Méthode d'écriture et d'analyse des algorithmes

Table des matières

1) Trace d'un programme

1.1) Définition

Il existe différentes manières de réaliser une trace de programme et/ou d'algorithmes. Une trace :
- permet de suivre pas à pas l'algorithme ;
- permet de détecter des erreurs ;
- permet de contrôler que l'algorithme fait bien ce que l'on avant prévu ;
- permet de déterminer ce que fait un algorithme.

Dans la mesure du possible, on peut organiser une trace d'exécution d'un algorithme en constituant un tableau avec toutes les variables de l'algorithme. Il faut numéroter toutes les lignes de votre algorithme. En colonne, il faut indiquer le nom des variables et en ligne les numéros de ligne.

1.2) Exemple

```
r=0
while r*r<= n :
        r=r+1
r=r-1
```

Il faut commencer par numéroter toutes les lignes de l'algorithme.

```
1  r=0
2  while r*r<= n :
3          r=r+1
4   r=r-1
```

Voici une trace de l'algorithme avec $n = 4$. quelle est la valeur de la variable r ?

N° ligne	n	r	commentaires
1	4	0	Initialisation
2	4	0	$0 \times 0 \leq 4$, on entre dans la ligne 3
3	4	1	On ajoute 1 à $r = 0$
2	4	1	$1 \times 1 \leq 4$, on reste dans la boucle
3	4	1	On ajoute 1 à $r = 1$
2	4	2	$2 \times 2 = 4$, on reste pour la dernière fois dans la boucle
3	4	3	On ajoute 1 à $r = 2$
2	4	3	$3 \times 3 = 9 \geq= 4$, on sort de la boucle
4	4	2	on retire 1 à $r = 3$

La variable r a pour valeur 2

1.3) Exercice

La fonction factorielle réalise le calcul suivant : $n! = n \times n-1 \times .... \times 2 \times 1$ Par exemple : $5! = 5 \times 4 \times 3 \times 2 \times 1$

Réaliser la trace du programme suivant pour $n = 4$:

```
def fact(n):
  x=1
  for i in range(2,n+1):
    x=i*x
  return x
```

2) Variants de boucles

2.1) Définition

On appelle variant d'une boucle une expression dont la valeur varie à chacune des itérations de la boucle.

2.2) Exemple

Calcul de la plus petite puissance de deux supérieure ou égale à un entier n.

```
def puissance(n:int)-> int:
p=1
while p<n:
  p=2p
return p
```

Dans l'algorithme ci-dessus, p est un variant de la boucle `while` car sa valeur (non nulle) est multipliée par 2 à chacune des itérations.

Un variant de boucle bien choisi permet de prouver qu'une boucle `while` se termine.

Dans l'algorithme précédent, le variant de boucle non nul p est multiplié par 2 à chaque itération, il finit donc par devenir supérieur ou égal à n et la boucle `while` se termine.

3) Invariant de boucle

3.1) Définition

On appelle invariant d'une boucle une propriété qui, si elle est vraie avant l'éxecution d'une itération le demeure pendant et après l'exécution de l'itération.

3.2) Exemple

L'exemple suivant calcule x à la puissance n.

```
def puissance(x:int,n:int)-> int:
r=1
for i in range(n):
  r=r*x
return r
```

Avant d'éxecuter le tour de boucle i, on a $r = x^i$. On vérifie en particulier que c'est bien vrai pour $i = 0$, car $r = r \times x = x^0 = 1$. On peut donc écrire que l'invariant est x à la puissance i.

3.3) Exercice

On considère la fonction suivante qui permet de calculer le quotient et le reste de la division euclidienne d'un entier positif par un entier strictement positif :

```
def division_euclidienne(a:int,b:int)-> int:
q=0
r=a
while r>=b:
  q=q+1 #on peut aussi écrire q+=1
  r=r-b #on peut aussi écrire r-=b
return q,r
```

1. Écrire la trace de l'algorithme pour l'entrée (a= 17, b= 5).
2. Montrer que la boucle `while` se termine en utilisant un variant de boucle
3. Montrer que la propriété $a = q \times b + r$ est un invariant de la boucle `while`, en déduire que l'algorithme produit le résultat attendu.

4) Une première approche de la complexité

4.1) Définition

La notion de complexité d'un algorithme va rendre compte de l'efficacité de cet algorithme. Pour un même problème, par exemple trié un tableau, il existe plusieurs algorithmes, certains algorithmes sont plus efficaces que d'autres (par exemple un algorithme A mettra moins de temps qu'un algorithme B pour résoudre exactement le même problème, sur la même machine).

Il existe 2 types de complexité : une complexité en temps et une complexité en mémoire. Nous nous intéresserons ici uniquement à la complexité en temps. La complexité en temps est directement liée au nombre d'opérations élémentaires qui doivent être exécutées afin de résoudre un problème donné. L'évaluation de ce nombre d'opérations élémentaires n'est pas chose facile, on rencontre souvent des cas litigieux.

4.2) Calcul de la complexité

a) Règles générales

Pour calculer la complexité, nous allons devoir examiner chaque ligne de code et y attribuer un coût en temps.

Le coût ainsi obtenu n'aura pas d'unité, il s'agit d'un nombre d'opérations dont chacune aurait le même temps d'execution :1.

Les opérations qui vont devoir être comptabilisées sont :

- Les affectations comptent pour 1 unité. Exemple : `a=2`.
- Les comparaisons comptent pour 1 unité. Exemple : `2<3`.
- L'accés aux mémoires compte pour une 1 unité.
- Chaque opération élémentaire compte pour 1 unité Exemple : .

Exemple 1 : déterminons le coût de la ligne suivante : . Le coût que l'on notera $T(n) = 1$ (affectation) $+1$ (accès à la mémoire)$+1$(addition)$=3$

b) Algorithme avec une boucle for

```
def sommeEntiers(n:int)-> int:
  somme = 0
  for i in range(n+1):
    somme += i
  return somme
```

Exercice 1 : déterminer la complexité $T(n)$ de cet algorithme.

La complexité de cet algorithme est dite linéaire. Ce sera le cas de tous les algorithmes avec $T(n) = an+b$ où a et b sont des réels.

c) Algorithmes avec deux boucles for imbriquées

La complexité de cet algorithme est dite quadratique. Ce sera le cas de tous les algorithmes avec $T(n) = an^2 + bn + c$ où a, b et c sont des réels.