# Chapitre 10 : Analyse d'un algorithme

## I- Validité d'un algorithme

Deux conditions de validité :

- L'algorithme finit par donner une réponse, c'est l'étude de la terminaison.
- La réponse donnée est celle attendue (conforme aux spécifications), c'est l'étude de la correction.

### a) Terminaison

Le problème de la terminaison d'un algorithme itératif se pose typiquement dans le cas où le programme fait usage de boucles while. Pour prouver la terminaison, on recherche une grandeur, appelée variant de boucle, dont la valeur prise au cours de chaque itération finie par converger vers la condition d'arrêt de la boucle. Dans l'exemple ci-dessous, le variant de boucle est $m$ :

```
m=0
p=0
while m<a:
    m=m+1 #fin 1e passage m=1, fin 2e passage m=2,  fin du aeme passage m=a (condition d'arrêt)
    p=p+10
#pour m=a, la condition d'arrêt est atteinte
```

### b) Correction

Après la terminaison, pour prouver qu'un algorithme itératif est correct, on repère un invariant de boucle c'est-à-dire une propriété $\mathcal{P}$ qui est vérifiée avant l'entrée en boucle et aussi vérifiée après chaque passage dans cette dernière (ce que l'on démontre en supposant la propriété vraie à la $k^{ème}$ itération et en démontrant qu'elle est aussi vraie à l'itération suivante). A la dernière itération, $\mathcal{P}$ doit permettre de démontrer que le résultat obtenu est celui attendu.
Dans l'exemple ci-dessous, l'enjeu est d'obtenir $p = a * 10$ et l'invariant est la propriété $p = m * 10$

```
m=0
p=0#au 1e passage p =m*10 vérifiée
while m<a:
    m=m+1 # fin 1e passage m=1, fin 2e passage m=2,  fin du aeme passage m=a (condition d'arrêt)
    p=p+10#fin 1e passage p=1*10,fin 2e passage p=2*10, fin , fin du aème : p=a*10
```

- Initialisation : avant de rentrer en boucle on a bien $p = m * 10$ car $p = 0$ avec $m = 0$
- Continuité : Supposons $p = m * 10$ vrai, à l'itération suivante $p' = p + 10$ et $m' = m + 1$ soit $p' = 10 * m + 10 = 10 * (m + 1) = 10m'$
- Terminaison : Elle a été prouvée, donc en sortie de boucle $m = a$ et le résultat retourné est bien $p = 10a$

## II- Tester un programme

### a) Tests possibles

On peut tester un programme en :

- Utilisant la fonction *print* pour afficher des résultats intermédiaires,
- Utilisant des assertions après la fonction pour s'assurer du bon résultat

```
def f(x):
    return 3*x
assert f(0)==0#pas de message d'erreur
```

### b) Construire un jeu de tests

Construire un jeu de tests pour une fonction consiste à définir un ensemble de données qui vont être utilisées pour vérifier que le programme donne bien le bon résultat. On peut tester une fonction :

- Avec quelques valeurs simples typiques
- Avec des valeurs extrêmes, des cas limites, des cas interdits
- Avec un nombre important de données (choisies de manière aléatoire par exemple) afin d'apprécier le temps d'exécution du programme

Considérons le code suivant en rajoutant un jeu de test avec quelques cas typiques :

**Commenté [AM1]:** Lorsqu'on écrit un programme, il est impératif que ce dernier produise un résultat correct et se termine après un nombre fini d'étapes. Pour un algorithme itératif construit avec des boucles, le nombre de passages dans chaque boucle doit être fini.

**Commenté [AM2]:** Dans le cas d'un programme utilisant des boucles for, la terminaison est évidente. On parle de boucles bornées. On pourra citer le cas exotique suivant à éviter :
L=[1]
for i in L:
  L.append(1)

**Commenté [AM5]:** Afin de rendre les tests et le débogage (correction) plus simples, il est important de décomposer son programme en sous-programme qui peuvent être testés indépendamment les uns des autres (utilisation de plusieurs fonctions, spécifications, assertions, …)

**Commenté [AM3]:** Pour un algorithme récursif, le nombre d'appels récursifs doit être fini pour assurer sa terminaison : on doit atteindre la condition triviale qui met fin au programme.
Pour la correction d'un algorithme récursif, on suit une démarche de récurrence :
-On vérifie que l'algorithme est vrai pour le cas trivial.
-On le suppose juste au rang $k$ *et* on vérifie qu'il l'est au rang $k + 1$

**Commenté [AM6]:** on parle de bug permanent si erreur à chaque test, de bug intermittent si erreur lors de certains tests

**Commenté [AM4]:** C'est une condition nécessaire mais pas suffisante pour tous les cas. D'ailleurs tous les invariants ne sont pas forcément en accord avec la spécification

**Commenté [AM7]:** On dit que l'on partitionne le domaine d'entrée. Avec cette méthode, on recherche les erreurs. En revanche il n'est pas possible d'affirmer que le programme ne présente pas d'autres erreurs.

Avant test :

```
def permute(liste):
    copie=liste[:]
    copie[0],copie[-1]=copie[-1],copie[0]
    return copie
```

Jeu de tests :

```
assert permute([1,2,3,4])==[4,2,3,1]
assert permute([[1,2],[3,4]])==[[3,4],[1,2]]
assert permute([1])==[1]
assert permute([])==[]#la fonction renvoie une erreur
```

Après test :

```
def permute(liste):
    if len(liste)==0:
        return []
    copie=liste[:]#copie superficielle
    copie[0],copie[-1]=copie[-1],copie[0]#permutation
    return copie
```

## III- Complexités

### a) Présentation

Lorsqu'un algorithme se termine correctement, il doit encore, avant d'être programmé et exécuté, satisfaire à deux impératifs en termes de consommation de ressources :

- Utiliser un espace en mémoire acceptable, on parle de complexité en espace ;
- Produire la réponse attendue en un temps acceptable, on parle de complexité temporelle

Etudier la complexité temporelle consiste à apprécier l'évolution du temps d'exécution $T(n)$ d'un algorithme en fonction de la taille $n$ des données en entrée et donc en analysant le nombre d'opérations à exécuter. Quelques hypothèses à retenir :

- Les temps d'exécution des opérations de bases (affectation, opération arithmétique, comparaisons) sont identiques et constituent une unité (ou coût $C$) de référence ($C = 1$),
- Le temps d'exécution d'une suite d'instructions est la somme des temps d'exécution de chaque instruction.
- Pour une boucle $for$ :

$$pour\ i\ variant\ de\ 1\ à\ p :$$
$$instruction$$

Le temps d'exécution est $p$ fois le temps d'exécution de chaque $instruction$ plus $p$ (liées aux affectations de $i$)

### b) Notation de Landau

Prenons l'exemple suivant :

```
def expo(x,n):
    res = 1
    j = n
    while j>=1:
        res = res*x
        j = j-1
    return res
```

$$T(n) = \begin{cases} 2\ pour\ les\ 2\ affectations \\ n \times \begin{cases} 2\ pour\ les\ 2\ affectations \\ 2\ pour\ les\ deux\ opérations\ élémentaires \\ 1\ pour\ une\ comparaison \end{cases} \\ 1\ pour\ la\ dernière\ comparaison \\ 1\ pour\ le\ return \end{cases}$$

$$T(n) = 4 + 5n$$

Mathématiquement, $T(n)$ est une fonction linéaire de $n$. Pour $n$ assez grand $T(n)$ peut être approximée par une fonction $O$ linéaire en $n$. On écrit alors : $T(n) = O(n)$

- Complexité constante $T(n) = O(1)$ : signifie que le temps d'exécution est borné et indépendant de $n$. C'est le cas, par exemple, pour obtenir le dernier élément d'une liste.
- Complexité logarithmique $T(n) = O(log_2(n))$ : signifie que le temps d'exécution double, en élevant au carré la taille des données (ce qui est très performant). C'est le cas avec la recherche dichotomique dans une liste triée.
- Complexité linéaire $T(n) = O(n)$ : Cet ordre de grandeur peut s'obtenir avec une boucle for pour le calcul d'une somme ou d'une moyenne d'une liste.
- Complexité log-linéaire (ou quasi-linéaire) $T(n) = O(nlog_2(n))$. C'est le cas des algorithmes de tri fusion (complexité améliorable mais acceptable)
- Complexité quadratique $T(n) = O(n^2)$: c'est la complexité d'algorithmes construits avec des boucles imbriquées. C'est le cas lors d'un traitement photo pixel par pixel (ligne*colonne)
- Complexité polynomial $T(n) = O(n^k)$ et complexité exponentielle $T(n) = O(k^n)$ sont associées à des algorithmes peu performants.

**Commenté [AM11]:** Soient deux suites $u_n$ et $v_n$. Si pout tout $n$, il existe une constante $k$ telle que $u_n \leq kv_n$ alors on dit que $u_n$ est majorée par $v_n$. On écrit alors :
$u_n = O(n)$ ou aussi noté $u_n \in O(n)$

**Commenté [AM8]:** La complexité (ou coût) en espace est liée à la taille des variables utilisées. Par exemple avec les algorithmes de tri fusion, la création d'une nouvelle liste à chaque appel récursif peut s'avérer très gourmand. En effet, plus les données à conserver ont une taille importante, plus elles sont stockées dans une partie éloignée du processeur avec un temps d'écriture et de lecture qui s'allonge.

**Commenté [AM9]:** On a typiquement :
-quelque Go de mémoire vive
-quelques centaines de Go de mémoire disponibles dans les disques dures

**Commenté [AM10]:** Pour la boucle while, c'est au cas par cas (en fonction de la convergence du variant de boucle) Pour des instructions conditionnelles, on prend comme référence l'instruction qui prend le plus de temps.

**Commenté [AM12]:** Exemple de la suite de Fibonacci en récursif