IF5b – Analyse d'un programme

Lors des tout débuts de l'informatique, dans l'immédiat après-guerre, on travaillait directement en langage machine ce qui demandait une connaissance technique très spécifique. Le besoin s'est fait sentir de faciliter la tâche du programmeur en créant un langage intermédiaire. Arrivé à IBM en 1950, John Backus est confronté à la difficulté de noter les nombres. Il élabore la syntaxe et le fonctionnement d'un langage intermédiaire entre le programmeur et la machine et conçoit le Fortran. De très nombreux langages suivirent très rapidement

Objectifs :
- Savoir prouver la terminaison et la correction d'un algorithme.
- Acquérir des compétences sur la notion de complexité.

I – Validité d'un algorithme

I-1) Introduction

Lorsqu'on écrit un algorithme, il est impératif de vérifier que cet algorithme produit un résultat après un nombre fini d'étapes et que ce résultat est correct dans le sens où il est conforme à la spécification précisée.

Un algorithme itératif est construit avec des boucles. Le nombre de passages dans une boucle doit être fini. (Si l'algorithme est récursif, le nombre d'appels récursifs doit être fini) Deux conditions sont donc à vérifier :

- L'algorithme donne une réponse, c'est l'étude de la terminaison ;
- La réponse donnée est celle attendue, c'est l'étude de la correction.

Si les deux conditions sont satisfaites, nous disons que l'algorithme est valide. Si lorsqu'il termine, l'algorithme donne la réponse attendue on parle de correction partielle. Si la terminaison est assurée dans tous les cas et que la réponse est correcte, on parle de correction totale. Pour prouver la terminaison d'un algorithme itératif, nous disposons de la notion de variant de boucle. Pour prouver qu'un algorithme itératif est correct, nous disposons de la notion d'invariant de boucle.

I-2) Terminaison

Pour prouver la terminaison, on va utiliser la notion de variant de boucle. Ainsi on exhibe une expression dont les valeurs prises au cours des itérations constituent une suite qui converge en un nombre fini d'étapes vers une valeur satisfaisant la condition d'arrêt de la boucle.

```
Entrée [1]: x=0
            a=16
            while x**2<a:
                x=x+1
                print(x)

            1
            2
            3
            4
```

```
Entrée [5]: m=0
            p=0
            a=3
            b=2
            while m<a:
                m=m+1
                p=p+b
                print(m,p)

            1 2
            2 4
            3 6
```

Sur le premier exemple si a est négatif (ou nul), il n'y a aucun passage dans la boucle. Sinon le nombre de passage dans la boucle est fini.

Dans le deuxième exemple, il y a exactement « a » passages dans la boucle. Ici « m » joue le rôle de variant de boucle.

Ces deux programmes se terminent bien : la terminaison est assurée.

I-3) Correction

> Un invariant de boucle est une propriété vérifiée avant l'entrée dans une boucle et après chaque passage dans cette boucle.

Pour démontrer qu'une propriété est un invariant d'une boucle, on commence donc par vérifier que la propriété est vraie avant l'entrée dans la boucle. On prouve ensuite que si la propriété est vraie avant un passage dans la boucle, alors elle est vraie après ce passage. On peut alors conclure, si la terminaison est assurée, que la propriété est vraie à la sortie de la boucle. Ceci traduit le fait que la boucle réalise bien la tâche souhaitée. Reprenons l'algorithme (2) de calcul précédent une boucle conditionnelle.

Nous allons montrer que la propriété $p = m \times b$ est un invariant de boucle.

- $m = 0, p = 0 \Rightarrow p = m \times b$ *vérifiée*
- Supposons que $p = m \times b$ *vérifiée* alors au tour suivant :
  $m' = m + 1$ *et* $p' = p + b$ *d'où* : $p' = (m+1) \times b = m \times b + b = p + b$

La propriété est donc vraie après ce passage dans la boucle et à la fin du programme on aura : $p = a \times b$. La correction est vérifiée.

II – Tests d'un algorithme

II-1) Bugs

Un circuit électronique, et plus généralement un ordinateur, fait ce qu'on lui demande de faire. Si une erreur est commise lors de l'exécution d'un programme, alors cette erreur est d'une certaine manière écrite dans le programme. De plus, ce qui peut sembler être une petite erreur peut avoir des conséquences très importantes. Considérons le programme qui suit.

Entrée [12]:
```
def f(x,y):
    return int((x+y)/2)
def f2(x,y):
    return int(x+(y-x)/2)

b=f2(10.0**308,10.0**308)
a=f(10.0**308,10.0**308)
print(b)
print(a)
```

Le premier résultat « b » donne le résultat attendu alors que « a » donne une « overflow error ». Ici le problème, est facile à deviner car il dépend de la capacité maximale des flottants en 64 bits, dès fois les problèmes sont moins simples à prévoir d'où la présence de jeu de tests dans les programmes afin d'éviter ces erreurs inattendues.

II-2) Jeu de tests

Les programmes que nous écrivons ne fonctionnent pas toujours au premier essai comme nous le prévoyons.

Des tests permettent d'observer le comportement d'un programme. Est-ce qu'il produit le résultat attendu dans un cas précis ? Le débogage consiste à corriger un programme lorsque nous savons qu'il ne fonctionne pas correctement.

Afin de rendre les tests et le débogage plus simples, il est important d'y penser lors de l'écriture d'un programme. Une bonne attitude est de décomposer le programme en éléments qui peuvent chacun être testés et débogués indépendamment des autres. Ainsi certaines démarches aident à l'élaboration de tests pertinents :

- L'utilisation de fonctions ;
- La documentation des programmes ;
- L'écriture de spécifications ;
- L'utilisation d'assertions.

Des tests peuvent permettre de faire apparaître différents types d'erreurs :

- Une erreur survient lors de certains tests et pas avec d'autres, c'est un bug intermittent ;
- Une erreur survient pour chaque test, c'est un bug permanent ;
- Le programme s'arrête de manière prématurée ou ne s'arrête pas, le bug est visible.

Les messages d'erreurs affichés lors de l'exécution d'un programme permettent d'effectuer des corrections. Quand le programme s'exécute entièrement, il est possible de vérifier que le résultat obtenu est celui qui est attendu en ajoutant au programme des assertions.

Lorsque des tests, aussi nombreux soient-ils, ont été effectués avec succès, il reste encore la possibilité d'avoir un bug caché avec un programme qui produit un résultat faux dans des cas qui n'ont pas été testés.

II-3) Exemples de jeux de tests

Construire un jeu de tests consiste à définir un ensemble de données qui vont être utilisées pour vérifier que le programme produit bien le résultat attendu avec ces données. On peut utiliser la fonction « print » ou la fonction « assert » pour effectuer ces tests. On distingue plusieurs types de tests :

- Tester quelques cas simples (le type par exemple, …) ;
- Tester des valeurs interdites, des cas limites (division par zéro, …) ;
- Tester un nombre important de données (tests aléatoires, …) ;
- Tester des cas qui pourraient nécessiter un temps d'exécution important afin d'évaluer l'efficacité du programme (nombre d'itération, fonction time(), …).

Prenons le cas de la fonction « permute » vu dans IF4a et ci-dessous. La fonction « permute » fonctionne bien sauf dans le cas particulier d'une liste vide. Il faut donc compléter le code avec une condition supplémentaire.

Entrée [20]:
```
def permute(liste):
    """liste est de type list - la fonction permute le premier
    et le dernier élément et renvoie une nouvelle liste"""
    if liste==[]: return[]
    else :
        copie=liste[:] #On copie la liste
        n=len(copie)
        copie[0],copie[n-1]=copie[n-1],copie[0] #Permutation des valeurs
        return copie
```

Entrée [17]:
```
L=[1,2,3,4]
permute(L)
```

Out[17]:
```
[4, 2, 3, 1]
```

Il peut être aussi intéressant de créer une fonction « test » contenant une batterie de test. Prenons exemple sur la division euclidienne :

Entrée [38]:
```
def division(a,b):
    """a est un entier naturel, b aussi mais non nul -
    on obtient le quotient puis le reste de la division euclidienne"""
    r=a ; q=0
    while r>=b:
        r=r-b
        q=q+1
    return(q,r)

def test_division():
    """Fonction test_division qui va tester un invariant de boucle -
    a==b*q+r"""
    m=20
    for a in range(m):
        for b in range(1,m):
            q,r=division(a,b)
            assert a==b*q+r and 0<=r<b

print(division(32,5))
test_division()
```

```
(6, 2)
```

Lors de l'exécution de « test_division » aucun problème n'est signalé. L'avantage de assert c'est que l'on sait que le test a échoué par le message « AssertionError ».

Les tests présentés sont simples, mais le plus important est d'avoir toujours en tête que même avec un très grand nombre de tests variés, l'objectif n'est pas de prouver qu'il n'y a aucune erreur mais d'en trouver.

III – Complexité d'un algorithme

III-1) Introduction

Lorsqu'un algorithme est correct, il doit encore, avant d'être écrit et exécuté, satisfaire à deux impératifs en termes de consommation de ressources :

- Utiliser un espace en mémoire acceptable, on parle de complexité en espace ;
- Produire la réponse attendue en un temps acceptable, on parle de complexité temporelle.

La complexité (ou le coût) en espace corporel correspond aux tailles des variables utilisées.

Etudier la complexité temporelle consiste à évaluer le temps d'exécution d'un algorithme en fonction de la taille des données en entrée.

III-2) Temps d'exécution

Le temps d'exécution d'un programme dépend de la machine, du langage utilisé, de l'algorithme. La part de l'algorithme est obtenue par une évaluation de sa complexité temporelle.

Nous posons les règles suivantes :

- Le temps d'exécution d'une affectation, d'une opération mathématique simple, d'une comparaison constituent une unité de base ;
- Le temps d'exécution d'une suite d'instructions est la somme des temps d'exécution de chaque instruction ;
- Le temps d'exécution d'une instruction conditionnelle « si » est inférieur ou égal au maximum des temps d'exécution des instructions ;
- Le temps d'exécution d'une boucle pour i variant de 1 à p est p fois le temps d'exécution de instructions, si ce temps est constant.
- Pour une boucle tant que, l'étude se mène aussi au cas par cas.

L'évaluation du temps d'exécution d'un algorithme se réduit ainsi à une évaluation en fonction d'un nombre n, (entier représentant la taille des données en entrée), du nombre total d'opérations élémentaires noté C_n. Le niveau de complexité correspond au type de croissance de la suite C_n.

Suivant les valeurs de l'entrée, un peut pendre des valeurs très différentes. Si, par exemple, nous parcourons une liste à l'aide d'une boucle, à la recherche d'un élément, celui-ci peut se trouver en premier et nous sortons de la boucle, c'est le cas le plus favorable. Il peut se trouver à la fin de la liste, c'est le pire des cas.

III-3) Niveaux de complexité

On a rencontré des niveaux de complexité différents lors de l'étude des tris. Cependant on retrouve souvent les mêmes complexités dans les algorithmes :

- Complexité constante : $C_n \sim 1$. Le temps d'exécution est borné (indépendant de n). C'est le cas, par exemple, pour obtenir le premier élément d'une liste.
- Complexité logarithmique : $C_n \sim \log_2(n)$. C'est le cas avec la recherche dichotomique dans une liste triée.
- Complexité linéaire : $C_n \sim n$. Cet ordre de grandeur peut s'obtenir avec une boucle non conditionnelle. Par exemple, le calcul de la somme ou de la moyenne de n termes, la recherche séquentielle dans une liste non triée de longueur n, ont une complexité en n.
- Complexité log-linéaire ou linéarithmique : $C_n \sim n \times \log_2(n)$. C'est la complexité de certains algorithmes de tri (fusion, rapide,…).
- Complexité quadratique : $C_n \sim n^2$. C'est la complexité d'algorithmes construits avec deux boucles imbriquées comme certains algorithmes de tri (tri par insertion).

Il existe d'autres complexités comme la complexité exponentielle en k^n ou polynomiale en n^k.

III-4) Exemples simples

Dans les codes qui suivent, les pointillés sous-entendent un nombre fixe d'opérations.

a) Premier cas : n est la taille de la donnée, k est un nombre fixé.

```
for i in range(n):
    ...
    for j in range(k):
        ...
```

Il y a n passages dans la boucle externe. À chaque passage dans cette boucle, il y a un nombre fixe q d'opérations puis k passages dans la boucle interne. Dans la boucle interne, nous avons un nombre fixe d'opérations r. Donc pour chaque valeur de i, nous avons $q + k \times r$ opérations. Le nombre total d'opérations est donc :

$$n \times (q + k \times r)$$

$\Rightarrow$ la complexité est linéaire : $C_n \sim n$.

b) Deuxième cas : n est la taille de la donnée.

```
for i in range(n):
    ...
    for j in range(n):
        ...
```

Il y a n passages dans la boucle externe. À chaque passage dans cette boucle, il y a un nombre fixe q d'opérations puis n passages dans la boucle interne. Dans la boucle interne, nous avons un nombre fixe d'opérations r. Donc pour chaque valeur de i, nous avons $q + n \times r$ opérations et le nombre total d'opérations est :

$$n \times (q + n \times r)$$

La complexité est quadratique : $C_n \sim n^2$

c) Troisième cas : n est la taille de la donnée.

```
for i in range(n):
    ...
    for j in range(i):
        ...
```

Il y a n passages dans la boucle externe. À chaque passage dans cette boucle, pour chaque valeur de i, il y a un nombre fixe d'opérations q puis i passages dans la boucle interne. Dans la boucle interne, nous avons un nombre fixe d'opérations r. Donc pour chaque valeur de i, nous avons $q + i \times r$ opérations. Les valeurs de i sont successivement 0, 1, 2,…, n -1.

Le nombre total d'opérations est donc :

$$\begin{aligned} & q + (q + 1 \times r) + (q + 2 \times r) + \ldots + (q + (n-1) \times r) \\ & = nq + r\big(1 + 2 + \ldots + (n-1)\big) \\ & = nq + r\frac{n(n-1)}{2} \end{aligned}$$

On retrouve une complexité quadratique : $C_n \sim n^2$.

III-5) Propriétés

Des algorithmes permettant de résoudre un même problème, peuvent être rangés suivant leur ordre de complexité du plus efficace au moins efficace. Le classement des ordres de complexité doit être connu : $1 \to \log_2 n \to n \to n \times \log_2 n \to n^2$ ...

Si deux blocs d'instructions successifs ont une complexité en C_n alors la complexité totale est en C_n. Exemple : $C_n \sim n \Rightarrow C'_n \sim 2n \sim n$.

Si on répète n fois un bloc d'instructions de complexité C_n alors la complexité totale est en $n \times C_n$.

Si deux blocs d'instructions successifs ont une complexité respectivement en C_n pour le premier et en C_m pour le second alors la complexité totale est en $\max(C_n, C_m) = C_n \; ou \; C_m$.