- LANGAGES ET PROGRAMMATION -

OUTILS POUR L’ ANALYSE ALGORITHMIQUE

&

Table des matières

[I. TRACE D’UN PROGRAMME 1](#_Toc124836771)

[1. Définition et premier exemple 1](#_Toc124836772)

[2. Tracer l’exécution d’une structure de choix 2](#_Toc124836773)

[3. Tracer l’exécution d’une boucle while 3](#_Toc124836774)

[4. Tracer l’exécution d’une boucle for 4](#_Toc124836775)

[II. VARIANT DE BOUCLE 4](#_Toc124836776)

[1. Définition 4](#_Toc124836777)

[2. Exemple 4](#_Toc124836778)

[III. INVARIANT DE BOUCLE 4](#_Toc124836779)

[1. Définition 4](#_Toc124836780)

[2. Exemple 5](#_Toc124836781)

[3. Exercice 5](#_Toc124836782)

[IV. UNE PREMIÈRE APPROCHE DE LA COMPLEXITÉ 6](#_Toc124836783)

[1. Définition 6](#_Toc124836784)

[2. Calcul de la complexité 6](#_Toc124836785)

[a) Règles générales 6](#_Toc124836786)

[b) Algorithme avec une boucle for 6](#_Toc124836787)

[c) Algorithmes avec deux boucles for imbriquées 6](#_Toc124836788)

# I. TRACE D’UN PROGRAMME

## 1. Définition et premier exemple

Il existe différentes manières de réaliser une trace de programme et/ou d’algorithmes. Une trace permet :

- de suivre pas à pas l’algorithme ;

- de détecter des erreurs ;

- de contrôler que l’algorithme fait bien ce que l’on avant prévu.

Dans la mesure du possible, on peut organiser une trace d’exécution d’un algorithme en constituant un tableau. Au préalable, toutes les lignes de l’algorithme doivent être numérotées.

Dans la 1ère colonne du tableau on indique le numéro de la ligne de code exécutée, dans les colonnes suivantes les noms des diverses variables puis dans la dernière colonne des commentaires.

Ce premier exemple montre comment faire la trace de l’exécution de la séquence d’instructions suivantes :

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | b = 5  a = b \* 3  c = a - b  b = c  b = b + 20  c = (b + a) \* 2  print(c) |

Le programme comporte 3 variables a, b et c.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| N° ligne | a | b | c | Commentaires |
| 1 |  | 5 |  | On déclare la variable b et on lui affecte la valeur de 5 : b 🠔 5 |
| 2 | 15 | 5 |  | On calcule b\*3 = 15 et on affecte la valeur à a : a 🠔 15 |
| 3 | 15 | 5 | 10 | a-b = 15 - 5 = 10 c 🠔 10 |
| 4 | 15 | 10 | 10 | b 🠔 c = 10 |
| 5 | 15 | 30 | 10 | b + 20 = 10 + 20 = 30 b 🠔 30 |
| 6 | 15 | 30 | 90 | (b + a )\*2 = (30 + 15)\*2 = 90 c 🠔 90 |
| 7 | 15 | 30 | 90 | On affiche la valeur de c=90 |

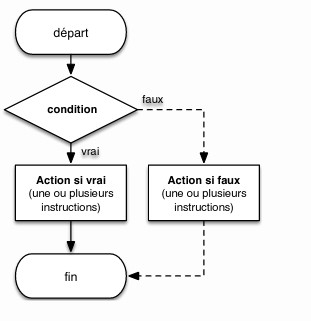
**Q1 :** Effectuer la trace du programme suivant.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | x = 5  a = 5  b = 10  tmp = a  a = b  b = tmp  y = a \* x + b  print(y) |

## 2. Tracer l’exécution d’une structure de choix

Pour faire la trace d’un programme, on doit parfois faire des suppositions. Dans l’exemple suivant, on demande à l’utilisateur de saisir un nombre. Comme la trace de l’exécution dépend de la valeur saisie, on doit admettre une valeur et exécuter le programme avec cette valeur.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | val = int(input("Rentrez une valeur entière"))  if val >= 10 and val <= 20 :      res = -val  else :      res = val \* 2  res = res + 20  print(res) |



De plus, ce programme contient une structure de choix dont la condition du choix dépend de la valeur saisie par l’utilisateur. Il y a donc deux chemins d’exécution possibles. Pour illustrer cela, la figure ci-contre montre un organigramme qui permet de bien visualiser les deux chemins que peut suivre le fil d’exécution dans une structure de choix.

La trace de l’exécution pour une valeur donnée ne suivra qu’un seul de ces deux chemins. Si l’on veut analyser les deux chemins, on doit faire la trace une première fois avec une valeur qui remplit la condition, puis une seconde fois avec une valeur qui ne la remplit pas.

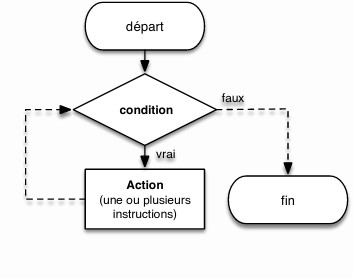
|  |  |  |  |
| --- | --- | --- | --- |
| N° ligne | val | res | Commentaires |
| 1 | 15 |  | val 🠔 15 |
| 2 | 15 |  | val >= 10 and val <= 20 ? oui, on exécute la ligne 3 |
| 3 | 15 | -15 | - val = - 15 res 🠔 -15 |
| 6 | 15 | 5 | res+20 = -15 + 20 = 5 res 🠔 5 |
| 7 | 15 | 5 | On affiche la valeur de res=5 |

**Q2 :** Effectuer la trace du programme avec un variable val qui ne satisfait pas la condition du if.

## 3. Tracer l’exécution d’une boucle while

Avec une boucle while (tant que), les instructions du corps de la boucle sont répétées tant que la condition de continuation est remplie ou, dit autrement, tant que l’expression booléenne est vérifiée.

|  |  |
| --- | --- |
| 1  2  3  4  5 | n = 4  r = 0  while r\*r <= n :  r = r+1  r = r-1 |



Comme le montre l’organigramme suivant, l’exécution d’une boucle while commence par le test de la condition de continuation.

Voici une trace de l’algorithme.

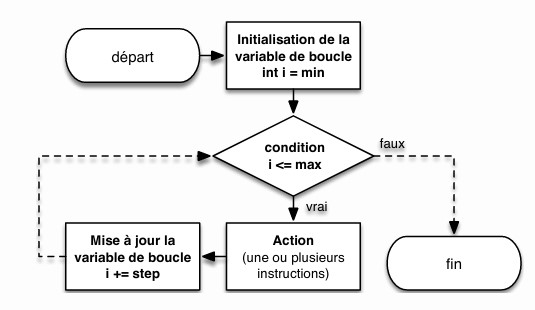
|  |  |  |  |
| --- | --- | --- | --- |
| N° ligne | n | r | Commentaires |
| 1 | 4 |  | n 🠔 4 |
| 2 | 4 | 0 | r 🠔 0 |
| 3 | 4 | 0 | r\*r<=n ? oui, on exécute la ligne 4 |
| 4 | 4 | 1 | r+1 = 0+1 = 1 r🠔1 |
| 3  4  3  4  3  5 | 4  4  4  4  4  4 | 1  2  2  3  3  2 | r\*r<=n ? oui, on exécute la ligne 4  r+1 = 1+1 = 2 r🠔2  r\*r<=n ? oui, on exécute la ligne 4  r+1 = 2+1 = 3 r🠔 3et on affecte le résultat à la variable r  r\*r<=n ? non, on exécute la ligne 5  r-1 = 3-1 = 2 r🠔 2 |

**Q3 :** Faire la trace d’exécution du programme suivant et expliquer ce que fait le programme. Proposer ensuite une manière plus simple d’écrire la condition de la boucle while, faire à nouveau la trace pour vérifier que l’on parvient au même résultat.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | a = 24  b = 6  q = 0  while ((a - b) > 0 or (b - a) == 0):  a = a - b  q = q + 1  r = a;  print(q)  print(r) |

## 4. Tracer l’exécution d’une boucle for

Voici l’organigramme d’une boucle for.



**Q4 :** La fonction factorielle réalise le calcul suivant : *n*! = *n* × *n*−1 × *....* × 2 × 1.

Par exemple : 5 ! = 5 × 4 × 3 × 2 × 1 = 120.

Réaliser la trace de la fonction suivante qui prend en argument la valeur 4 :

|  |  |
| --- | --- |
| 1  2  3  4  5 | **def** fact(n):      x=1      for i in range(2,n+1):      x=i\*x      return x |

# II. VARIANT DE BOUCLE

## 

## 1. Définition

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

## 

## 2. Exemple

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

|  |  |
| --- | --- |
| 1  2  3  4  5 | **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.

# III. INVARIANT DE BOUCLE

## 

## 1. Définition

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

## 

## 2. Exemple

La fonction suivante calcule 2 à la puissance n.

|  |  |
| --- | --- |
| 1  2  3  4  5 | **def** puissance(n: int)-> int:      r = 1      for i in range(1,n+1):          r = r \* 2      return r |

**On peut vérifier que r=2i est un invariant de boucle.**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| N° ligne | n | r | i | Commentaires |
| 1 | 3 |  |  |  |
| 2 | 3 | 1 |  |  |
| 3 | 3 | 1 | 1 | i<n+1 ? oui, on exécute la ligne 4 |
| 4 | 3 | 2 | 1 | 1ère exécution du corps de boucle |
| 3 | 3 | 2 | 2 | i<n+1 ? oui, on exécute la ligne 4 |
| 4 | 3 | 4 | 2 | 2ème exécution du corps de boucle |
| 3 | 3 | 4 | 3 | i<n+1 ? oui, on exécute la ligne 4 |
| 4 | 3 | 8 | 3 | 3ème exécution du corps de boucle |
| 3 | 3 | 8 | 4 | i<n+1 ? non, on exécute la ligne 5 |
| 5 | 3 | 8 | 4 | La fonction retourne la valeur 8 |

Avant de rentrer dans la boucle, on peut considérer que i vaut 0 et on a r=1 : on a bien r=2i = 20=1

A la 1ère exécution du corps de boucle :

On a n=3, r=2 et i=1 : on a bien r=2i

2ème exécution du corps de boucle :

On a n=3, r=4 et i=2 : on a bien r=2i

3ème et dernière exécution du corps de boucle :

On a n=3, r=8 et i=3 : on a bien r=2i

**On peut donc bien écrire que l’invariant est 2 à la puissance i.**

## 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 :

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | **def** division\_euclidienne(a:int,b:int)-> int:          q=0          r=a          while r>=b:              q=q+1 *#on peut aussi écrire q+=1 6*              r=r-b *#on peut aussi écrire r-=b*          return q,r |

**Q5 :**

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* ×*b*+*r* est un invariant de la boucle while ;

4. En déduire que l’algorithme produit le résultat attendu.

# IV. UNE PREMIÈRE APPROCHE DE LA COMPLEXITÉ

## 

## 1. Définition

La notion de complexité d’un algorithme va rendre compte de l’efficacité de cet algorithme. Pour un même problème, par exemple trier 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.

## 

## 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’exécution, à savoir 1.

Les opérations qui vont devoir être comptabilisées sont :

- Les affectations qui comptent pour 1 unité, par exemple : a = 2 ;

- Les comparaisons qui comptent pour 1 unité, par exemple : 2 < 3 ;

- Les accès aux mémoires qui comptent pour une 1 unité ;

- Les opérations élémentaires qui comptent pour 1 unité, par exemple : 2 + 3.

*Exemple :*

Déterminons le coût de la ligne suivante que l’on notera *T*(*n*) :

a = a + 1

*T*(*n*) = 1 (accès à la mémoire) +1 (addition) + 1 (affectation) =3

### 

### b) Algorithme avec une boucle for

|  |  |
| --- | --- |
| 1  2  3  4  5 | **def** sommeEntiers(n:int)-> int:          somme = 0          for i in range(n+1):              somme += i          return somme |

**Q6 :** 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*) = *a × n* + *b* où *a* et *b* sont des réels.

### c) Algorithmes avec deux boucles for imbriquées

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | **def** tables\_multiplication():      for i in range(n):          for j in range(n):              print(i\*j)      return None |

**Q7 :** Déterminer la complexité *T*(*n*) de cet algorithme.

La complexité des algorithmes contenant deux boucles for est dite quadratique. Ce sera le cas de tous les algorithmes avec T(n) = a × n² + b × n + c où a, b et c sont des réels.