IF4 – Ecriture et 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 :
- Concevoir des assertions pour vérifier certaines conditions.
- Savoir prouver la terminaison et la correction d'un algorithme.
- Acquérir des compétences sur la notion de complexité.

I – Instructions et spécifications

I-1) Instructions

Une instruction est un morceau de code minimal qui produit un effet. Une instruction est exécutée par une machine. Une instruction simple peut s'écrire sur une seule ligne. On peut en écrire plusieurs séparées par des points-virgules. Outre l'expression et l'affectation, quelques instructions simples sont :

- Affirmer avec *assert*, qui est suivi d'une expression, (les assertions sont précisées plus loin dans ce chapitre) ;
- Renvoyer avec *return*, qui est suivi d'une expression et s'emploie uniquement dans une fonction ;
- Arrêter avec *break*, mot isolé qui permet d'arrêter une boucle ;
- Importer avec *import*, suivi d'un nom de module, ou de fonction, appartenant à un module qui est précisé.

Une instruction composée est une instruction sur une ligne terminée par deux-points suivie d'une ou plusieurs instructions indentées : on dispose par exemple de :

- *if* qui peut être suivi de *elif*, de *else* pour exécuter des instructions selon une condition ;
- *for* et de *while* pour exécuter des instructions de manière répétée ;
- *def* pour définir une fonction.

I-2) Spécifications

Une spécification permet d'informer les utilisateurs de la tâche effectuée par la fonction, de préciser les contraintes imposées pour les paramètres et ce qui peut être attendu des résultats. Elle peut aussi préciser les messages d'erreurs affichés en cas de mauvaise utilisation.

La spécification est destinée à l'utilisateur. Elle n'est pas utilisée par l'interpréteur « Python ». On la représente par à l'aide de « docstrings » identifiés par des triples guillemets. Si on souhaite obtenir des informations de l'interpréteur, on utilisera la fonction help.

```
help(divmod)
```

```
Help on built-in function divmod in module builtins:

divmod(x, y, /)
    Return the tuple (x//y, x%y).  Invariant: div*y + mod == x.
```

La spécification d'une fonction est écrite, comme l'est un commentaire dans un programme, à l'attention des utilisateurs qui ont besoin de savoir comment l'utiliser. L'objectif est de les éclairer, de les aider à saisir rapidement le rôle d'une ou plusieurs instructions. Un choix pertinent dans les noms de variables et de fonctions participe aussi à la compréhension d'un code. Voici comment définir une fonction avec sa spécification et quelques annotations (représentées ici par #) :

```
Entrée [67]: def permute(liste):
                 """liste est de type list
                 la fonction permute le premier et le dernier élément
                 et renvoie la nouvelle liste"""
                 n=len(liste)
                 copie=liste[:] #copie de la liste originale
                 copie[0],copie[n-1]=copie[n-1],copie[0] #permutation des valeurs
                 return copie

             help(permute)
             liste2=[1,2,3]
             permute(liste2)

             Help on function permute in module __main__:

             permute(liste)
                 liste est de type list
                 la fonction permute le premier et le dernier élément
                 et renvoie la nouvelle liste


  Out[67]: [3, 2, 1]
```

Il faut bien comprendre qu'une spécification est une sorte de contrat entre l'utilisateur et l'auteur du code. L'auteur garantit un résultat sous réserve d'une utilisation correcte précisée par la spécification.

II – Annotations, assertions et tests

II-1) Annotations et commentaires

Un programme doit pouvoir être lu et relu facilement par l'auteur mais aussi par quelqu'un qui découvre le programme. Il est important pour cela d'annoter certaines lignes de code ou des blocs d'instructions afin de préciser leur rôle. On utilise pour cela un commentaire qui est une ligne de texte précédée du symbole #.

Comme c'est le cas pour une spécification, un commentaire n'est pas utilisé par l'interpréteur Python. Un choix de structure ou de méthode par exemple peut aussi être précisé et expliqué à l'aide de commentaires.

II-2) Assertion

Une spécification permet d'éclairer sur les données en entrées, le type des valeurs autorisées, la plage de valeurs acceptées. On peut ajouter des instructions qui vont arrêter le programme en cas de mauvaises utilisation. Une assertion est l'affirmation qu'une propriété est vraie. Elle est composée du mot *assert* suivi d'une expression dont la valeur est interprétée comme une valeur booléenne. Si l'expression a la valeur True il ne se passe rien, sinon le programme est interrompu et un message d'erreur s'affiche *AssertionError*. Voici un exemple simple :

```
Entrée [2]: def inverse(x):
                """x est un nombre non nul de type float/int, on renvoie l'inverse de x"""
                assert x!=0
                return 1/x

            inverse(3)

   Out[2]: 0.3333333333333333

Entrée [3]: inverse(0)

            ---------------------------------------------------------------------------
            AssertionError                            Traceback (most recent call last)
```

Dans cet autre exemple, on vérifie que k est bien une clé présente dans le dictionnaire.

```
def animaux(dico,k):
    """dico est un dictionnaire, k est une clé de ce dictionnaire"""
    assert k in dico
dico={"félin":"chat","oiseau":"faucon"}
animaux(dico,"félin")
animaux(dico,"mammifère")
```

```
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
Cell In[12], line 6
      4 dico={"félin":"chat","oiseau":"faucon"}
      5 animaux(dico,"félin")
----> 6 animaux(dico,"mammifère")

Cell In[12], line 3, in animaux(dico, k)
      1 def animaux(dico,k):
      2     """dico est un dictionnaire, k est une clé de ce dictionnaire"""
----> 3     assert k in dico

AssertionError:
```

II-3) Tests

Pour s'assurer qu'un programme fonctionne, il faut le tester, soit dans son ensemble (test système), soit morceau par morceau (test unitaire).

Pour effectuer ces tests, on peut utiliser la méthode « assert » et il faut définir un certain nombre de cas dont on connaît la réponse et vérifier que l'exécution produit bien le résultat attendu.

Tester un programme est une condition nécessaire au bon fonctionnement du code mais pas suffisante. Ce n'est pas parce que tous les tests ont réussi qu'il n'y a pas de bug, en revanche si un des tests échoue, on sait qu'il y en a au moins un ;-)

Revenons sur l'exemple de la fonction « permute » :

```
def permute(liste):
    """liste est de type list,la fonction permute le premier et le dernier élément
    et renvoie la nouvelle liste"""
    n=len(liste)
    copie=liste[:] #copie de la liste originale
    copie[0],copie[n-1]=copie[n-1],copie[0] #permutation des valeurs
    return copie

assert (permute([1,2,3,4])==[4,2,3,1])
assert permute([1])==[1]
assert permute([])==[]
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
Cell In[18], line 14
     12 assert (permute([1,2,3,4])==[4,2,3,1])
     13 assert permute([1])==[1]
---> 14 assert permute([])==[]

Cell In[18], line 6, in permute(liste)
      4 n=len(liste)
      5 copie=liste[:] #copie de la liste originale
----> 6 copie[0],copie[n-1]=copie[n-1],copie[0] #permutation des valeurs
      7 return copie

IndexError: list index out of range
```

On remarque ainsi qu'on avait pas prévu le cas de la liste vide il nous faudra donc modifier notre programme en tenant compte de ce cas ou le spécifier.

III – Terminaison et Correction

III-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.

Lorsqu'il se 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.

III-2) Variant de boucle

Dans la plupart des cas, on utilise une (ou des variables) qui va permettre de démontrer que la boucle s'arrête (variant de boucle) et que le programme est correct (invariant de boucle)

Le variant de boucle est une **expression** dont les valeurs prises au cours des itérations constituent une suite convergente en un nombre fini d'étapes vers une valeur satisfaisant la condition d'arrêt tout au long de la boucle, en général c'est une variable :

- À valeurs entières ;
- Toujours positive en entrée de boucle ;
- Et, qui diminue (ou augmente) strictement après chaque itération sans dépasser une valeur choisie.

On peut alors en conclure que la boucle se termine.

```
def div(N,d):
    q=0
    r=N
    while r>=d:
        q=q+1
        r=r-d
    return(N,q,r)

div(19,3)
```

```
(19, 6, 1)
```

Dans notre exemple la variable r :

- Reste positive tout au long de l'algorithme.
- Diminue à chaque itération.
- Décroit jusqu'à la valeur de $1 > 0$.

Ainsi la variable r joue le rôle de variant de boucle : le programme se termine.

III-3) Invariant de boucle

Un invariant de boucle est une **propriété** :

- Qui est vérifiée avant d'entrer dans la boucle,
- Qui si elle est vérifiée avant une itération est vérifiée après celle-ci,
- Qui lorsqu'elle est vérifiée en sortie de boucle permet d'en déduire que le programme est correct.

La démonstration se fait par récurrence, convenablement rédigée : initialisation, hérédité, conclusion.

```
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 cet exemple, « m » joue le rôle de variant de boucle :

- C'est un entier positif
- Qui croît jusqu'à une valeur limite « a ».

Le programme se termine bien.

Montrons que la quantité $p = m \times b$ est un invariant de boucle.

- Initialisation

$m = 0, p = 0 \Rightarrow p = m \times b$ *vérifiée* au rang 0.

- Hérédité

Supposons que $p = m \times b$ *vérifiée* alors au tour suivant :

$$m' = m + 1 \; et \; p' = p + b \; d'où :$$
$$p' = (m + 1) \times b = m \times b + b = (m + 1)b = m' \times b$$

- Conclusion

La propriété est donc vraie au rang $m + 1$. $p = m \times b$ est bien un invariant de boucle. À la fin du programme on aura : $p = a \times b$. La correction est vérifiée.

Si le programme se termine (variant de boucle) et qu'il donne le bon résultat (invariant de boucle) on dit qu'il y a : correction totale.

IV – Complexité

IV-1) Définitions

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 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.

Il existe une règle pratique qui s'observe fréquemment : pour une même tâche, on peut souvent :

- Optimiser la complexité en mémoire, au détriment de la complexité en nombre d'opérations ;
- Et inversement.

Pour les algorithmes que nous rencontrerons en pratique, c'est la complexité en nombre d'opérations qui nous posera des problèmes. On essayera de l'optimiser.

On distingue plusieurs types de complexité :

- Complexité dans le pire des cas : complexité dans le cas où toutes les itérations sont effectuées, pour toutes les données : c'est la complexité maximale, importante pour les systèmes lourds voire critiques pour lesquels le pire peut arriver ;
- Complexité dans le meilleur des cas : lorsqu'un minimum d'opérations sont effectuées, cette complexité est rarement intéressante dans une première approche ;
- Complexité en moyenne : qui utilise des notions de probabilité ; elle est utilisée par exemple pour les algorithmes du quotidien comme les moteurs de recherche.

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, C_n 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,…). *(Chapitre suivant)*
- 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

a) Boucle « for »

```
n=5
for i in range(n):
    x=i+i
    print("x=",x)
```

```
x= 0
x= 2
x= 4
x= 6
x= 8
```

Dans la boucle « for » on a deux opérations de complexité 1 (la somme et l'affichage). Vu que la boucle s'effectue deux fois :

$$C_n = 2n \sim n$$

b) Boucles imbriquées

```
n=4
for i in range(n):
    x=i+i
    for j in range(n):
        y=x*j
        print("y=",y,end=",")
```

```
y= 0,y= 0,y= 0,y= 0,y= 0,y= 2,y= 4,y= 6,y= 0,y= 4,y= 8,y= 12,y= 0,y= 6,y= 12,y= 18
```

Dans la boucle externe (celle sur i) il y a n passages avec une opération.
Dans la boucle interne (celle sur j) il y a n passages avec deux opérations.
Le nombre total d'opérations est donc :

$$n \times (1 + n \times 2) \sim n + 2n^2 \sim n^2$$

```
n=4
k=3
for i in range(n):
    x=i+i
    for j in range(k):
        y=x*j
        print("y=",y,end=",")
```

```
y= 0,y= 0,y= 0,y= 0,y= 2,y= 4,y= 0,y= 4,y= 8,y= 0,y= 6,y= 12
```

Dans la boucle externe (celle sur i) il y a n passages avec une opération.
Dans la boucle interne (celle sur j) il y a k passages avec deux opérations.
Le nombre total d'opérations est donc :

$$n \times (1 + k \times 2) \sim n + 2nk$$

Si on suppose $k \ll n$ alors $C_n \sim n$

```
n=4

for i in range(n):
    x=i+i
    for j in range(i):
        y=x*j
        print("y=",y,end=",")
```

```
y= 0,y= 0,y= 4,y= 0,y= 6,y= 12
```

Dans la boucle externe (celle sur i) il y a n passages avec une opération.
Dans la boucle interne (celle sur j) il y a i passages avec deux opérations.
Pour chaque valeur de i on a donc : $1 + i \times 2$ opérations
Le nombre total d'opérations est donc :

$$1 + (1 + 2) + (1 + 4) \ldots + (1 + (n-1) * 2) = n + 2\big(1 + 2 + \cdots + (n-1)\big)$$
$$= n + \frac{2\big((n-1)(n)\big)}{2} = n^2$$
$$\Rightarrow C_n \sim n^2$$

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 : $1 \rightarrow \log_2 n \rightarrow n \rightarrow n \times \log_2 n \rightarrow n^2 \ldots$

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 $\max(C_n, C_m) = C_n \text{ ou } C_m$.

III-6) Factorielle

```
In [2]: def fact(n):
            if n==0:
                return 1 #Le cas de base
            else:
                return n*fact(n-1)

        fact(3)

Out[2]: 6
```

Revenons sur le tri récursif de la factorielle, vérifions sa terminaison et sa correction.

- Terminaison

On considère le variant de boucle « n ». À chaque appel de la fonction récursive, il décroît d'une unité et finit par atteindre la valeur 0 correspondant à la condition d'arrêt. Le programme se termine donc dans tous les cas si n ≥ 0. Par contre, le programme ne se termine pas si n < 0.

- Correction

Soit P_n : « La fonction fact(n) retourne n! » :

- P_0 est vraie puisque c'est la condition d'arrêt.
- Supposons P_n est vraie. La fonction fact(n+1) réalise l'opération : $(n+1) \times fact\,(n)$. Comme P_n est vraie, alors le programme retourne : $(n+1) \times fact\,(n) = (n+1)n! = (n+1)!$. La propriete P_{n+1} est donc vraie.

On a démontré par récurrence que la propriété P_n est vraie pour tout entier naturel n. La correction de la fonction $fact(n)$ est donc démontrée puisque la propriété P_n est bien un invariant de boucle.

- Complexité

Le coût en temps d'un algorithme récursif est lié au nombre d'appels récursifs en fonction de n représentant le nombre ou la taille de l'objet en entrée. Ce coût peut généralement s'exprimer par une relation de récurrence.
On note C_n le coût en fonction de n et nous comptons le nombre de test et d'opérations arithmétiques.

- Pour $n = 0$, nous avons un seul test donc $C_0 = 1$
- Pour $n > 0$, nous avons un test, une multiplication et un appel de la fonction $fact\,(n-1)$. Soit $C_n = 2 + C_{n-1}$. On reconnaît une suite arithmétique de raison 2 et de premier terme $C_0 = 1$ d'où :

$$C_n = 1 + 2n \sim n$$

Le coût de la fonction factorielle est linéaire en n.