Programmes : correction et complexité

Introduction

Ce premier cours reprend certaines notions importantes vues en première année :

- preuves de programmes ;
- complexité d'un programme ;

ces deux notions étant abordées dans le cadre d'un programme écrit de manière itérative ou récursive.
L'objectif est également de revoir, en particulier dans le TP associé, les bases de la programmation en python d'un point de vue pratique.

1 Correction d'un programme

1.1 Principe général

La première étape de conception d'un algorithme est sa spécification, c'est à dire l'explicitation des caractéristiques des données d'entrée et des caractéristiques du résultat attendu. Prouver la correction de cet algorithme, c'est montrer que cette spécification est respectée (c'est à dire que, sous l'hypothèse que les données d'entrée possèdent les bonnes caractéristiques, la sortie possède bien les caractéristiques attendues).
Nous n'allons pas traiter de la correction d'un algorithme en général, mais de la correction d'un algorithme constitué d'une boucle (qui constituent souvent les briques composant un algorithme plus général). L'outil pour démontrer la correction d'un algorithme constitué d'une boucle est l'invariant de boucle. Il s'agit d'une assertion ayant pour objet des variables mises en jeu dans la boucle qui, si elle est vérifiée avant un passage dans la boucle reste vraie après ce passage.
Il faut également montrer que la boucle termine (et on le fait en premier en général).
Une manière efficace (et en fait en un certain sens la manière standard) de démontrer la correction d'un programme comportant une boucle (tant que) est d'adopter la démarche suivante :

- Montrer que la boucle (tant que) se termine. Ceci est fait souvent en exhibant une suite d'entiers naturels qui décroît strictement à chaque passage dans la boucle.
- Dégager un invariant de la boucle (et montrer qu'il s'agit bien d'un invariant).
- Montrer : Conditions initiales $\Rightarrow$ Invariant réalisé
- Montrer : Condition d'arrêt et Invariant réalisé $\Rightarrow$ Résultat attendu.

1.2 Exemple

On considère la fonction suivante.

```
def mystere(x, n):
    p = n
    a = 1
    b = x
    while p != 0:
        if p % 2 == 1:
            a = a*b
        b = b*b
        p = p // 2
    return a
```

1. Exécuter à la main l'algorithme de cette fonction avec $x = 2$ et $n = 3$ puis avec $x = 2$ et $n = 4$. Que semble faire cette fonction ?
2. Montrer que la propriété $a.b^p = x^n$ est un invariant.
3. Montrer la correction partielle puis totale de cette fonction.

2 Complexité

2.1 Principe général

La complexité (en temps) d'un algorithme est le nombre d'opérations élémentaires nécessaires à son achèvement (il faut bien sûr préciser ce que l'on entend par " opération élémentaire ").
La complexité peut dépendre de la valeur des données traitées. Elle s'exprime souvent comme une fonction $C(n)$ de la taille des données (par exemple la taille n d'une liste). Plus que l'expression exacte de $C(n)$, c'est son ordre de grandeur qui nous intéresse. Nous présenterons ainsi souvent les résultats sous la forme $C(n) = O(f(n))$ ou $C(n) = \Theta(f(n))$, ce qui signifie que $C(n) = O(f(n))$ et $f(n) = O(C(n))$.
Par défaut, la complexité que l'on recherche s'entend dans le pire des cas. Sur certains exemples, il sera cependant intéressant de se demander ce qu'il en est dans le meilleur des cas ou bien en moyenne.

2.2 Exemple : valeur du i-ème élément d'une pile

Cet exemple est destiné à souligner le fait que la complexité est fortement liée au choix de la structure de données. Rappelons que le terme " tableau " désigne un type de données abstrait : c'est un n-uplet T constitué de valeurs (d'entiers par exemple, ou d'éléments de n'importe quel ensemble totalement ordonné lorsqu'il s'agit de tri) muni des fonctionnalités suivantes :

- initialisation à une taille voulue, obtention de cette taille ;
- accès à une case ;
- affectation d' une nouvelle valeur à la i-ème case.

En particulier dans cette structure l'accès à la valeur contenue dans la i-ème case est une opération élémentaire.
En python, on utilise en général des listes pour représenter des tableaux (en se restreignant aux opérations ci-dessus).

La structure de pile est une structure de données linéaire dans laquelle seul le dernier élément ajouté peut être lu ou retiré.
Dans une pile, les opérations autorisées sont les suivantes (on précise pour chacune la façon de la programmer en python, en faisant le choix de représenter une pile par une liste dont le dernier élément est le sommet de la pile) :

- Créer une pile `p` vide : `p = []` ;
- Tester si une pile est vide : `p == []` ;
- Lire le sommet de la pile `p[-1]` ;
- Dépiler le sommet de la pile (renvoie également sa valeur) : `p.pop()` ;
- Empiler un élément `a` : `p.append(a)`.

Les opérations d'empilage et de dépilage sont alors considérées comme élémentaires.

1. Écrire une fonction `lecturePile` qui prend en argument une pile `p` et un entier naturel `i` et qui renvoie la valeur du `i`-ème élément de la pile (en partant de sa base, élément numéro 0) si la hauteur de la pile est strictement supérieure à `i` et `None` sinon.
   On demande que la pile `p` ne soit pas affectée par la fonction.

```
def lecturePile(p,i):
    # On commence par  vider p et on empile ses éléments sur pileAux
    pileAux = []
    while ...
        a = p.pop()
        ...
    # On reconstruit ensuite p en  notant h le niveau auquel on empile
    # Au passage on détermine le i-ème élément, noté res
    h = 0
    res = None
    while ...
        a = ...
        ...
        if ...
            res = a
        h += 1
    return res
```

2. Quelle est la complexité de cette fonction (en fonction de la hauteur h de la pile) ?

Remarque : Clairement la structure de pile n'est pas adaptée à l'obtention rapide de cette information.

2.3 Ordres de complexité de certains algorithmes

De nombreux autres exemples de calculs de complexité ont été vus lors de l'année de sup. Il est utile de retenir les suivants (donnés par ordre croissant) :

- $O(\log(n))$, logarithmique (exemple : dichotomie) ;
- $O(n)$, linéaire (exemple : recherche d'un élément dans une liste quelconque) ;
- $O(n\log(n))$, qualifiée parfois de quasi-linéaire (exemple : tri fusion) ;
- $O(n^2)$, quadratique (exemple : tri sélection) ;
- $O(n^\alpha)$, polynômiale (exemples : les algorithmes quadratiques, le pivot de Gauss est en $O(n^3)$);
- $O(\alpha^n)$ $(\alpha > 1)$, exponentielle.

Remarque sur la complexité exponentielle : Aucun algorithme de complexité exponentielle n'est cité ci-dessus. La raison en est que cette complexité entraîne une impossibilité de mise en œuvre en pratique (dès que la taille des données dépasse un certain seuil, souvent vite atteint). Aucun algorithme classique n'est donc de complexité exponentielle !
Par contre, on sera confronté à cette complexité dans la situation où, avant d'expliquer un algorithme efficace, on nous demandera le nombre d'opérations qu'aurait nécessité un algorithme naïf.
Prenons un exemple. Le problème du sac à dos est le suivant : un sac peut contenir un poids maximal M et on dispose de n marchandises qui ont chacune un poids et une valeur. Comment faire pour remplir le sac avec un certain nombre de ces marchandises en maximisant la valeur transportée ?
La première question posée est alors souvent la suivante : si on souhaite tester toutes les combinaisons, combien d'opérations sont nécessaires ?

3 Fonctions récursives

3.1 Principe

Rappelons qu'une fonction récursive est une fonction qui s'appelle elle-même. Les appels récursifs sont stockés dans un espace mémoire appelé pile de récursivité jusqu'à ce que l'on tombe sur un appel pour un paramètre pour lequel la valeur retournée par la fonction est connue. Les valeurs pour les autres paramètres sont alors calculées de proche en proche.
Un exemple très basique est celui du calcul de $n!$:

```
def factRec(n):
    if n == 0:
        return 1
    else:
        return n * factRec(n-1)
```

Dans ce premier exemple il existe un algorithme itératif simple permettant d'obtenir le résultat. De plus un inconvénient important d'une fonction récursive est l' occupation de place mémoire (et probablement perte d'efficacité) dans la pile de récursivité et au pire saturation de cette pile (on ne peut pas calculer le résultat de la fonction précédente pour des valeurs de quelques milliers).
Alors quel est l'intérêt de cette méthode ? On illustre cela dans l'exemple qui suit.

3.2 Exemple

On peut déjà observer sur le premier exemple que la version récursive se rapproche plus de la façon dont est formulé le problème. Ce n'était pas un avantage décisif dans cet exemple mais cela peut le devenir.
Voici un exemple pour lequel la version récursive est plus facile à construire.

Il s'agit du calcul de la puissance d'un réel x^n $(n \geq 0)$ en minimisant le nombre de multiplications. Au lieu d'utiliser la relation $x^n = x \times x^{n-1}$ on fait la remarque suivante : si n est pair, $x^n = x^{n//2} \times x^{n//2}$ et si n est impair $x^n = x^{n//2} \times x^{n//2} \times x$; avec les cas de base $x^0 = 1$ et $x^1 = x$.

1. Écrire une fonction récursive `puis(x,n)` qui calcule x^n en se basant sur cette remarque.

```
def puis(x,n):# x r\'eel (flottant), n>=0
    if n==0:
        ...
    else :
        if  n%2 == 0:
            a = puis(x,n//2)
            return ...
        elif  n%2 == 1:
            ...
            ...
```

2. En considérant la multiplication comme une opération élémentaire, quel est la complexité de ce programme ?

Conclusion : L'exemple qui précéde illustre le fait que dans certaines situations, construire une fonction récursive est très simple et produit un programme facilement lisible.
Par ailleurs, l'exemple précédent illustre également une technique importante en informatique et que nous avons utilisée dans un certain nombre de situations : le principe de cette technique est " diviser pour régner " (" divide and conqueer "). On a en effet utilisé le fait que pour calculer x^n, il suffisait de savoir calculer $x^{n//2}$; on a divisé la taille du problème pour le résoudre plus facilement (jusquà tomber sur un cas évident).

3.3 Correction et terminaison

Terminaison

Pour démontrer la terminaison, il faut montrer que l'on aboutit à un des *cas de base*, c'est à dire à un cas où le calcul du résultat de la fonction ne nécessite pas d'appel à la fonction. Cela se fait souvent en exhibant une suite qui décroît strictement à chaque appel récursif et prend une valeur d'un cas de base après un certain nombres d'itérations.
Dans l'exemple `puis(x,n)`, on prend n comme valeur qui décroît à chaque appel et dont on voit facilement qu'elle aboutit à un des cas où $n = 0$.

Correction

La démonstration de la correction se rapproche d'une démonstration par récurrence. On introduit une hypothèse de récurrence $\mathcal{P}_n$ (on a pris un seul paramètre n mais il pourrait y en avoir plusieurs). On montre que dans le(s) cas de base $\mathcal{P}_n$ est vérifiée. Puis on montre que si $\mathcal{P}_n$ est vérifiée dans les cas appelés par la fonction récursive, alors $\mathcal{P}_n$ est vérifiée par le résultat de la fonction.

Exemple : prouver la correction de la fonction `puis(x,n)`.
Corrigé : Pour prouver la correction de `puis(x,n)`, la preuve prend la forme d'une récurrence forte.

- On veut montrer que la propriété suivante est vraie pour tout n : $\mathcal{P}_n$: `puis(x,n)` renvoie la valeur x^n
- si $n = 0$, c'est clairement le cas.
- Supposons, pour $n \geq 2$ que $\mathcal{P}_k$ soit vérifiée pour tout $k \in \{0, 1, \ldots, n-1\}$. Comme $n \geq 2$, $n//2 < n$, donc, d'après l'hypothèse de récurrence, lors de l'appel récursif à `puis(x,n//2)`, `puis(x,n//2)` renvoie bien $x^{n//2}$.
  alors si n est pair, `puis(x,n//2)*puis(x,n//2)` donne bien la valeur x^n et si n est impair,
  `x*puis(x,n//2)*puis(x,n//2)` donne bien la valeur x^n.
  Donc $\mathcal{P}_n$ est donc vérifiée.
- Conclusion : en application du principe de récurrence forte, $\mathcal{P}_n$ est vérifiée pour tout $n \geq 0$.

3.4 Principe général pour la complexité

Reprenons notre premier exemple.
Notons $T(n)$ le nombre d'opérations élémentaires nécessaires au calcul de `factRec(n)`. Pour calculer `factRec(n)` à partir de `factRec(n-1)`, on a besoin de deux opérations élémentaires ; on a donc $T(n) = T(n-1) + 2$. Cette suite arithmétique nous donne $T(n) = \Theta(n)$.

Principe : pour l'étude de la complexité d'une fonction fonction récursive, en notant $T(n)$ le nombre d'opérations élémentaires nécessaires pour traiter une donnée de taille n, on est amené à étudier une relation de récurrence sur $T(n)$.

Premier cas : Si cette relation est du type $T(n) = T(n-1) + C$, on obtient une complexité linéaire.

Deuxième cas : Si cette relation est du type $T(n) = \alpha T(n-1) + C$ (suite arithmético-géométrique), $\alpha > 1$, on obtient une complexité exponentielle.

Dans l'exemple du calcul rapide d'une puissance, on a vu que l'on avait $T(n) = T(n/2) + 2$ et que cela donnait une complexité de l'ordre de $\log_2 n$. C'est un cas particulier du cas typique suivant :
Troisième cas : Si cette relation est du type $T(n) = T(n/\alpha)+C$, $\alpha > 1$, on obtient une complexité logarithmique.

La situation suivante intervient dans le tri insertion par exemple :
Quatrième cas : Si cette relation est du type $T(n) = T(n-1) + C \times n$, on obtient une complexité en $O(n^2)$.

Enfin le tri fusion fournit un exemple du cas suivant (voir TP et cours de sup) :
Cinquième cas : Si cette relation est du type $T(n) = 2 * T(n/2) + C \times n$, on obtient une complexité en $O(n \ln(n))$.

3.5 Récursivité dans les tableaux

Ce dernier paragraphe concerne un point un peu plus technique.
Les algorithmes récursifs sont souvent utilisés sur des tableaux (par exemple pour les tris), ou des structures de données assez importantes. Dans ces cas, le fait de recopier tout ou partie de la structure traitée est une perte importante d'efficacité.
On illustre sur l'exemple qui suit une méthode pour éviter ces copies.

On souhaite écrire une fonction qui prend en argument un tableau et effectue, en place, la symétrie de ce tableau par rapport à son milieu.

1. Écrire une première fonction `symTableau(T)` utilisant un algorithme itératif.
2. Écrire une fonction récursive `symTableauRec(T)` dans laquelle on s'autorise la copie de sous-tableaux. Quels sont les inconvénients de cette fonction ?
3. Écrire une fonction récursive `symTableauRec2(T)` qui n'effectue aucune copie de sous-tableaux. Pour cela, on écrira tout d'abord une fonction `symTableauRec2Partiel(T,i)` qui effectue la symétrie uniquement dans `T[i:n-i]`.