Chapitre F.5 - Coût, terminaison et preuve d'un algorithme

I. Introduction

Peut-on évaluer l'efficacité de notre algorithme ? Parmi différents algorithmes permettant de résoudre le même problème, comment savoir lequel est le plus intéressant ? Comment s'assurer que notre algorithme fonctionne, dans tous les cas de figure possible ? Afin d'évaluer les algorithmes que nous avons écrit, nous nous basons sur différents critères : La ..... La ..... La .....

II. Complexité d'un algorithme

A. C'est quoi ?

La complexité (ou coût) d'un algorithme permet quantifier la performance d'un algorithme. Il existe deux types de complexités, la complexité en ..... et la complexité en .....
Cette année nous étudierons la complexité en temps. L'intérêt de connaître la complexité d'un algorithme est de pouvoir comparer son temps d'exécution par rapport à d'autres algorithme qui répondent au même problème.
La complexité nous aide donc à choisir l'algorithme le plus approprié à nos besoins. Pour des données volumineuses la différence entre les durées d'exécution de deux algorithmes ayant la même finalité peut être de l'ordre de plusieurs jours.
Pour se faire une idée de la complexité d'un algorithme, il serait possible de contrôler le temps que met l'algorithme à s'exécuter en fonction des valeurs que nous lui donnons en entrer. Cependant, les résultats que nous obtiendrions seraient en parti lié à la qualité de la machine qui l'exécute. C'est pourquoi, le calcul de la complexité algorithmique se fait à partir de la version papier de l'algorithme.

B. Calcul de la complexité

Pour calculer la complexité, nous allons devoir examiner chaque ligne de code et lui attribuer un coût. Le coût ainsi obtenu n'aura pas d'unité, il s'agit d'un nombre d'opérations élémentaires de notre algorithme.

> **Notation :**
> On note généralement la complexité d'un algorithme C(n)
> Dans la plupart des cas, n correspond à la taille de la donnée en entrée.

Exemple :
Déterminons le coût de l'algorithme suivant :

```
def ma_fonction(n):
    a = a * a
    b = a + a
    return b
```

Dans l'algorithme précédent, nous effectuons ..... opérations élémentaires.
Le résultat se note : C(n)= .....

Lorsque l'on souhaite obtenir la complexité d'un algorithme, ce qui nous intéresse, c'est d'obtenir un ordre de grandeur. On note donc la complexité : C(n)= O(1)

Différents types de complexité

Temps	Type complexité
$O(1)$	
$O(log(n))$	
$O(n)$	
$O(n^2)$	
$O(n^3)$	
$O(2^n)$	

III. Terminaison d'un algorithme

Pour démontrer qu'un algorithme est correct, dans un premier temps s'assurer que celui-ci s'arrête dans tous les cas.
Il existe des cas dans lequel nous pouvons être certain que notre algorithme s'arrête :

- Un algorithme ne contenant pas de boucle s'arrête toujours

  (Sauf s'il fait appel à des fonctions qui risquent de ne pas de terminer)

- Un algorithme contenant des boucles bornée s'arrête toujours

  (Sauf s'il fait appel à des fonctions qui risquent de ne pas de terminer)

Il faut alors démontrer la terminaison d'un algorithme uniquement dans le cas ou il contient une ou plusieurs boucles non bornées. Il faut alors démontrer que chacune des boucles non bornées qui constitue notre algorithme s'arrête. Il s'agit de vérifier que la condition de chaque boucle non bornée de l'algorithme deviendra fausse tôt ou tard afin que la boucle se termine.
L'une des techniques permettant de démontrer qu'une boucle s'arrête et de trouver un variant de boucle.

> Un variant de boucle est une suite d'élément :
> - Entière
> - Positive (ou négative)
> - Qui décroît strictement à chaque tour de boucle. (qui croit strictement à chaque tour de boucle)

Exemple :
On prend la fonction multiplication suivante :

```
def multiplication(a,b):
    m = 0
    while b > 0:
        m = m + a
        b = b - 1
    return m
```

Les valeurs successives de b constituent une suite d'entier positifs strictement décroissantes. b est donc un variant de boucle. La fonction s'arrête dans tous les cas.

IV. Preuve d'un algorithme

Pour démontrer qu'un algorithme est correct, il faut démontrer que la valeur renvoyée par notre algorithme correspond au bon résultat dans tous les cas.
Lorsque notre algorithme ne contient pas de boucle, il faut simplement vérifier que le résultat correspond bien à la spécification.

Exemple :

On cherche à démontrer la preuve le l'algorithme suivant qui prend un entier en paramètre et qui renvoie sa valeur au carré :

```
def carre(n):
    res = n * n
    return res
```

A la fin de l'exécution de la fonction, la variable res vaut n×n ce qui correspond bien au carré de n.
Dans le cas de l'utilisation d'une boucle, il faut trouver un invariant de boucle permettant de démontrer qu'après la boucle le résultat correspond bien au résultat attendu.

> Un invariant de boucle est une propriété qui est vraie pour tous les passages dans une boucle.
> L'invariant doit être vrai avant le début de la boucle (Initialisation), vrai pour chaque tour de la boucle (conservation) et toujours vrai après la boucle (Terminaison) .

Trouver un invariant de boucle permet de démontrer la preuve d'un algorithme.
On prend la fonction multiplication suivante :

```
def multiplication(a,b):
    m = 0
    while b > 0:
        m = m + a
        b = b - 1
    return m
```

INITIALISATION

Avant le premier tour de la boucle, m= 0=a×0

CONSERVATION

Après le $n^{ième}$ tour de la boucle, $m = a \times n$ Après le $n + 1^{ième}$ tour de la boucle, $m = a \times n + a = a \times (n + 1)$

TERMINAISON

Après le dernier tour de la boucle, c'est-à-dire après le $b^{ième}$ tour de la boucle $m = a \times b$.
Par conséquent, notre algorithme renvoie le bon résultat.