Programmes : correction et complexité

Avec corrigé

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. Les exemples seront pris le plus souvent parmi les tris, qui permettent d'illustrer les différents aspects de ces notions.
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 positifs 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 : le tri sélection

Trier un tableau, c'est le transformer en un tableau contenant les mêmes nombres avec la même multiplicité classés en ordre croissant (ceci constitue la spécification du programme). On effectuera dans la mesure du possible ces tris " en place ", c'est à dire sans créer de nouveau tableau. Ceci est bien sûr préférable du point de vue de la complexité en espace.
Rappelons que le terme " tableau " désigne un type de données abstrait : c'est un n-uplet T constitué ici de nombre réels (ou d'éléments de n'importe quel ensemble totalement ordonné) muni des fonctionalités suivantes :

- accéder à une case ;
- affecter une nouvelle valeur à la i-ème case.

Pour les calculs de complexité, il est d'usage de ne considérer que les comparaisons entre deux éléments (bien qu'en toute rigueur les accès aux cases et affectations nécessitent aussi du temps).
En pratique, nous programmerons ces tris en utilisant des listes python. On ne s'autorisera comme opérations que la lecture d'une valeur dans une case, l'affectation d'une valeur dans une case, et l'échange de deux valeurs (combinaison des deux précédentes).

Le principe du tri sélection est le suivant : on détermine le plus petit élément de la liste, on l'échange avec celui d'indice 0, puis on détermine le plus petit parmi les éléments d'indice 1 à $n-1$ et on l'échange avec celui d'indice 1 ; et ainsi de suite.
Les fonctions ci-dessous permettent de l'implémenter.

```
def indiceDuMin(L,k):
    """ prend en argument une liste d'entiers
    et un indice k tel que 0<=k<len(L)-1;
    renvoie l'indice du minimum de L[k:]"""
    n = len(L)
    ind_min = k
    for i in range(k+1,n):
        if L[i] < L[ind_min]:
            ind_min = i
    return ind_min

def triSelect(L):
    """trie une liste d'entiers selon le principe du tri s\'election"""
    n = len(L)
    k = 0
    while k < n-1:
        ind = indiceDuMin(L,k)
        L[k] , L[ind] = L[ind] , L[k]
        k += 1
    # On ne retourne rien, cette fonction transforme une liste
    # pass\'ee en r\'ef\'erence
```

En admettant la correction de la fonction `indiceDuMin`, montrer celle de la fonction `triSelect`.
Corrigé :

- La boucle termine bien car k est incrémenté de 1 à chaque tour et atteint donc bien $n-1$.
- L'invariant est la conjonction des deux propriétés :
  - $I_1(k)$: `L[:k]` est trié.
  - $I_2(k)$: les éléments de `L[k:]` sont supérieurs à `L[k-1]`.
- Cet invariant est bien réalisé à l'initialisation ($k == 0$) : les deux conditions sont vides.
- $I_1(n-1)$ et $I_2(n-1)$ impliquent bien le résultat attendu (le tableau est trié).

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 (ici la taille n du tableau). 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

Reprendre l'exemple du tri sélection et déterminer la complexité de cet algorithme.
Corrigé :
On compte uniquement les comparaisons (ce qui ne change pas l'ordre de grandeur).

- La boucle interne est de longueur $n-k-1$ avec à chaque tour une comparaison.
- Cela nous donne une complexité $C(n)$:

$$C(n) = \sum_{k=0}^{n-2}(n-k-1) = \sum_{j=1}^{n-1} j = \frac{n(n-1)}{2} .$$

On obtient donc une complexité en $\mathcal{O}(n^2)$.

2.3 Ordres de complexité de certains algorithmes

Les différents types de complexité sont les suivants (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 ;
- $O(\alpha^n)$ $(\alpha > 1)$, exponentielle.

On s'intéresse aussi parfois à la complexité en espace, c'est à dire à l'espace mémoire nécessité par l'algorithme.

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. Nous verrons cela en particulier lors du cours sur les tris fusion et rapide. En attendant, 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.
2. En considérant la multiplication comme une opération élémentaire, quel est la complexité de ce programme ?

Corrigé :

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

   Remarque : particulièrement simple à écrire à partir de la remarque.

2. Notons K le nombre de D.E. de n par 2 avant d'arriver à 0 ou 1 ; qui est aussi le nombre d'appels récursifs nécessaire au calcul de x^n par notre fonction. On a :

$$n \geq 2^K \qquad \text{avec égalité si } n \text{ est une puissance de 2 .}$$

   Lors de chaque appel à la fonction récursive, on fait au plus 2 multiplications, ce qui donne en tout au plus $2K$ multiplications. Or $K \leq \log_2(n)$, d'où une complexité logarithmique en n.
   Remarque : le calcul de la puissance en utilisant $x^n = x^{n-1} * x$ donnait de l'ordre de n.

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 reverrons également lors de l'étude de certains tris : 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 $n = 1$ ou $n = 0$.

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 $n = 0$ ou $n = 1$, 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 $T(n) \leq T(n/\alpha) + C$, $\alpha > 1$, on obtient une complexité logarithmique.

Enfin le tri fusion fournit un exemple du cas suivant (voir TP) :
Quatrième cas : Si cette relation est du type $T(n) \leq 2 * T(n/2) + C \times n$, 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]`.

Corrigé :

1.
```
def symTableau(T):
    n = len(T)
    i = 0
    while i < n//2 :
        T[i] , T[n-1-i] = T[n-1-i] , T[i]
        i +=1
```

2.
```
def symTableauRec(T):
    if len(T) <= 1 :
        return T
    else :
        return [T[-1]] + symTableauRec(T[1:len(T)-1]) + [T[0]]
```

3.
```
def symTableauRec2Partiel(T,i):
    """ effectue la sym\'etrie dans T[i:n-i] """
    n = len(T)
    # cas de base : si i==n//2, ne rien faire
    if i < n//2 :
        T[i] , T[n-1-i] = T[n-1-i] , T[i]
        symTableauRec2Partiel(T,i+1)

def symTableauRec2(T):
    symTableauRec2Partiel(T,0)
```