Chapitre 14 :
Ecriture d'un programme et complexité

I Ecriture d'un programme

1.1 Instructions

1.1.1 Instruction, expression et affectation

- Une **instruction** est un morceau de code qui doit être exécuté par la machine. Une instruction simple peut s'écrire sur une seule ligne. Une instruction composée se termine par : et est suivie d'au moins une instruction indentée. Une instruction n'a, en général, pas de valeur.
- Une **expression** est une instruction qui est une combinaison formée de constantes, de variables, d'opérateurs et de fonctions. Une expression a une valeur.
- Une **affectation** est une instruction qui lie un nom (existant ou nouveau) à une valeur.

Par exemple, `x=1` est une affectation mais `x==1` est une expression qui a une valeur booléenne.
Les mots `if`, `elif`, `while` doivent être suivie d'une expression afin d'avoir une valeur à tester.

1.1.2 Effet de bord

On dit qu'une fonction a un **effet de bord** si son exécution modifie un de ses paramètres. Cela peut être le cas avec des fonctions qui manipulent des objets mutables comme les listes.

➪ **Exemple 1 :**

```
def echange1(l,i,j) :
    l[i],l[j]=l[j],l[i]
    return l

>>> l=[1,2,3,4]
>>> echange1(l,0,2)
    [3, 2, 1, 4]
>>> l
    [3, 2, 1, 4]
```

La liste `l` a été modifiée lors de l'exécution de la fonction `echange1`. La fonction `echange1` a donc un effet de bord.

```
def echange2(a,b) :
    a,b=b,a
    return a,b

>>> a,b=1,2
>>> echange2(a,b)
    (2,1)
>>> a,b
    1,2
```

Une fonction analogue mais agissant sur les flottants n'a pas d'effet de bord.

```
def echange3(l,i,j) :
    L=l[ :]
    L[i],L[j]=L[j],L[i]
    return L

>>> l=[1,2,3,4]
>>> echange3(l,0,2)
    [3, 2, 1, 4]
>>> l
    [1,2,3,4]
```

La commande `L=l[ :]` définie `L` comme étant une copie (superficielle) de `l`. Ainsi, la fonction `echange3` n'a pas d'effet de bord.

En général, on évitera d'écrire des fonctions ayant un effet de bord afin de ne pas modifier, sans s'en rendre compte, des valeurs.

1.2 Spécification d'une fonction

La **spécification d'une fonction** consiste à indiquer :

- le nom de la fonction,
- l'objectif de la fonction,
- les arguments d'entrée et leur type,
- les valeurs renvoyées et leur type.

La spécification peut être écrite documentation de la fonction. La documentation est inscrite au début de la fonction et est délimité par des triples guillemets. On y accède avec la commande help.

➪ **Exemple 2 :**

```
def somme(x,y) :
    """fonction qui renvoie la somme de deux entiers ou flottants x et y"""
    return x+y

>>> help(somme)
Help on function somme in module _main_ :
somme(x, y)
    fonction qui renvoie la somme de deux entiers ou flottants x et y
```

En pratique, la spécification figure dans les énoncés. La fonction doit renvoyer le résultat demandé si les recommandations de la spécification sont vérifiées.

En reprenant l'exemple précédent, on a :

```
>>> somme('1','1')
    '11'
```

Le résultat n'est pas la somme de 1 et 1 car l'utilisateur n'a pas respecté la spécification en appliquant la fonction à des chaines de caractères.

1.3 Assertions

la spécification indique à l'utilisateur le type des données d'entrée mais elle n'impose pas ce type. Afin de l'imposer, on utilise une **assertion**. Si l'assertion n'est pas vérifiée, le programme s'arrête et renvoie un message signalant une erreur d'assertion.

➪ **Exemple 3 :**

```
def somme(x,y) :
    """fonction qui renvoie la somme de deux entiers ou flottants x et y"""
    assert (type(x)==int or type(x)==float) and (type(y)==int or type(y)==float)
    return x+y

>>> somme(1,1)
2
>>> somme('1','1')
Traceback (most recent call last) :
...
    assert (type(x)==int or type(x)==float) and (type(y)==int or type(y)==float)
AssertionError
```

1.4 Annotations et commentaires

Afin d'être relu et compris par une personne extérieure, un programme doit être commenté.
Les commentaires sont précédés du symbole #. Comme la documentation, ils ne sont pas lus lors de l'exécution du programme.
Les commentaires ne doivent pas être une paraphrase des instructions, ils peuvent :

- donner la définition des variables introduites,
- préciser le découpage d'un programme,
- justifier l'utilisation d'un type plutôt qu'un autre,
- ...

II Complexité

On veut évaluer le coût informatique de l'exécution d'un programme. Pour cela, il faut avoir défini quelles sont les ressources qui ont un coût.
Les ressources ayant un coût les plus généralement utilisées sont :

- l'espace mémoire (on parle alors de complexité spatiale)
- le temps (on parle alors de complexité temporelle).

On s'intéressera ici à la complexité temporelle.On ne raisonne pas en termes de durée mais en nombre d'opérations élémentaires. En effet, la durée d'exécution d'un programme dépend des performances de l'ordinateur ce qui n'est pas le cas du nombre d'opérations élémentaires qui ne dépend que du programme écrit.
De plus, on ne s'intéressera pas au nombre exact d'opérations élémentaires effectuées mais uniquement à leur ordre de grandeur.

2.1 Généralités

2.1.1 Opérations élémentaires

La première étape consiste à préciser quels sont les opérations élémentaires (i.e celles considérées comme ayant un coût constant indépendamment de leurs paramètres).
On considérera en général comme opérations élémentaires les opérations suivantes :

- les opérations arithmétiques (addition `+`, soustraction `-`, division `/`, multiplication `*`, division entière `//`, reste `%` ...)
- les affectations de type numérique
- les comparaisons avec des données qui ne sont pas des séquences

Le nombre d'opérations élémentaires requis par un algorithme doit être exprimé en fonction de « la taille »des données. Cette notion dépend des paramètres d'entrée et donc du problème étudié. En général, on prend :

- la longueur d'une liste, si l'algorithme prend une liste en paramètre ;
- l'entier n, lorsque le problème dépend d'un entier naturel n ;
- ...

En notant C_n le nombre d'opérations élémentaires d'un algorithme pour une donnée de taille n, on peut donc voir ce nombre d'opérations élémentaires comme une suite.

2.1.2 Outils mathématiques

Définition : Soient (u_n) et (v_n) deux suites réelles telles que (v_n) ne s'annule pas à partir d'un rang n_0.
On dit que (u_n) est dominée par (v_n) et on note $u_n = O(v_n)$ si et seulement si le quotient $\left(\frac{u_n}{v_n}\right)_{n\geq n_0}$ est borné

si et seulement si : $\exists M \in \mathbb{R}^+, \forall n \geq n_0, \left|\frac{u_n}{v_n}\right| \leq M$

si et seulement si : $\exists M \in \mathbb{R}^+, \forall n \geq n_0, \ |u_n| \leq M|v_n|$

Par exemple : $n^2 + 10n = O(n^2)$, $10000n^4 - 8n^3 = O(n^4)$.

2.1.3 Classes de complexité

On va évaluer les algorithmes en étudiant leur complexité. Afin de pouvoir les comparer, nous allons les classer par ordre croissant de complexité :
On parle de :

- **complexité constante** si le nombre d'opérations élémentaires est un $\mathbf{O(1)}$
- **complexité logarithmique** si le nombre d'opérations élémentaires est un $\mathbf{O(\ln n)}$
- **complexité linéaire** si le nombre d'opérations élémentaires est un $\mathbf{O(n)}$
- **complexité quasi-linéaire** si le nombre d'opérations élémentaires est un $\mathbf{O(n \ln n)}$
- **complexité quadratique** si le nombre d'opérations élémentaires est un $\mathbf{O(n^2)}$
- **complexité polynomiale** si le nombre d'opérations élémentaires est un $\mathbf{O(n^k)}$
- **complexité exponentielle** si le nombre d'opérations élémentaires est un $\mathbf{O(a^n)}$, avec $a > 1$

Afin de prendre conscience des différences d'échelles considérables entre ces ordres de grandeur, considérons un processeur effectuant 10^9 opérations par seconde et étudions le temps d'exécution d'un algorithme en fonction de la taille n des données (10,10^2, ...) et du nombre C_n d'opérations élémentaires requis par l'algorithme (n, $\ln n$, n^2 ...) :

C_n / taille n de l'entrée	$ln(n)$	n	$nln(n)$	n^2	n^3	2^n	$n!$
10	2 ns	10 ns	23 ns	100 ns	1 μs	1 μs	3.6 ms
10^2	5 ns	100 ns	461 ns	10 μs	1 ms	4.10^{13} années	3.10^{141} années
10^3	7 ns	1 μs	7 μs	1 ms	1 s	3.10^{284} années	
10^4	9 ns	10 μs	92 μs	100 ms	17 min		
10^6	14 ns	1 ms	14 ms	17 min	32 années		
10^8	18 ns	100 ms	2 s	116 jours	3.10^7 années		

Temps nécessaire à l'exécution d'un algorithme en fonction de sa complexité

2.1.4 Complexité dans le meilleur cas et complexité dans le pire cas

Lors du calcul de la complexité temporelle, on ne prendra pas en compte tous les cas possibles. On pourra calculer des complexité temporelles différentes :

- la complexité dans le meilleur cas,
- la complexité en moyenne,
- la complexité dans le pire cas.

On étudiera la complexité dans le pire cas. En effet, il s'agit de la complexité la plus importante car lorsqu'on applique un programme, on ne sait pas, en général, dans quel cas on se trouve.

2.2 Complexité des structures de contrôle usuelles

On notera C(opération) la complexité d'une opération :

2.2.1 Branchement conditionnel

Supposons que l'on ait un programme de la forme :

```
if condition :
    instruction1
else :
    instruction2
```

La complexité de ce programme est :

$$O(C(\text{condition})) + \max(O(C(\text{instruction1})), O(C(\text{instruction2})))$$

Remarque : Le coût de la condition est toujours pris en compte car on la teste dans tous les cas. On ne sait pas, une fois la condition testée, quelle instruction va être effectuée, c'est pourquoi, on prend en compte le maximum du coût des instructions.

2.2.2 Boucles for

Supposons que l'on ait un programme de la forme :

```
for k in range(a,b) :
    instruction
```

La complexité de ce programme est :

$$\sum_{k=a}^{b-1} O(C(\text{instruction}))$$

et dans le cas particulier où le coût de l'instruction ne dépend pas de k :

$$(b-a)O(C(\text{instruction}))$$

2.2.3 Boucles while

Supposons que l'on ait un programme de la forme :

```
while condition :
    instruction
```

La complexité de ce programme est :

$$(\text{nombre d'itérations}) \times (O(C(\text{condition})) + O(C(\text{instruction})))$$

où on suppose ici que le coût de la condition et le coût de l'instruction sont constants.

Remarque : La difficulté est d'estimer le nombre d'itérations. En pratique, on cherche uniquement un majorant du nombre d'itérations.

2.3 Cas particulier : test d'appartenance

La complexité du test d'appartenance à une séquence dépend du type de la séquence, car elle dépend de l'accès aux éléments de la séquence. On peut le remarquer sur la mesure de temps suivante :

```
import time

def TestIn(n) :
    l=[k for k in range(n)]
    d={k :k for k in range(n)}
    t=time.time()
    n in l
    print('pour une liste de longueur n :',time.time()-t)
    t=time.time()
    n in d
    print('pour un dictionnaire de longueur n :',time.time()-t)

>>> TestIn(10**8)
pour une liste de longueur n : 16.755928993225098
pour un dictionnaire de longueur n : 0.0
```

La complexité de l'opération (x in s) est :

- $O(len(s))$ si s est une liste, un tuple ou une chaîne de caractères,
- $O(1)$ si s est un dictionnaire.

2.4 Exemples

2.4.1 Exemples classiques

➪ **Exemple 4 :**

```
1 def somme(n) :
2     s=0
3     for i in range(n+1) :
4         s+=i
5     return s
```

sum(n) renvoie $\sum_{i=0}^{n} i$.

On effectue :

- 1 affectation,
- une boucle for de longueur $n+1$ avec, à chaque itération, 1 incrémentation,
- 1 renvoi.

Le nombre total d'opérations est donc : $1+(n+1)+1 = O(n)$.
La complexité est donc $O(n)$, c'est-à-dire linéaire.

➪ **Exemple 5 :**

```
1 def max(l) :
2     m=l[0]
3     imax=0
4     for i in range(len(l)) :
5         if m<l[i] :
6             m=l[i]
7             imax=i
8     return [imax,m]
```

max(l) renvoie l'indice du premier maximum et le maximum des éléments d'une liste l.

On pose : $n = len(l)$. On effectue :

- 2 affectations,
- une boucle for de longueur n avec, à chaque itération, 1 comparaison et au plus 2 affectations,
- 1 renvoi.

Le nombre total d'opérations est donc : $2+3n+1 = O(n)$.
La complexité est donc $O(n)$, c'est-à-dire linéaire.

➪ **Exemple 6 :**

```
def recherche(l,x) :
    i=0
    while i<len(l) and l[i] !=x :
        i+=1
    if i==len(l) :
        return False
    else :
        return True
```

recherche(l,x) renvoie True si l'entier x est dans la liste l et False sinon.

On pose : $n = len(l)$. On effectue :

- 1 affectation,
- une boucle while de longueur au plus n avec, à chaque itération, 2 comparaisons et 1 incrémentation,
- 1 comparaison avec, dans tous les cas, 1 renvoi.

Le nombre total d'opérations est donc : $1+3n+2 = O(n)$.
La complexité est donc $O(n)$, c'est-à-dire linéaire.

Remarque : Il s'agit ici de la complexité dans le pire des cas. Dans le meilleur des cas, la complexité est $O(1)$.

➪ **Exemple 7 :**

```
def deuxfor1(n) :
    s=0
    for i in range(1,n+1) :
        for j in range(1,n+1) :
            s+=i*j**2
    return s

def deuxfor2(n) :
    s=0
    t=0
    for i in range(1,n+1) :
        s+=i
    for j in range(1,n+1) :
        t+=j**2
    return s*t
```

deuxfor1(n) et deuxfor2(n) renvoient : $\sum_{i=1}^{n}\sum_{j=1}^{n} i.j^2$.

Pour la fonction deuxfor1, on effectue :

- 1 affectation,
- une boucle for de longueur n avec, à chaque itération, une boucle for de longueur n avec 1 incrémentation, 1 produit et 1 carré, soit $3n$ opérations à chaque itérations,
- 1 renvoi.

Le nombre total d'opérations est donc : $1+n.3n+1 = O(n^2)$.
La complexité est donc $O(n^2)$, c'est-à-dire quadratique.

Pour la fonction deuxfor2, on effectue :

- 2 affectations,
- une boucle for de longueur n avec, à chaque itération, 1 incrémentation,
- une boucle for de longueur n avec, à chaque itération, 1 incrémentation et 1 carré,
- 1 produit et 1 renvoi.

Le nombre total d'opérations est donc : $2+n+2n+2 = O(n)$.
La complexité est donc $O(n)$, c'est-à-dire linéaire.

Remarque : La complexité de deux boucles for imbriquées est plus grande que celle de deux boucles for qui se suivent, il ne faut donc imbriquer des boucles for que lorsque cela est indispensable.

➪ **Exemple 8 :**

```
def somme2(n) :
    s=0
    for i in range(n+1) :
        for j in range(i+1) :
            s+=i**j
    return s
```

somme2(n) renvoie $\sum_{i=0}^{n}\sum_{j=0}^{i} i^j$.

On effectue :

- 1 affectation,
- une boucle for avec $i \in [\![0,n]\!]$ avec, à chaque itération, 1 boucle for de longueur i avec 1 incrémentation et 1 puissance,
- 1 renvoi.

Le nombre total d'opérations est donc :

$$1+\sum_{i=0}^{n}(2i)+1 = 2+2\frac{n(n+1)}{2} = 2+n(n+1) = O(n^2).$$

La complexité est donc $O(n^2)$, c'est-à-dire quadratique.

2.4.2 La recherche par dichotomie

Dans toute la suite, on considère une liste l d'**entiers**, triée par **ordre croissant**. On va étudier la fonction suivante, qui renvoie True si l'entier x est dans la liste l et False sinon.

```
def recherchedicho(l,x) :
    '''Recherche, par la méthode de dichotomie, si un élément x est dans une liste l
    triée par ordre croissant. '''
    l.sort()
    i=0
    j=len(l)
    while i<j-1 :
        k=(i+j)//2
        if x<l[k] :
            j=k
        else :
            i=k
    if x==l[i] :
        return True
    else :
        return False
```

Cette fonction utilise le principe de la dichotomie : à chaque étape, on ne considère plus que la moitié de la liste.

• Tableau d'avancement pour recherchedicho([0,2,3,8,12],3) :

i	0	…	…	…	…	…	…
j	5	…	…	…	…	…	…
k	2	…	…	…	…	…	…

A la fin de la boucle while, on a : i=…… donc l[i]=…… ainsi recherchedicho([0,2,3,8,12],3) renvoie : ……

• Tableau d'avancement pour recherchedicho([0,2,3,8,12],9) :

i	0	…	…	…	…	…	…
j	5	…	…	…	…	…	…
k	2	…	…	…	…	…	…

A la fin de la boucle while, on a : i=..... donc l[i]=..... ainsi recherchedicho([0,2,3,8,12],9) renvoie : .....

Dans les calculs qui suivent, on ne considèrera pas la ligne 3, donc on n'évaluera pas le coût du tri. On posera n=len(l).

Calcul de la complexité :

On effectue :

- 2 affectations,
- une boucle while de longueur au plus N (que l'on déterminera dans la suite) avec, à chaque itération, 1 comparaison, 1 affectation, 1 addition, 1 division, 1 comparaison avec dans tous les cas 1 affectation,
- 1 comparaison avec, dans tous les cas, 1 renvoi.

Le nombre total d'opérations est donc : $2+6N+2=0(N)$.

De plus, à chaque itération la longueur de la liste est divisée par 2, donc, après N passages dans la boucle, la longueur de la liste est majorée par $\frac{n}{2^N}$. Or la boucle s'arrête lorsque la longueur de la liste est inférieure ou égale à 1. Or :

$$\frac{n}{2^N} \leq 1 \Leftrightarrow n \leq 2^N \Leftrightarrow \ln(n) \leq (N)\ln(2) \Leftrightarrow \frac{\ln(n)}{\ln(2)} \leq N.$$

Ainsi : $N = O(\ln(n))$.

La complexité est donc $O(\ln(n))$ c'est-à-dire logarithmique. Cette complexité est meilleure que la complexité linéaire de la fonction naturelle.

2.5 Complexité des fonctions récursives

Certaines fonction peuvent être écrites de façon récursives ou itératives. On va étudier les deux écritures.

➪ **Exemple 9 :** Les deux fonctions suivant calculent la factorielle d'un entier positif.

```
def fact1(n) :
    if n==0 :
        return 1
    else :
        return n*fact1(n-1)

def fact2(n) :
    f=1
    for j in range(2,n+1) :
        f*=j
    return f
```

La fonction fact1 est la version récursive de la factorielle.

Pour fact1(n), on effectue :

- 1 comparaison,
- dans le pire cas, une multiplication et un appel à fact1(n-1).

Donc, si on note $C(n)$ le nombre d'opérations pour fact1(n), on a :

$$C(n) = 2 + C(n-1).$$

On reconnaît alors une suite arithmétque de raison 2. Comme $C(0) = 1$, on a :

$$C(n) = 2n+1$$

Ainsi la complexité de la fonction fact1 est :

$$O(n).$$

La fonction fact2 est la version itérative de la factorielle.

La complexité temporelle est celle d'une boucle for, donc la complexité de la fonction fact2 est :

$$O(n).$$

Les deux fonctions sont donc de complexité linéaire.

➪ **Exemple 10 :** On souhaite étudier la suite de Fibonacci qui est définie par :

$$\begin{cases} u_0 = 1 \\ u_1 = 1 \\ \forall n \in \mathbb{N}, u_{n+2} = u_{n+1} + u_n. \end{cases}$$

On considère les fonctions suivantes qui renvoient u_n :

```
def fib1(n) :
    if n==0 :
        return 1
    elif n==1 :
        return 1
    else :
        return fib1(n-1)+fib1(n-2)

def fib2(n) :
    p=1
    q=1
    for j in range(n) :
        (p,q)=(q,p+q)
    return p
```

La fonction fib1 est récursive et la fonction fib1 est itérative.

Pour fib1(n), on effectue :

- 1 comparaison,
- dans le pire cas, une addition , un appel à fib1(n-1) et un appel à fib1(n-2).

Donc, si on note $C(n)$ le nombre d'opérations pour fib1(n), on a :

$$C(n) = 2 + C(n-1) + C(n-2).$$

On peut montrer par récurrence double que :

$$\forall n \in \mathbb{N}, C(n) \leq 4.2^n$$

On obtient donc une complexité exponentielle, c'est-à-dire une fonction qui, en pratique, ne permettra pas d'obtenir de résultat.

Pour fib2(n), la complexité est celle d'une boucle for, donc la complexité de la fonction fib2 est $O(n)$, c'est-à-dire linéaire.