10^5 n^3)$.
5. Montrer que $n^2 = O(10^{-5} n^3)$.
6. Montrer que $12n^2 - 13n + 17 = O(n^2)$.
7. Montrer que $17n^4 + 12n^2 - 13n + 17 = O(n^4)$.
8. Montrer que $3^{1000} * 2^{n+1000} = O(2^n)$.
9. Montrer que $f(n) \in O(g(n)) \Rightarrow g(n) \in \Omega(f(n))$.
10. Montrer que $f(n) \in \Theta(g(n)) \Leftrightarrow f(n) \in O(g(n)) \; et \; f(n) \in \Omega(g(n))$.
11. Montrer que $7 * f(n) + 14 * g(n) \in \Theta(max(f(n), g(n))$.

1.4 Classes de complexité

Il est possible de définir des classes complexités . Des algorithmes appartenant à une même classe seront alors considérés comme de complexité équivalente. Cela signifiera que l'on considèrera qu'ils ont la même efficacité [2].

Le tableau suivant récapitule les complexités de référence.

O	Classe
1	constant
$log(n)$	logarithmique
n	linéaire
$nlog(n)$	quasi-linéaire
n^2	quadratique
n^3	cubique
2^n	exponentiel
$n!$	factoriel

Exercice :

1. En supposant que $n = 100$ et que la durée d'exécution d'une opération élémentaire est $10^{-6}s$, rajouter une colonne pour estimer le temps d'exécution de chaque classe.
2. Classer les fonctions suivantes dans l'ordre croissant de leur valeur asymptotique. Justifier votre réponse : n^{logn}, $n!$, n^3logn, 2^{2n}, n^n, 2^n, $n^{\sqrt{n}}$ et $log(n^3)$.

1.4.1 Exemples d'algorithmes itératifs

• Complexité en fonction de deux paramètres

Dans cette partie, il vous est demandé de calculer la complexité des algorithmes décrits ci-dessous. Dans ce qui suit, m et n sont deux entiers positifs.

```
def Algorithm_A :
    i = 1
    j = 1
    while (i ≤ m) and (j ≤ n) :
        i = i + 1
        j = j + 1
```

```
def Algorithm_B :
    i = 1
    j = 1
    while (i ≤ m) or (j ≤ n) :
        i = i + 1
        j = j + 1
```

```
def Algorithm_C :
    i = 1
    j = 1
while (j ≤ n) :
        if i ≤ m :
            i = i + 1
        else :
            j = j + 1
```

```
def Algorithm_D :
    i = 1
    j = 1
while (j ≤ n) :
        if i ≤ m :
            i = i + 1
        else :
```

$j = j + 1$
$i = 1$

- **Recherche séquentielle d'une valeur dans un tableau**

Le calcul de la complexité se fait comme suit :
— Pire des cas : il s'agit d'un tableau où x n'y figure pas. Le nombre d'itérations est alors n. Donc $T(n) = O(n)$.
— Meilleur des cas : il s'agit d'un tableau où x est le premier élément du tableau. Donc $T(n) = O(1)$.
— Complexité moyenne : on suppose que les données sont réparties selon la probabilité $\frac{1}{n}$. Donc $T(n) = \Sigma_{i=1}^{n} \frac{1}{n} * i = \frac{1}{n} * \frac{n(n+1)}{2} = \frac{n+1}{2}$. Il s'ensuit que $T(n) = O(n)$

- **Recherche dichotomique d'une valeur dans un tableau trié**

Nous considérons un tableau trié et étudions la complexité de la recherche dichotomqiue d'une valeur dans le tableau.

def *recherche_dic*(x, t) :
    $d = 0$
    $f = len(t) - 1$
    while $(d \leq f)$:
        $m = (d + f)//2$
        if $t[m] == x$:
            return 0
        if $t[m] < x$:
            $d = m + 1$
        else :
            $f = m - 1$
    return -1

On suppose que le tableau contient $n = 2^k$ éléments (où k est un entier positif). Soit v_n le nombre maximum d'itérations de la boucle de *recherche_dic*(x, t). Pour $n = 1$, nous avons $v_1 = 1$. Pour $n = 2^k$, $k > 0$, le tableau est divisé en deux sous-tableaux possédant au plus 2^{k-1} éléments et $v_{2^k} \leq 1 + v_{2^{k-1}}$. On en déduit que $v_{2^k} \leq k + v_{2^0}$. On a $T(n) = O(log_2(n)) = O(log(n))$. Nous avons considéré que n était une puissance de 2. Si ce n'est pas le cas, il faut alors l'encadrer entre deux puissances de 2 , puis effectuer le raisonnement précédent sur ces deux puissances.

- **Tri de sélection non récursif**

Si l'on s'intéresse qu'au nombre des instructions colorées, la complexité pour le pire des cas du tri de sélection est calculée de la manière suivante :

def *tri***Selection**(t) :
    $n = len(t)$
    for i *in* $range(n - 1)$:
        $indMini = i$
        for j *in* $range(i + 1, n)$:
            if $t[j] < t[indMini]$:
                $indMini = j$
        if $i! = indMini$:
            $x = t[i]$
            $t[i] = t[indMini]$
            $t[indMini] = x$

Le calcul de la complexité se fait comme suit :

— Pire des cas : il s'agit d'un tableau trié dans l'ordre décroissant. Donc $T(n)= 5(n-1) + \Sigma_{i=1}^{n-1} 2 * i = 5(n-1) + \frac{2n(n-1)}{2} = 5n - 5 + n^2 - n = n^2 + 4n - 5$. Donc $T(n) = O(n^2)$.

- **Tri d'insertion non récursif**

Ecrire une fonction python pour le tri par insertion et étudier la complexité du tri d'insertion.

- **Produit matriciel**

On considère deux matrices A et B de dimensions respectives (m, n) et (n, p). Le produit de A par B est une matrice C de matrice (m, n) définie par :

$$C_{i,j} = \Sigma_{k=1}^{n} A_{i,k} * B_{k,j}$$

Ecrire un algorithme pour calculer le produit matriciel et donner sa complexité.

1.5 Bibliographie

[1] Cormen T.H., Leiserson C.E., Rivest R.L. et Stein C. Algorithmique. Dunod. 2010, 1188 pages.

[2] *https* : *//www.supinfo.com/cours/2ADS/chapitres/01−notion−complexite−algorithmique.*

Chapitre 2

Complexité des algorithmes récursifs

Ce deuxième chapitre est dédié au calcul de la complexité des algorithmes récursifs. Nous abordons d'abord des algorithmes récursifs simples, ensuite nous présenterons des algorithmes récursifs complexes.

2.1 Récursivités simples

Un algorithme récursif consiste à décomposer un problème en sous-problèmes de même nature que le problème initial et à recomposer les solutions des sous-problèmes. La complexité d'un algorithme récursif est une suite définie par récurrence.

• **Complexité de la fonction factorielle récursive**

def $factorielleRecursive(n)$:
  if $n == 0$:
    $return\ 1$
  else :
    $return\ n * factorielleRecursive(n-1)$

Calcul de la complexité :
- Si $n = 0$, seule une comparaison a lieu. Donc $T(n) = 1$.
- Si $n > 0$, une comparaison, une multiplication et une soustraction sont effectuées lors de l'appel récursif.

De ce fait, nous pouvons représenter cet algorithme par l'équation récurrente suivante : $T(n) = T(n-1) + 3$. Ainsi, la résolution de la suite arithmétique donne lieu à $T(n) = 3n + 1$. Donc $T(n) = \Theta(n)$, cet algorithme est linéaire.

• **Complexité de l'algorithme classique pour des tours de Hanoï**

def $hanoi(de, vers, via, n)$:
  if $n == 1$:
    $print(de, "vers", vers)$
  else :
    $hanoi(de, via, vers, n-1)$
    $print(de, "vers", vers)$
    $hanoi(via, vers, de, n-1)$

Calcul de la complexité :
- Si $n = 1$, une comparaison est réalisée (nous ignorons le coût de l'affichage). Donc $T(n) = 1$.
- Si $n > 0$, un test est réalisé lors de l'appel récursif. Nous pouvons remarquer que $T(n) = 2 * T(n-1) + 1$.

Pour résoudre $T(n)$, nous considérons les équations suivantes :
$T(n) = 2T(n-1) + 1$
$T(n-1) = 2T(n-2) + 1$

...
$T(1) = 1$
$T(n) = \Sigma_0^{n-1}\ 2^k$.
Donc $T(n) = \Theta(2^n)$, cet algorithme est exponentiel.

2.2 Suites récurrentes linéaires

La complexité de certains algorithmes récursifs tels que la fonction Fibonnaci est une *suite récurrente linéaire homogène d'ordre deux*. Nous montrons brièvement comment résoudre une suite recurrente linéaire d'ordre deux [2].

Définition : une suite u_n est une suite récurrente linéaire d'ordre 2 s'il existe deux nombres a et b tels que, pour tout entier n, on a

$$u_{n+2} = a * u_{n+1} + b * u_n.$$

Pour étudier ces suites, nous utilisons l'équation caractéristique de la suite, définie par $x^2 - a * x - b = 0$. Celle-ci permet de résoudre la suite de la manière suivante :

1. *Premier cas :* l'équation caractéristique admet deux solutions distinctes x_1 et x_2. Il existe alors deux réels λ et μ tels que, pour tout entier n, on a $u_n = \lambda * x_1^n + \mu * x_2^n$. Les réels λ et μ peuvent être déterminés à partir des valeurs de u_0 et u_1.
2. *Deuxième cas :* l'équation caractéristique admet une racine double x. Il existe alors deux réels λ et μ tels que, pour tout entier n, on a $u_n = \lambda * x^n + \mu * n * x^n$.
3. *Troisième cas :* l'équation caractéristique n'a pas de solution. Ce cas ne se présente pas dans les calculs de complexité [2].

• Complexité de la fonction fibonnaci

La fonction fibonnaci est définie comme suit :

$$fib(n) = \begin{cases} 0 & si\ n = 0 \\ 1 & si\ n = 1 \\ fib(n-1) + fib(n-2) & si\ n \geq 2 \end{cases}$$

Il est évident que la complexité de la fonction fibonnaci est donnée par l'équation récurrente $T(n) = T(n-1) + T(n-2) + 5$, $n \geq 2$, (deux tests, deux soustractions et une addition), avec $T(0) = 0$ et $T(1) = 1$. Il a été montré que la résolution de cette équation consiste à résoudre l'équation $T(n) = T(n-1) + T(n-2)$, i.e. on ignore la partie constante [1][2]. En appliquant la technique de résolution d'une suite récurrente linéaire homogène d'ordre deux, on a

$$\forall n \in N,\ T(n) = \frac{1}{\sqrt{5}}(\frac{1+\sqrt{5}}{2})^n - \frac{1}{\sqrt{5}}(\frac{1-\sqrt{5}}{2})^n$$

A partir de la relation que nous venons d'illustrer, il est possible de déduire que $T(n) \geq \frac{1}{\sqrt{5}}((\frac{1+\sqrt{5}}{2})^n - 1)$. Cette relation sera utilisée pour le calcul de la complexité du PGCD que nous présentons dans ce qui suit.

• Complexité de la fonction PGCD

Le calcul de la complexité de l'algorithme d'euclide pour le calcul du PGCD (plus grand commun diviseur) est un peu délicat, car il n'est pas possible de le représenter par une équation récurrente. Nous allons montrer comment utiliser la suite fibonnaci pour analyser la complexité du PGCD.

```
def pgcd(x, y) :
    if y == 0 :
        return x
```

else :

return $pgcd(y, x \ mod \ y)$

Nous passons en revue les principales étapes pour analyser et estimer la complexité du PGCD :

- Nous pouvons aisément vérifier que $pgcd(x, 0) = x$, $\forall x \in N$.
- $nb\_div(x, y)$ représente le nombre de divisions (c-a-d *mod*) effectuées par l'algorithme $pgcd(x, y)$.
- Nous pouvons facilement prouver que $nb\_div(x, 0) = 0$, $\forall x \in N$.
- Si $nb\_div(x, y) = k$, avec $x > y > 0$, alors $nb\_div(y, x \ mod \ y) = k - 1$. En effet, par définition du pgcd, si l'algorithme a effectué k divisions pour calculer $pgcd(x, y)$, cela signifie que k englobe la division $x \ mod \ y$ et donc $k - 1$ divisions pour $pgcd(y, x \ mod \ y)$. Ainsi $nb\_div(y, x \ mod \ y) = k - 1$.
- Nous allons montrer que pour $x > y > 0$, si $nb\_div(x, y) = k \Rightarrow x \geq F_{k+2}$ où F_{k+2} est le $(k + 2)^{\text{ème}}$ terme de la suite Fibonacci.
  1. Pour $k = 0$, aucune division n'est effectuée. Il s'ensuit que $y = 0$. Comme $x > y$, nous pouvons écrire que $x \geq 1 = F_2$.
  2. Pour $k = 1$, une division est effectuée. Il s'ensuit que $y \geq 1$ et que $x = q * y$, $q > 1$, c-a-d $x \ mod \ y = 0$. Comme $x > y$, nous pouvons écrire que $x \geq 2 = F_3$.
  3. Supposons que la propriété $nb\_div(x, y) = j \Rightarrow x \geq F_{j+2}$ est vraie pour tout $j \leq k$.
  4. Posons $pgcd(y, z) = k$, avec $z = x \ mod \ y$. Donc, d'après l'hypothèse de récursion, on a : $y \geq F_{k+2}$.
  5. On peut affirmer, par définition du pgcd, que $pgcd(z, y \ mod(x \ mod \ y)) = k - 1$. Donc, d'après l'hypothèse de récursion, on a : $z \geq F_{k+1}$.
  6. Nous avons $x = q * y + z$, avec $q > 0$, avec $0 \leq z < y$. Donc on en déduit que $x \geq y + z \geq F_{k+2} + F_{k+1} = F_{k+3}$. La propriété est donc vraie.
  7. On a $x \geq \frac{1}{\sqrt{5}}(\phi^{k+3} - 1)$, avec $\phi = (\frac{1+\sqrt{5}}{2})$. Nous pouvons aussi affirmer que $\sqrt{(}5)x+1 \geq \phi^{k+3}$. On obtient que $k \leq log_\phi(\sqrt{5} + 1) - 3$.
  8. La complexité de l'algorithme PGCD est donc O(log x).

2.3 Algorithmes de type "diviser pour régner"

Dans cette partie, nous étudions plusieurs méthodes pour le calcul de la complexité des algorithmes conçus selon l'approche "diviser" pour "régner". Rappelons que cette approche consiste à :

- *Diviser* : diviser le problème en sous-problèmes de même nature que le problème initial.
- *Régner* : résoudre récursivement les sous problèmes.
- *Combiner* : combiner les solutions des sous-problèmes pour produire la solution du problème initial.

Avant d'aborder la description théorique du calcul de la complexité des algorithmes de type "diviser pour régner", nous étudions le tri par fusion qui est un exemple de ce type d'algorithme.

2.3.1 Tri par fusion

```
def fusion(t, first, middle, last) :
    pro = [ ]
    i = first
    j = middle + 1
    while (i ≤ middle) and (j ≤ last) :
        if t[i] ≤ t[j] :
            pro.append(t[i])
            i+ = 1
        else :
            pro.append(t[j])
            j+ = 1
    while i ≤ middle :
```

```
            pro.append(t[i])
            i+ = 1
    while j ≤ last :
            pro.append(t[j])
            j+ = 1
    for  k in range(last − first + 1) :
            t[first + k] = pro[k]


def triFusionRec(t, first, last) :
    if first < last :
        triFusionRec(t, first, (first + last)//2)
        triFusionRec(t, ((first + last)//2) + 1, last)
        fusion(t, first, (first + last)//2, last)
```

• Explications :
L'algorithme décrit ci-dessus comporte les étapes suivantes :

1. Diviser le tableau en deux tableaux de même taille (à un élément près).
2. Trier récursivement les deux tableaux.
3. Fusionner les deux-tableaux triés en un seul.
4. Le coût de la fusion est en $\Theta(n)$.
5. Par conséquent, $T(n) = 2T(\frac{n}{2}) + \Theta(n)$ et $T(1) = 1$.

• Déroulement et généralisation :
Supposez que $n = 2^3$.

1. Dérouler l'algorithme pour le tableau suivant :

8	7	6	5	4	3	2	1

2. Schématiser l'arbre des appels récursifs où chaque noeud est la taille du tableau.
3. Donner le coût de chaque niveau en supposant que le coût de la fusion d'un tableau de taille n est n.
4. Donner le nombre de niveaux de l'arbre et en déduire le coût du tri de fusion.
5. Reprendre les questions 2 et 4 pour un tableau de taille $= 2^k$, $k > 1$.

Nous avons considéré que n était une puissance de 2. Si ce n'est pas le cas, il faut alors l'encadrer entre deux puissances de 2 , puis effectuer le raisonnement précédent sur ces deux puissances.

2.3.2 Quelques équations récurrentes à résoudre

Résoudre les équations récurrentes suivantes :

1. $T(0) = 1$, $T(n) = 3 * T(n-1) + 12$.
2. $T(1) = 1$, $T(n) = T(n/2) + n/2$.

2.3.3 Le théorème Master

Le théorème Master permet de calculer la complexité de certains algorithmes de type "diviser pour régner" [1][2]. Voici une description sommaire du théorème.

— Soient $a \geq 1$ et $a \geq 1$, T une équation récurrente $T(n) = a * T(\frac{n}{b}) + f(n)$.

1. n est la taille du problème.
2. Le problème est divisé en a sous-problèmes.
3. La taille de chaque sous-problème est $\frac{n}{b}$.
4. $f(n)$ est le coût de la subdivision et de la combinaison des solutions des sous-problèmes.

— Dans le cas où $f(n) = \Theta(n^d)$, le théorème de Master permet de calculer $T(n)$ comme suit :

$$T(n) = \begin{cases} \Theta(n^d) & si\ a < b^d \\ \Theta(n^d * log(n)) & si\ a = b^d \\ \Theta(n^{log_b(a)}) & si\ a > b^d \end{cases}$$

Applications :

• En utilisant ce théorème, reprendre le tri par fusion.

• Reprendre l'exercice sur la multiplication de deux matrices A et B, noté $C = A * B$. On suppose que n est une puissance exacte de 2. Décomposer A, B et C en sous-matrices de taille $\frac{n}{2} * \frac{n}{2}$. L'équation $C = A * B$ peut alors s'écrire comme suit :

$$\begin{pmatrix} C_{1,1} & C_{1,2} \\ C_{2,1} & C_{2,2} \end{pmatrix} = \begin{pmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{pmatrix} \begin{pmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{pmatrix}$$

En développant cette équation, nous obtenons :

$$C_{i,j} = A_{i,1} * B_{1,j} + A_{i,2} * B_{2,j}, \text{ avec } i,\ j \in \{1, 2\}.$$

1. En se basant sur le principe *diviser pour régner* que nous venons de décrire, l'algorithme récursif pour calculer la matrice C peut être défini comme suit :

   Fonction $prod\_rec(A, B, n)$
   Si (n=1) **alors**
   $c_{11} = a_{11} * b_{11}$
   Sinon
   Définir les matrices A_{11},A_{12} , A_{21},A_{22},B_{11},B_{12} , B_{21},B_{22}
   $C_{11} = prod\_rec(A_{11}, B_{11}, \frac{n}{2}) + prod\_rec(A_{12}, B_{21}, \frac{n}{2})$
   $C_{12} = prod\_rec(A_{11}, B_{12}, \frac{n}{2}) + prod\_rec(A_{12}, B_{22}, \frac{n}{2})$
   $C_{21} = prod\_rec(A_{21}, B_{11}, \frac{n}{2}) + prod\_rec(A_{22}, B_{21}, \frac{n}{2})$
   $C_{22} = prod\_rec(A_{21}, B_{12}, \frac{n}{2}) + prod\_rec(A_{22}, B_{22}, \frac{n}{2})$
   retourner C

2. Dérouler cet algorithme pour les matrices suivantes :

$$A = \begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix} et\ B = \begin{pmatrix} -1 & -1 \\ -1 & -1 \end{pmatrix}$$

3. Dites pourquoi l'équation récurrente $8T(\frac{n}{2}) + \Theta(n^2)$ peut être associée à *prod_rec*. Calculer la complexité de votre algorithme en utilisant le théorème Master. Peut-on parler d'amélioration de complexité par rapport à l'algorithme du produit matriciel classique que nous avons abordé dans le chapitre précédent ?

2.4 Bibliographie

[1] Cormen T.H., Leiserson C.E., Rivest R.L. et Stein C. Algorithmique. Dunod. 2010, 1188 pages.

[2] *https* : *//www.supinfo.com/cours/2ADS/chapitres/01−notion−complexite−algorithmique*.

Chapitre 3

Correction et preuve de programmes

3.1 Problématique

Produire des logiciels sans erreurs est une tâche difficile, à cause de leurs tailles, des modifications et extensions dont ils peuvent faire l'objet. L'erreur pour les systèmes critiques (dans le domaine de l'aérospatial, du nucléaire, ...) n'est pas du tout tolérée. La preuve des programmes vise à vérifier *formellement* qu'un programme est correct et qu'il réalise bien ce qu'on attend de lui. Cette approche s'avère très ambitieuse et compliquée à mettre en pratique.

C'est la raison pour laquelle, on s'oriente vers des outils de preuve de programmes semi-automatiques nécessitant l'implication des concepteurs d'algorithmes et permettant de certifier la correction des programmes.

A un programme est associée une spécification décrivant le service attendu de ce programme. L'élaboration d'une telle spécification n'est pas sans poser de problèmes. Un programme est *correct* s'il est conforme à sa spécification pour tous les cas possibles.

L'analyse *automatique* et exacte de **n'importe** quel programme est *impossible* et *indécidable*. C'est la raison pour laquelle, on vérifie certaines propriétés. En plus de prouver que le programme réalise la tâche qu'on lui a confiée, on souhaite, par exemple, prouver la terminaison d'un programme ou l'absence de deadlock [1].

3.2 Validité d'un algorithme itératif

Un algorithme itératif repose sur des boucles dont il faut démontrer l'*effet* et la *terminaison* . Pour montrer l'effet d'une boucle sur les variables d'un algorithme, on a recourt à un invariant de boucle.

Définition : un invariant est une propriété logique sur les valeurs des variables qui caractérise tout ou une partie de l'état interne du programme et qui doit être *toujours* vraie.

3.2.1 Principe de la démarche à suivre

En général, les étapes à suivre se résument comme suit :

1. Analyser les boucles de l'algorithme, une par une en commençant par la *boucle la plus interne* en cas de boucles imbriquées.
2. Concevoir pour chaque boucle, un invariant de boucle.
3. Prouver que les invariants de boucle sont valides (par récurrence).
4. Utiliser les invariants de boucle pour prouver que l'algorithme se termine.
5. Utiliser les invariants de boucle pour prouver que l'algorithme calcule le résultat correct.

Pour prouver l'invariant après un nombre arbitraire d'itérations d'une certaine boucle, il faut :

1. Montrer d'abord que la proposition est vraie avant la première itération.

2. Puis montrer que si la proposition est vraie pour k itérations, elle l'est aussi pour $k+1$ itérations.

• **Somme des n premiers nombres**

def *somme*(t) :
    $S = 0$
    $n = len(t)$
    for i *in* $range(n)$:
        $S+ = t[i]$
    return S

∗ *La terminaison de la fonction somme :*
— $n > 0$ par hypothèse ;
— $i = 0$ par initialisation et est incrémenté de 1 à chaque tour de boucle ; donc i atteindra la valeur d'arrêt n après n itérations ;
— Aussi le corps de la boucle ne contient que l'addition et l'affectation qui s'exécutent chacune en un temps fini donc l'algorithme se termine.
— Donc l'algorithme se termine.

∗ *La validité de somme :*
On la démontre par récurrence. On montre qu'à chaque itération le résultat est celui recherché et ceci en formalisant l'expression du résultat d'une itération. Dans ce chapitre, on numérote les itérations à partir de 1. On considère la propriété suivante :

A la fin de l'itération i, la variable S contient la somme des i premiers éléments du tableau t".
Plus formellement : $S_i = \Sigma_{k=0}^{i-1} A[k]$, avec $i > 0$.

Cette propriété est dite : "l'invariant de la boucle". Pour prouver l'invariant que nous avons associé à la boucle, nous faisons une preuve par récurrence comme suit :

1. $S_{initial}$ =0 et $S_1 = S_{initial} + A[0]$=$A[0]$, donc la propriété est vraie pour $i = 1$.
2. On suppose que la propriété est vraie jusqu'à la fin de l'itération i, i.e. $S_i = \Sigma_{k=0}^{i-1} A[k]$ et montrons qu'elle reste vraie à l'itération $i+1$.
3. A la fin de l'itération $i+1$, la variable S contient ce qu'elle contenait à l'étape i, plus $A[i]$. Donc S_{i+1}= S_i + $A[i]$ = $\Sigma_{k=0}^{i-1} A[k]$ + $A[i]$= $\Sigma_{k=0}^{i} A[k]$. Donc la propriété est vraie à la fin de l'itération $i+1$. L'algorithme se termine à la fin de l'itération n et on aura donc calculé : S_n= $\Sigma_{k=0}^{n-1} A[k]$. L'algorithme est donc valide. De plus, la complexité est $\Theta(n)$.

Exercices :

1. Ecrire un algorithme qui calcule la somme des éléments pairs d'un tableau. Prouver la correction de votre algorithme.
2. Reprendre la recherche séquentielle d'une valeur d'un tableau. Prouver la correction de l'algorithme.

• **Tri à bulles**

Prouver la correction du tri à bulles.

def *tri_bulles*(t) :
    for i $reversed(range(1, len(t)))$:
        for j *in* $range(i-1)$:
            if $T[j+1] < T[j]$:
                $echanger(T[j+1], T[j])$

Pour la preuve du tri à bulles, nous retenons les points suivants :
— Dans un premier temps, nous considérons la *boucle interne* à laquelle nous associons l'invariant suivant : à la fin de l'itération k, $k \geq 1$, nous avons $t[k] = max\{t[j] \mid j = 0 \ldots k\}$.

— Pour $k = 1$, nous avons $t[1]$ est la maximum entre $t[0]$ et $t[1]$. Observons que, si initialement $t[1]$ n'était pas le maximum, nous aurions permuté.
— Supposons que l'invariant soit vérifié jusqu'à l'itération k de la boucle interne. Etudions l'invariant pour l'itération $k+1$. Nous avons $t[k] = max\{t[j] \mid j = 0 \dots k\}$, l'hypothèse de récursion supposée vraie à l'itération k. Lors de l'itération $k+1$, si $t[k+1]$ est plus petit que $t[k]$, une permutation est effectuée. Ainsi nous pouvons affirmer que $t[k+1] = max\{t[j] \mid j = 0 \dots k+1\}$.
— Considérons maintenant la *boucle externe*. A la fin de la première itération de la boucle externe, nous avons $t[len(t)-1] = max\{t[j] \mid j = 0 \dots len(t)-1\}$ en se basant sur l'invariant de la boucle interne, déjà prouvé. A la fin de la deuxième itération de la boucle externe, $t[len(t)-2] = max\{t[j] \mid j = 0 \dots len(t)-2\}$, ainsi de suite. Ceci permet de prouver que le tableau est trié à la sortie de la boucle externe.
— La preuve de la terminaison de l'algorithme se fait de manière similaire aux algorithmes décrits précédemment.

Exercice :
Prouver que l'algorithme du tri de sélection que nous avons présenté dans le chapitre 1 est correct.

3.3 Validité d'un algorithme récursif

Prouver un programme récursif, c'est :

1. Prouver que le résultat produit dans le cas d'arrêt est correct
2. Prouver que les résultats produits dans les cas d'appels récursifs, sous hypothèse que ces appels récursifs soient corrects et produisent bien ce qu'il est attendu d'eux.
3. Prouver la convergence, c'est-à-dire que toute séquence d'appels récursifs conduit toujours à une situation d'arrêt.

def $somme(t, n)$:
  if $n == 0$:
    $return\ 0$
  else :
    $return\ t[n-1] + somme(t, n-1)$

Pour calculer les premiers éléments d'un tableau t, il suffit d'appeler $somme(t, len(t))$.

Terminaison de *somme* **:** Nous prouvons la terminaison de *somme* par récurrence sur n : Quand n= 0, il est évident que le programme se termine sans appel récursif. On suppose qu'à l'étape n, l'algorithme termine donc $somme(t, n)$ termine et évaluons $somme(t, n+1)$. Celle-ci appelle d'abord $somme(t, n)$, puis réalise une addition entre $somme(t, n)$ et $t[n+1]$. Il est évident que $somme(t, n)$ se termine, donc l'addition se termine.

Validité de *somme* **:**
La fonction reproduit la relation de récurrence suivante :

$$S_n = \begin{cases} 0 & si\ n = 0 \\ S_{n-1} + A[n-1] & si \end{cases}$$

Le lecteur peut remarquer que prouver cette suite est une suite simple.

3.4 Bibilographie

[1] Cormen T.H., Leiserson C.E., Rivest R.L. et Stein C. Algorithmique. Dunod. 2010, 1188 pages.