Terminaison-correction-complexite

December 13, 2022

$$x^2 + 98754x + 9822456822556332$$

#

NUMERIQUE ET SCIENCES INFORMATIQUES 1ère NSI

0.1 Séquence N° 6 : Terminaison - Correction - Complexité

[ ]:
```
# This is an example snippet!
# To create your own, add a new snippet block to the
# snippets.json file in your jupyter nbextensions directory:
# /nbextensions/snippets/snippets.json
import this
```

Les objectifs de cette séquence sont :

- Monter la terminaison de quelques algorithmes ;
- Monter qu'un algorithme produit bien le résultat escompté ;
- Évaluer la complexité d'un algorithme

0.1.1 1- Introduction

Lorsque l'on écrit un algorithme, on souhaite plusieurs choses : 1. Qu'il se termine (pas de boucle infinie) ; 2. Qu'il produise le résultat attendu (résultat correct) ; 3. Qu'il soit le plus rapide possible et nécessite le moins de mémoire possible.

Une erreur peut avoir un impact :

- Sur des vies humaines (conduite des lignes de métro automatiques, secteur militaire …) ;
- économique (lancement d'engin spacial …).

Un algorithme est un programme de calcul qui s'exécute en un nombre fini d'étapes. Si dans le cadre des **boucles bornées** (boucle **for** en python) on a la certitude que le calcul se finira effectivement, lors de l'exécution d'une boucle **non bornée** (boucle **while** en python), nous n'avons pas cette certitude.

De manière plus détaillée, nous allons : - Montrer qu'une boucle se **termine bien**. On appelle ce problème la **terminaison**. - Montrer que si la boucle s'arrête, elle **calcule bien** ce qu'elle est supposée calculer. On appelle ce problème la **correction partielle**.

La **correction totale** est la **terminaison** et la **correction partielle**.

0.1.2 2- Terminaison d'un algorithme : variant de boucle

Prenons un exemple : Le programme ci-dessous calcule la valeur de $2^{exposant}$ au moyen d'une boucle (la méthode utilsée ne sera pas remise en cause). La valeur de l'exposant est saisi au clavier par l'utilisateur.

[ ]:
```
def puissance_2(exp):
    """
    La fonction calcule la valeur de 2 élevé
    à la puissance de l'exposant fourni en paramètre
    et retourne le résultat
    Entrée : exp valeur entier (int)
    Sortie : valeur valeur entière (int)
    """
    valeur=1
    while exp >= 0:
        valeur = 2*valeur
        exp = exp-1
    return valeur
exposant=int(input('Donnez la valeur de l\'exposant :'))
resultat = puissance_2(exposant)
print(f'le résultat est : {resultat}')
```

Travail à faire

- Après avoir analyser le fonctionnement de la fonction, testez la avec différentes valeurs de votre choix, mais aussi avec une valeur négative.

Ne me planter (PLANTEZ !) pas le serveur ! : → **Noyau, redémarrer et effacer les sorties**

- Proposer des solutions permettant d'éviter le problème (je rappelle que la méthode utilisée ne sera pas remise en cause).

Variant de boucle Les variants de boucle sont un outil qui va permettre de montrer la terminaison d'algorithmes.

Définition

Un variant de boucle est une quantité entière définie en fonction des variables $(x_1, \ldots, x_k)$ constituant l'état de la machine, et de n, le nombre de passages effectués dans la boucle qui :

1. est un entier strictement positif avant l'exécution de la boucle
2. décroît strictement à chaque itération
3. lorsqu'elle est inférieure à un certain nombre (en particulier lorsqu'elle arrête d'être strictement positive) rend la condition d'exécution de la boucle conditionnelle fausse

Remarque :

Dans le cas d'une boucle for, nous avons dit que nous avions la certitude de terminaison : on peut toujours construire un variant simple. Si la boucle est donnée par la structure :

`for i in range(n) : Pour i ← 0 à n-1, un variant simple est (n-1)-i`

Travail à faire

1. Dans le programme précédent, quel est le variant de boucle ?
2. Déterminer quel est le variant de boucle du script et de l'algorithme suivants :

[6]:
```
# script 1
def Mystere(x,n) : # x et n sont des entiers positifs
    res=1
    while n>0 :
        if n%2==1 :
            res=res*x
        n=n//2
        x=x*x
    print(res)
```

Algorithme 1 : multiplication de 2 nombres a et b en n'effectuant que des additions !

Entrées : 2 nombres a et b Sortie : le résultat z Variables : a, b, x, y Début a ← valeur saisie par l'utilisateur x← b y ← 0 Tant que x > 0 y ← y + a x ← x – 1 Fin tant que Fin

Variant :n=n//2

0.1.3 3- Correction d'un algorithme : invariant de boucle

[ ]:
```

```

Les invariants de boucle sont un outil qui va permettre de montrer la correction partielle d'algorithmes.

Définition (Invariant de boucle)

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

Trouver l'invariant de boucle est une chose peu aisée, mais il faut ensuite démontrer qu'il est correct (une fois de plus l'étude d'un cas particulier ne vaut pas démonstration). La démonstration doit se faire en 3 étapes.

En mathématiques, on appelle cela une démonstration par récurrence.

Travail à faire

1. Reprenons l'algorithme 1 ci-dessus et Faites le « tourner » sur un exemple (a = 13, b = 3) en complétant le tableau.
2. Compléter le texte à trous

Si l'invariant de boucle est la propriété P : y + a.x = a.b, les valeurs trouvées dans le tableau sont-elles correctes ? ….

Démonstration :

Initialisation : y = 0, x = b donc a.b = a.b la propriété P(n) est vraie.

Hérédité : Supposons que P(n) soit vraie et montrons que P(n + 1) est vraie.

Au prochain tour (n+1), on ajoutera a à y et on enlèvera 1 à x.

(y+a) + a.(x-1) = a.b

développer y+a+ax-a = a.b

simplifier y+ax = a.b

Conclusion : À l'initialisation, la propriété est vraie et on a montré que la propriété était conservée après une itération, en conclusion, en vertu du principe de raisonnement par récurrence, la propriété est vraie.

0.1.4 4- Complexité d'un algorithme : complexité temporelle

Nous avons maintenant un algorithme qui produit le résultat demandé et qui se termine . . . mais quand ? Avant que les données traitées soient périmées au moins !

On va maintenant s'intéresser à la durée d'exécution du programme ou plus précisément à la manière dont la durée d'exécution augmente lorsque la taille des données augmente.

Le temps exact d'exécution dépend de beaucoup de choses : - Du langage dans lequel l'algorithme est implémenté (compilé ou interprété) ; - Du matériel informatique (microprocesseur …) ; - Du nombre de processus qui s'exécutent en même temps ;

On va donc essayer de caractériser l'efficacité d'un algorithme indépendamment de son implémentation en comptant le nombre d'opérations élémentaires nécessaires à son exécution. On va donc éviter d'utiliser les fonctions built-in de python comme "len" ou "max" qui peuvent cacher un nombre important d'opérations.

Analysons la complexité de la fonction ci-dessous :

[7]:
```
#%%time

# -*- coding: utf-8 -*-
"""
Programme ne servant à rien
son but est d'évaluer
une complexité !
"""
def complexe1():
    # n : nombre de valeurs
    n = 10000
    sp = 0

    for i in range(n):
       sp = sp + i
    return sp

V=complexe1()
print('La valeur trouvée pour V = ',V)
```

```
La valeur trouvée pour V =  49995000
```

[ ]:
```
%timeit complexe1() # Temps d'exécution de la fonction complexe1
```

Prenons un second exemple :

[ ]:
```
#%%time
# -*- coding: utf-8 -*-
"""
Programme ne servant à rien
son but est d'évaluer
une complexité !
"""

def complexe2():
    # n : nombre de valeurs
    n = 10000
    sp = 0

    for i in range(n):
        for j in range(n-1):
            sp = sp + (i*j)
    return sp

V=complexe2()
print('La valeur trouvée pour V = ',V)
```

[ ]:
```
%timeit complexe2() # Temps d'exécution de la fonction complexe2
```

Si l'on considère une taille de donnée importante, seul le terme de plus grand ordre n pour le premier exemple et n2 pour le second est réellement significatif, aussi on se limite à dire que l'algorithme a un temps d'exécution qui croit de la même façon que n ou n2 et on parle de complexité en temps de O(n) et O(n2).

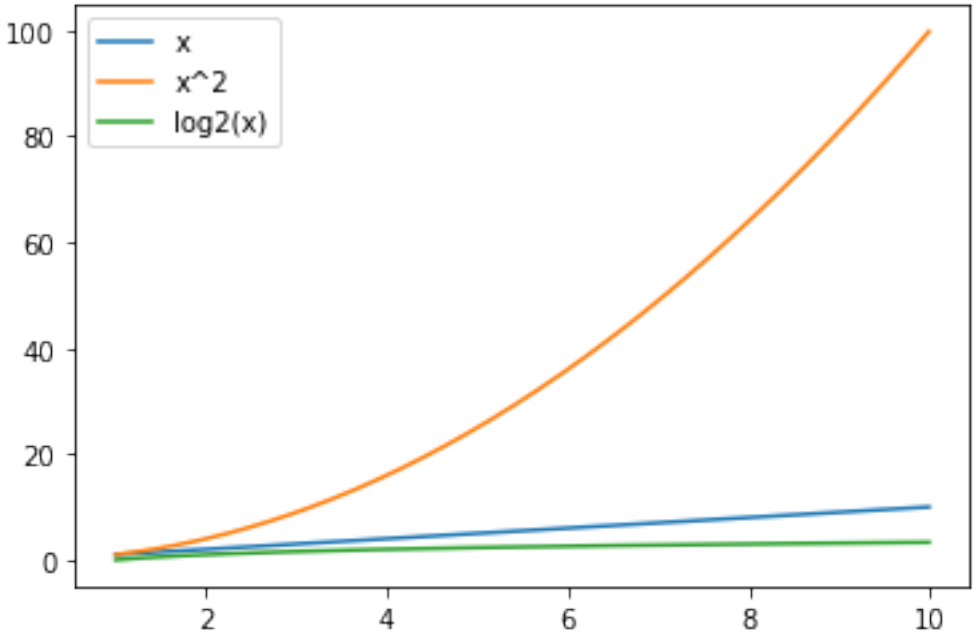
Complexité	Appellation	Ordre de grandeur en temps pour n=106	Observation - Exemple
O(1)	Constante	1 ns	Le temps ne dépend pas de la taille des données. Rechercher les 3 premiers d'une liste par exemple pour connaître le podium d'une compétition.
O(log2(n))	Logarithmique	10 ns	**Recherche dichotomique dans une liste triée**
O(n)	Linéaire	1 ms	**Recherche d'un extrémum (min, max ...)**
O(n2)	Quadratique	15 min	**Tri par sélection et insertion**
O(nk)	Polynomiale	30 ans (k=3)	Optimisation par programmation dynamique

Complexité	Appellation	Ordre de grandeur en temps pour n=106	Observation - Exemple
O(2n)	Exponentielle	> 10300000 milliards d'années	Inutilisable à l'exception de très petite taille de données

[9]:
```
from matplotlib.pylab import plot, arange, cos, sin, log2, plt

"""
Commentez l'affichage des graphes
en particulier pour Y4
"""
X = arange(1,10,0.01)
Y1, Y2 , Y3, Y4 = X, X**2, log2(X), 2**X,
plot(X,Y1,label="x")
plot(X,Y2,label="x^2")
plot(X,Y3,label="log2(x)")
#plot(X,Y4,label="2^x")
plt.legend()
```

[9]:
```
<matplotlib.legend.Legend at 0x7f0ac00715c0>
```



Travail à faire

Déterminer la complexité des scripts python suivants :

[ ]:
```
2n+4
```

[ ]:
```
# script 1
n = 100
s = 0
for i in range(n):
    s = s + 3
for j in range(n):
    s = s + 3
```

[ ]:
```
n²+3
```

[ ]:
```
# script 2
n = 20
s = 0
for i in range(n):
    for j in range(i,n):
        s = s + j**2
```

Script 2 plus complexe que script 1

[ ]:
```

```