Cours I

Étude d'algorithme

Lycée Louis Thuillier - Informatique - PCSIB -

Ce cours est largement inspiré du livre Toute l'informatique en CPGE scientifique de E. Cochard, S. Rainero, M. RoussangeE. Sebert-Cuvillier
Merci également à Paul Stienne pour m'avoir aidé sur le côté "math" !

Table des matières

On a écrit un programme Python. On peut se poser alors trois questions
1. mon programme se termine-t-il ?
2. mon programme donne-t-il le résultat recherché ?
3. combien de temps mon programme va-t-il prendre pour donner le résultat ?

Objectif du cours : mettre en place des outils et des méthodes pour répondre de façon précise à ces 3 questions.

1 Preuve de terminaison ou d'arrêt d'un algorithme

Il nous est à tous arrivé d'exécuter un programme qui tournera sans jamais s'arrêter. Cela arrive dans deux cas classique :
▷ l'utilisation d'une boucle *while*
▷ l'utilisation d'une fonction récursive
Il est donc nécessaire de s'assurer **à l'avance** que le programme se terminera toujours, et ce quelque soit les paramètres/données qu'on utilisera en entrée.

Exemple 1 :

```
S=0
i=0
while i<100 :
    S=S+1
```

Ici la condition de fin/sortie de boucle n'est jamais vérifiée : le programme ne se terminera jamais.

1.1 Terminaison d'une boucle while

Propriété. Variant de boucle
Une boucle *while* se termine si on peut construire une suite à valeur entière positive strictement décroissante qui décroit à chaque itération de la boucle.
C'est ce qu'on appelle **un variant de boucle**.

L'intérêt étant qu'une telle suite ne peut prendre qu'un nombre fini de valeur : par conséquent la boucle se termine forcément.

Exemple 2 :

```
S=0
i=0
while i<100 :
    S=S+i
    i=i+1
```

Le "compteur" i est modifié à chaque tour de boucle : comme il augmenter on "pressent" que la boucle se termine un jour. Reste à le prouver ...

Variant de boucle
On construit pour cela le variant de boucle : un nombre qui décroit strictement à chaque itération. On choisit ici : $u_i = 100 - i$.
Comme i augmente à chaque itération ***et*** *qu'il est nécessairement plus petit que 100 sinon la boucle s'arrête le nombre u_i décroit strictement et est toujours positif.*

La boucle se termine forcément.

Exercices d'application

Application 1 : Démontrer que la fonction division euclidienne suivante se termine toujours.

```
def division_euclidienne(a,b):
    r=a
    q=0
    while r>b :
        r=r-b
        q=q+1
    return(q,r)
```

Application 2 : On propose l'algorithme suivant pour la fonction PGCD qui renvoie le plus grand diviseur commun de deux nombres A et B :

- a=A
- b=B
- ***Tant que*** $a \neq b$:
  - *si* a>b ***alors*** a=a-b
  - *sinon* b=b-a

Retourner a

1. *Écrire le code Python correspondant*
2. *Donner une preuve formelle de sa terminaison.*

Application 3 :

1. *Que calcule cette fonction ?*
2. *Donner une preuve formelle de terminaison de cette fonction.*

```
def fonction(n):
    '''
    l'argument n est un entier naturel
    '''
    r=2
    i=0
    while i<n :
        r=r*r
        i=i+1
    return r
```

2 Preuve de correction

Pour justifier qu'une boucle produit l'effet désiré, on construit **un invariant de boucle**, c'est-à-dire une propriété qui est vraie avant la boucle et qui, si elle est vraie avant une itération de la boucle, elle reste vraie après "un tour de boucle".
Ainsi, comme l'invariant étant vrai avant la boucle il est vraie après. On construira alors un invariant de boucle qui permet de montrer que le programme fait ce qu'il faut.

Astuce pratique : on construit l'invariant en faisant le lien entre
▷ ce que doit faire le programme
▷ la variable que renvoie le programme

Exemple 3 : On reprend notre fonction de division euclidienne. Montrer que $a = bq + r$ est un invariant de la boucle.

```
def division_euclidienne(a,b):
    r=a
    q=0
    while r>b :
        r=r-b
        q=q+1
    return(q,r)
```

On va chercher à montrer que $a = bq + r$ est vrai tout au long de la boucle.

▷ *avant la boucle : $q = 0$ et $r = a$ donc $a = bq + r$: OK !!*

▷ *on suppose que c'est vrai à la fin d'une itération de la boucle $a = bq + r$*
  ▷ *on appelle r' et q' les nouvelles valeur que vont prendre r et q à la fin de la boucle*

$$q' = q + 1 \quad et \quad r' = r - b$$

  ▷ *Si on calcule $bq' + r' = b(q + 1) + r - b = bq + r = a$: la propriété reste vraie*

▷ *La propriété $a = bq + r$ est bien un invariant de la boucle est est vraie à la fin.*

Utilisation de la condition d'arrêt :
$0 \leq b < r$ *on a donc bien écrit la division euclidienne de a par b avec q le quotient et r le reste*

Nous allons étudier en fin de chapitre deux exemples.

3 Temps de calcul et notion de complexité

La méthode d'intégration par des rectangles est l'occasion de se pencher sur la notion de complexité d'un algorithme et ses conséquences pratiques. La question qu'on se pose est :

Est-ce que mon algorithme se termine-t-il en un temps raisonnable ?

3.1 Notion de complexité

Lorsqu'on exécute un programme, on utilise les ressources de l'ordinateur dont les deux principales sont :
- ▷ le temps de calcul pour exécuter les opérations
- ▷ la place de mémoire nécessaire pour stocker les données **ET** le programmes en cours d'exécution.

L'analyse de la complexité d'un programme consiste à mesurer ou à estimer ces deux grandeurs pour comparer différents algorithmes qui fonctionnent, afin de choisir le mieux adapté ou le plus efficace.
L'étude de la complexité en mémoire consiste en l'évaluation de l'espace mémoire nécessaire pour effectuer le calcul. Ce point important pour l'étude d'un programme ne sera pas étudié dans ce cours.

▶ Complexité en temps

Un ordinateur prend autant de temps à faire une multiplication, une addition, un test d'égalité, ajouter un élément à une liste, ... C'est ce qu'on appelle une opération élémentaire. Étudier la complexité en temps d'un programme revient alors à évaluer le nombre d'opérations qu'effectue le programme. Le temps de calcul est alors :

$$\text{nombre d'opérations} \times \text{durée d'une opération}$$

▶ Taille des données

De nombreux systèmes informatiques manipulent des données de très grandes tailles, constituées de millions, voir milliards, de nombres (météorologie, gestion du trafic aérien, statistiques de la population, traitement d'images, serveurs, réseaux sociaux, ...). On note N la taille des données manipulées, par exemple le nombre d'éléments dans une liste ou un tableau. On va chercher à étudier le lien entre le nombre d'opérations et la taille N des données.

> ***Remarque :*** *En réalité, il faudrait également prendre en compte le matériel (vitesse du processeur, taille et vitesse d'accès à la mémoire, temps de transfert disque...), le logiciel (langage de programmation, compilateur/interpréteur...), la charge de la machine (nombre de processus qui s'exécutent), la charge du réseau (accès aux données, écriture des résultats...)...*

3.2 Opérations élémentaires

On appelle opération élémentaires toutes les opérations consistant en :
- ▷ une comparaison, un test, une opération logique
- ▷ une opération arithmétique
- ▷ l'affectation d'un variable
- ▷ l'accès à un élément d'une structure de données
- ▷ l'impression, le retour d'une donnée

La recherche de la complexité d'un algorithme consiste à évaluer le nombre d'opérations élémentaires à effectuer à partir d'une donnée en entrée de taille N. De plus, on se placera toujours dans le pire des cas, c'est-à-dire le cas où l'algorithme doit effectuer le maximum d'opérations. De plus on s'intéressera à la façon dont croît cette complexité lorsque le nombre N tend vers l'infini.
Pour illustrer ce principe, on deux programmes cherchant :
- ▷ un élément dans une liste
- ▷ un élément dans un tableau

▶ Recherche d'éléments dans une liste

> ***Exemple 4 :*** *Proposer une fonction "brute de force", qui prend en entrée une liste L (de taille n) et une élément x et qui renvoie la liste des positions de x dans la liste L.*
>
> .....
> .....
> .....
> .....
> .....
> .....
> .....
> .....
> .....

Nombre d'opérations

La fonction va tester tout les éléments de la liste L. Donc, pour chacun des n éléments de la liste L, on réalise :

- ▷ une affectation à chaque nouveau pas de la boucle
- ▷ un test d'égalité $L[i] == x$
- ▷ on ajoute un élément à L

Le programme effectue donc, au pire, $3n$ opérations : la complexité est équivalente à $3n$. On parle de complexité linéaire.

▶ Recherche d'éléments dans un tableau

> ***Exemple 5 :*** *Proposer une fonction qui prend en entrée un tableau carré Tab (de taille n) et une élément x et qui renvoie la liste des positions $[i, j]$ de x dans le tableau Tab.*
>
> .....
> .....
> .....
> .....
> .....
> .....
> .....
> .....
> .....

Nombre d'opérations

Raisonnons sur chaque boucle :

- ▷ on affecte une valeur à k
- ▷ on affecte une valeur à l
- ▷ on fait un test d'égalité
- ▷ on ajoute un élément à L

Chaque opération s'effectue **pour chaque valeur de** k et **pour chaque valeur de** l soit $n \times n$ fois. La complexité de l'algorithme est $4n^2$. On parle de complexité quadratique.

Attention ! dans le plupart des cas, on s'intéresse au comportement asymptotique, *i.e.* quand n devient très grand !

> *Exemple 6 : Nombre d'opération* $= 3n + 2n^2 \to 2n^2$ *: complexité quadratique.*

3.3 Les complexités "classiques"

Linéaire $\sim n$

Propriété. Complexité linéaire
La complexité d'un algorithme est dite linéaire si sa complexité en temps, *i.e.* le nombre d'opération élémentaire, croît de façon linéaire avec l'augmentation de la taille n des données en entrée.

Une complexité linéaire est typique :
- ▷ d'une boucle $for/while$
- ▷ de plusieurs boucles $for/while$ les unes à la suite des autres

> *Exemple 7 : Si pour votre ordinateur, faire tourner le programme avec une liste de taille* $n = 5$ *prends* $10ns$ *alors il prendra :*
> - ▷ $100ns$ *pour une liste de taille* 10
> - ▷ $100\mu s$ *pour une liste de taille* 10^3
> - ▷ $10ms$ *pour une liste de taille* 10^6

Linéaire $\sim log(n)$

Propriété. Complexité logarithmique
La complexité d'un algorithme est dite logarithmique si sa complexité en temps, *i.e.* le nombre d'opération élémentaire, croît de façon logarithmique avec l'augmentation de la taille n des données en entrée.

Une complexité linéaire est typique des programmes utilisant des méthodes dichotomiques (*voir TD associé).*

> *Exemple 8 : Si pour votre ordinateur, faire tourner le programme avec une liste de taille* $n = 5$ *prends* $10ns$ *alors il prendra :*
> - ▷ $100ns$ *pour une liste de taille* 10
> - ▷ $30ns$ *pour une liste de taille* 10^3
> - ▷ $60ns$ *pour une liste de taille* 10^6

Linéaire $\sim n^2$

Propriété. Complexité quadratique (polynomiale)
La complexité d'un algorithme est dite quadratique si sa complexité en temps, *i.e.* le nombre d'opération élémentaire, croît de façon quadratique (polynomial) avec l'augmentation de la taille n des données en entrée.

Une complexité quadratique est typique
- ▷ de deux boucles for imbriquées
- ▷ de programme parcourant des tableaux à deux dimensions

> *Exemple 9 : Si pour votre ordinateur, faire tourner le programme avec une liste de taille* $n = 5$ *prends* $10ns$ *alors il prendra :*
> - ▷ $1\mu s$ *pour une liste de taille* 10
> - ▷ $10ms$ *pour une liste de taille* 10^3
> - ▷ 2.8 *heures pour une liste de taille* 10^6

4 Calcul de puissance

On va mettre en pratique nos connaissances sur une exemple "classique" : le calcul de x^n. L'objectif est d'analyser et de comparer deux programmes :

```python
def exponentielle_1(x,n):
    result=1
    for k in range(n):
        result=result*x
    return result
```

```python
def exponentielle_2(x,n):
    result =1
    N=n
    X=x
    while N>0 :
        if N%2==0:
            N=N//2
            X=X*X
        else :
            N=N-1
            result=result*X
    return result
```