Chapitre 2 : Méthodes de programmation et analyse des algorithmes

I Expressions, instructions et effets de bord

Définition I.1
- Une **expression** est une combinaison de constantes, de variables, d'opérateurs et d'appels de fonctions qui est **évaluée** en respectant les règles de priorité pour obtenir une valeur.
- Une **instruction** est un fragment de code qui exprime une action à **exécuter** pour produire un effet sur la machine.

■ **Remarque 1** La valeur obtenue en évaluant une expression est en général utilisée pour exécuter une instruction. Les principaux exemples d'instructions sous Python sont :

- affecter une variable avec `=` ;
- définir une fonction avec `def` ;
- faire renvoyer une valeur à une fonction avec `return` ;
- arrêter une boucle avec `break` ;
- importer une bibliothèque avec `import` ;
- affirmer une condition avec `assert` ;
- `if` ... `elif` ... `else` ... ;
- `for` ... et `while` ...

■

■ **Exemple 1**
- Quelques expressions :

```
>>> 3+5
>>> [n**2 for n in range(10)]
```

- Quelques instructions :

```
if x == 0:
    x = 1
else:
    x = 1 + 1

def f(x):
    return x*x
```

■

Définition I.2 On dit qu'une opération, une fonction ou une expression a un **effet de bord** (**side effect** en anglais) si elle a un effet observable autre que le renvoie d'une valeur à l'utilisateur, qui est l'effet attendu.

■ **Remarque 2** Ces effets de bord apparaissent souvent lorsqu'on manipule des objets mutables comme les listes et les dictionnaires. ■

■ **Exemple 2**
- Voici une instruction suivie d'une expression :

```
>>> L = [1, 2, 3]
>>> 1 + L.pop()
4
```

L'appel de `L.pop()` renvoie la valeur 3 ce qui évalue l'expression à 4, mais la liste `L` est modifiée !

```
>>> L
[1, 2]
```

- Voici deux fonctions qui renvoient la même chose, mais la première a un effet de bord :

```
def ajoute(L, x):
    L.append(x)
    return L

def ajoute2(L, x):
    L = L + [x]
    return L
```

■

II Spécifications, annotations

II.1 Spécification

La **spécification** d'un programme est une documentation permettant d'informer l'utilisateur du cahier des charges que suit le programme. Pour nous, il s'agira de préciser pour chaque fonction :

- ce qu'elle attend comme arguments ;
- ce qu'elle renvoie ;
- ses effets de bord ;

On ne manquera pas non plus de compléter avec des commentaires sur des endroits particuliers.

Pour préciser les arguments attendus, on donne la **signature** de la fonction :

```
def ma_fonction(arg1 : type1, arg2 : type2, ...) -> typeR:
```

Il n'est pas indispensable de préciser les types de tous les arguments et **les types annoncés ne sont pas une contrainte** imposée à l'utilisateur : la fonction peut être appelée avec d'autres types et pourra alors renvoyer une valeur inatendue.

■ **Exemple 3**

- Par exemple, la fonction

```
def somme(x, y):
    return x + y
```

est pensée pour faire la somme de deux nombres, mais renvoie un résultat aussi pour d'autre types d'entrées : par exemple, `somme([1], [2])` renvoie la liste `[1, 2]`.
Il est important ici de préciser la signature :

- On souhaite écrire une fonction `division_euclidienne` qui prend en arguments deux entiers `a` et `b` et renvoie le quotient et le reste de la division euclidienne de `a` par `b`. La signature de la fonction est alors :

```
def division_euclidienne(
```

■

On précise l'effet de la fonction avec un commentaire venant juste après la signature : c'est la **docstring**. Ces commentaires ne sont pas utilisés par l'interpréteur et sont destinés à l'utilisateur. Celui-ci peut les consulter en utilisant la fonction `help`.

■ **Exemple 4** Par exemple pour la fonction précédente :

```
def division_euclidienne(
    '''



    '''
```

On peut alors exécuter `help(division_euclidienne)` pour obtenir la spécification de la fonction.
■

II.2 Précondition, postcondition

Une **précondition** est une information supposée vraie avant l'exécution d'un bloc de code, et une **postcondition** est une information supposée vraie après l'exécution d'un bloc de code. Ces conditions permettent de faciliter la lecture d'une partie de code techniquement complexe.

■ **Exemple 5** Voici un exemple :

```
def racine_carree(n : int) -> int:
    ''' Renvoie la partie entière de la racine carrée de n
    Précondition : n >= 0
    '''
    x = 0 #Pour stocker le résultat
    while x*x < n:
        x = x + 1
    #Postcondition : x est la partie entière + 1 de sqrt(n)
    return x - 1
```

■

II.3 Assertions

Pour forcer la vérification d'une précondition, on utilisera parfois le mot clé `assert` suivi d'une expression de type booléen et d'une chaîne de caractères séparées d'une virgule.
L'expression est évaluée au moment où l'instruction est exécutée : si elle est vraie, on continue l'exécution normalement, sinon une `AssertionError` est renvoyée et la chaîne de caractères fournie est affichée.

■ **Exemple 6**

- Voici une fonction utilisant `assert`

```
def inverse(x : float) -> float :
    ''' renvoie l'inverse du flottant x
    '''
    assert x != 0, 'x est nul'
    return 1/x
```

- Un autre exemple est la fonction `maximum` du Chapitre 1.6 qui renvoie le maximum d'une liste non vide :

```
def maximum(
    ''' Renvoie le maximum de la liste
    '''
```

```
    return
```

■

III Terminaison, correction

III.1 Variant de boucle

Définition III.1 On dit qu'un algorithme **termine** s'il s'arrête après un nombre fini d'opérations, et ce quelles que soient les entrées.

■ **Remarque 3** Le problème de la terminaison se pose en général lorsqu'on utilise une boucle `while` ou une fonction récursive (voir Chapitre 1.7).
Une boucle `for` suivie d'un `range` termine toujours (en Python). ■

■ **Exemple 7** Prouver la terminaison n'est pas toujours une question facile. Personne ne sait démontrer la terminaison de la fonction suivante :

```
def syracuse(n : int) -> int:
    '''Renvoie le temps de vol de la suite de Syracuse
    '''
    u = n
    i = 0
    while u != 1:
        if u%2 == 0:
            u = u // 2
        else:
            u = 3*u + 1
        i = i + 1
    return i
```

■

Le principal outil pour justifier qu'une boucle termine est l'utilisation d'un variant de boucle :

Définition III.2 Un **variant de boucle** est une expression dépendant des variables du problème qui s'évalue en un entier qui doit être positif au début de chaque itération et qui diminue strictement à chaque itération.

■ **Exemple 8**

- Considérons la fonction

```
def somme(n : int) -> int:
    ''' Renvoie
    Précondition :
    '''
    s = 0
    while n>0:
        s = s + (n%10)
        n = n//10
    return s
```

Un variant de boucle est :

- Reprenons la fonction `division_euclidienne`.

```
def division_euclidienne(
    '''



    '''









    return
```

Un variant de boucle est :

■

Ainsi, si une boucle admet un variant qui vaut n au début, alors elle terminera après au maximum n itérations.

Théorème III.1 Si une boucle admet un variant, alors elle termine.

III.2 Correction

Le fait qu'un algorithme termine ne garantit pas qu'il répond au problème voulu.

Définition III.3 Un algorithme est **partiellement correct** si quelle que soit l'entrée, s'il termine alors le résultat obtenu satisfait la spécification.
Un algorithme est **correct** s'il termine et il est partiellement correct.

Pour prouver la correction, les difficultés principales sont les boucles (de tout type). On utilise dans ce cas un invariant de boucle :

Définition III.4 Un **invariant de boucle** est une propriété qui

- est vraie en entrée de la boucle ;
- si elle est vraie au début d'une itération, elle est aussi vraie à la fin.

■ **Remarque 4**

- C'est un raisonnement par récurrence !
- On identifie en général la fin d'une itération avec le début de la suivante, sauf bien sûr pour la dernière itération.

■

■ **Exemple 9**

- Voici un algorithme qui dépend des variables `a` et `b`.

```
p = 0
m = 0
#Invariant : p == m*b
while m < a:
    m = m+1
    p = p + b
```

Cet algorithme termine car :
Un invariant de boucle est : « $p = m \times b$ ». En effet :

▷ en entrée de la boucle, on a $p = 0$ et $m = 0$, et $0 = 0 \times b$;

▷ supposons qu'en entrée d'une itération, on a $p = m \times b$. Notons m' et p' les nouvelles valeurs des variables en fin d'itération : alors $m' = m + 1$ et $p' = p + b$. On a donc $m' \times b = m \times b + b = p + b = p'$, donc la propriété est vérifiée à la fin de l'itération.

Ainsi, après la boucle, on a $p = m \times b$. Or, la boucle s'arrête lorsque $m = a$. Donc $p = a \times b$.

- Reprenons l'exemple de la fonction `somme`. Un invariant de boucle est : « à la fin de la i-ième itération, `s` contient la somme des i derniers chiffres de x et `n` contient $\left\lfloor \frac{x}{10^i} \right\rfloor$ (où x est la valeur de l'argument) ». En effet :

  ▷ en entrée de la boucle :

  ▷ supposons l'invariant vrai à la fin de l'itération k, alors lors de l'itération suivante,

  En sortie de boucle, `n` vient de prendre la valeur 0, donc le nombre d'itérations vaut $\lfloor \log_{10}(x) \rfloor + 1$ qui est le nombre de chiffre de x. Ainsi, `s` vaut bien la somme des chiffres de x.

- Reprenons encore l'exemple de la fonction `division_euclidienne`. Un invariant de boucle est :

■

III.3 Tests

Même si on a prouvé la correction d'un algorithme, personne n'est à l'abris d'une faute de frappe ou d'une erreur d'indice ou autre... Il est donc important de tester ses programmes pour détecter ces erreurs.

Pour tester une fonction, le plus simple est de l'appeler avec différentes valeurs pour ses arguments. On ne peut en général pas être exhaustif, on peut :

- tester quelques cas simples ;
- tester des valeurs extrèmes, des valeurs interdites ;
- tester un nombre important de données, notamment en générant des entrées aléatoires ;

On essaye de cibler les différents problèmes qui peuvent apparaître et de partitionner le domaine d'entrée. On prend alors une valeur dans chaque partie pour chaque argument pour effectuer un test.

IV Complexité

Une fois la correction d'un algorithme théoriquement établie, se pose la question de son exécution réelle : il y a alors deux points à prendre en compte

- a-t-on assez d'espace mémoire pour exécuter l'algorithme ?
- va-t-il terminer de notre vivant ?

Il nous faut donc un outil pour évaluer le temps et la mémoire pris par l'algorithme : c'est la notion de **complexité**.
La complexité d'un algorithme dépend en général de la taille des paramètres en entrée, qu'on note d'habitude n.

IV.1 Complexité temporelle

Le temps mis pour exécuter un algorithme dépend de la taille n des entrées, mais peut aussi dépendre de la forme des entrées.

■ **Exemple 10** Considérons la fonction suivante pour trouver le maximum d'une liste :

```
def max_1(L : list):
    ''' Renvoie le maximum de L
    '''
    assert L != []
    for e in L:
        n = 0
        #Postcondition : n est le nombre d'éléments de L >= à e
        for f in L:
            if e >= f:
                n = n +1
        if n == len(L):
            return e
```

Lorsque le maximum est en première position, l'exécution de cette fonction est très rapide. Par contre, si la liste est triée par ordre croissant... ■

Définition IV.1 La **complexité temporelle dans le pire des cas** d'un algorithme est le nombre maximum u_n d'instructions élémentaires exécutées en fonction de la taille n de l'entrée.

Pour évaluer celle-ci, il nous faut d'abord identifier la forme de l'entrée correspondant au pire des cas, puis on utilise les règles suivantes :

- une instruction élémentaire (un calcul arithmétique ou booléen, une affectation, l'accès à un élément d'une liste, l'ajout d'un élément à une liste) compte pour 1 ;
- le coût d'une suite d'instructions est la somme des coûts de chaque instruction ;
- le coût d'une instruction conditionnelle est inférieur ou égal au maximum des coûts des blocs qui la compose auquel on ajoute tous les coûts d'évaluation des conditions ;
- le coût d'une boucle inconditionnelle est la somme des coûts des instructions du bloc pour chaque itération de la boucle. Lorsque ce coût ne varie pas d'un itération à l'autre, il suffit de multiplier le coût commun au nombre d'itérations ;
- le coût d'une boucle conditionnelle, c'est le même principe sauf qu'on ne connaît pas forcément le nombre total d'itérations. En général, on cherche une majoration ;
- pour des boucles imbriquées, on fait une somme double !

Il ne sert à rien d'évaluer de façon très précise cette complexité car elle va dépendre du système sur lequel est exécuté l'algorithme. D'autre part, les problèmes de temps d'exécution ne vont apparaître que lorsque la taille des entrées devient grande. On ne s'intéresse donc qu'à des ordres de grandeur lorsque n tend vers $+\infty$.

Définition IV.2 Soient (u_n) et (v_n) deux suites numériques qui ne s'annulent pas. On dit que :

- $u_n = O(v_n)$ si $\dfrac{u_n}{v_n}$ est bornée ;
- $u_n = \Theta(v_n)$ sur $u_n = O(v_n)$ et $v_n = O(u_n)$.

On va alors essayer de comparer la complexité u_n obtenue aux principaux niveaux de complexité qui sont les suivants :

- **complexité constante** : $u_n = \Theta(1)$, lorsque le temps d'exécution ne dépend pas de la taille de l'entrée ;
- **complexité logarithmique** : $u_n = \Theta(\log(n))$ (quelle que soit la base du logarithme) ;
- **complexité linéaire** : $u_n = \Theta(n)$;
- **complexité quasi-linéaire** : $u_n = \Theta(n\log(n))$;
- **complexité quadratique** : $u_n = \Theta(n^2)$;
- **complexité polynomiale** : $u_n = \Theta(n^\alpha)$ (avec $\alpha > 0$) ;
- **complexité exponentielle** : $u_n = \Theta(a^n)$ (avec $a > 1$).

■ **Remarque 5** Lorsque `L` est une liste et `e` un objet, l'instruction `e in L` teste si `e` est un élément de `L`. Attention, sa complexité est $O($`len(L)`$)$!
Par contre, si `d` est un dictionnaire, l'instruction `e in d` qui teste si `e` est une clé de `d` est en temps constant ! ■

n	10	20	30	40
$\log(n)$	0,001 s	0,0013 s	0,0015 s	0,0016 s
$\sqrt{n}$	0,003 s	0,004 s	0,005 s	0,006 s
n	0,01 s	0,02 s	0,03 s	0,04 s
n^2	0,1 s	0,4 s	0,9 s	1,6 s
2^n	1 s	17,5 min	12,4 jours	34,9 ans
$n!$	1 h	$7,7 \times 10^5$ siècles	$8,4 \times 10^{19}$ siècles	$2,6 \times 10^{35}$ siècles

Temps de calcul pour un ordinateur effectuant 1000 opérations élémentaires par seconde.

■ **Exemple 11** • Voici une fonction :

```
def est_premier(n : int) -> bool:
    ''' Renvoie True si n est premier et False sinon
    '''
    assert n >= 0
    for i in range(2, n):
        if
            return
    return
```

Le pire des cas est :
La complexité de cette fonction dans ce cas est :
On peut améliorer cette complexité :

- Quelques exemples :

```
def somme(n : int) -> int:
    s = 0
    for i in range(n):
        for j in range(n):
            s = s + i + j
    return s
```

Complexité :

```
def somme2(n : int) -> int:
    s = 0
    for i in range(n):
        for j in range(i):
            s = s + i + j
    return s
```

Complexité :

■

IV.2 Complexité en mémoire

Celle-ci est moins importante de nos jours car les capacités de stockage sont devenues très importantes. Son évaluation suit les mêmes principes que la complexité temporelle.

Exercices

Exercice 1 Pour les deux fonctions suivantes écrire la spécification précise, montrer la correction et déterminer la complexité dans le pire des cas.

```
def fonction1(L):
    i = 0
    while i < len(L)-1 and L[i] <= L[i+1]:
        i = i+1
    return i == len(L)-1
```

```
def fonction2(s, m):
    r = 0
    for i in range(len(s)):
        if s[i:i+len(s)] == m:
            r = r + 1
    return r
```

■

Exercice 2 On considère la fonction suivante.

```
def multiplication(x:int, y:int, c:int) -> int:
    m = 0
    while y != 0:
        m = m + x*(y%c)
        x = x*c
        y = y//c
    return m
```

On suppose que $c \geq 2$ et $y \geq 0$.

1. Montrer que cet algorithme termine.
2. On note `a` et `b` les valeurs d'entrées pour `x` et `y`. Montrer que la propriété : « $ab = m + xy$ »est un invariant de boucle.
3. Montrer la correction de cet algorithme. Que se passe-t-il si $y < 0$? .
4. Écrire la spécification de la fonction.

■

Exercice 3 On considère la fonction suivante qui calcule la somme de `a` et `b`.

```
def f(a, b):
    s, k = a, b
    while k > 0:
        s = s + 1
        k = k - 1
    return s
```

1. Que doit-on supposer sur `a` et `b` pour assurer la correction de la fonction `f` ? Écrire la spécification correspondante de `f`.
2. Montrer dans ce cas la correction de `f`.

■

Exercice 4

1. Pour un entier naturel `n` et un flottant `x`, de combien de multiplications a-t-on besoin pour calculer `x**n` ?
2. On considère un polynôme $P = \sum_{k=0}^{n} a_i X^i$ représenté sous Python par une liste de $n+1$ coefficients $P[i] = c_i$ pour $i \in [\![0, n]\!]$. On considère la fonction suivante pour évaluer un polynôme :

   ```
   def evalue(P:list, x:float) -> float:
       val = 0
       for i in range(len(P)):
           val = val + P[i] * x**i
       return val
   ```

   Quelle est la complexité de cette fonction ?
3. La méthode de Horner pour évaluer un polynôme consiste à remarquer que :

   $$a_n X^n + a_{n-1} X^{n-1} + \cdots + a_1 X + a_0 = ((\cdots((a_n X + a_{n-1})X + a_{n-2})X + \cdots)X + a_1)X + a_0$$

   Écrire une fonction `horner` qui met en oeuvre la remarque précédente.
4. Justifier la correction de la fonction.
5. Quelle est la complexité de `horner` ?

■