COURS 10 - Les Algorithmes - Partie 2

3. Validité et coût

Lorsqu'on écrit un algorithme, il est impératif de vérifier que cet algorithme va produire un résultat en un temps fini et que ce résultat sera correct dans le sens où il sera conforme à une spécification précise. Nous dirons alors que l'algorithme est valide.

3.1. Validité d'un algorithme itératif

Un algorithme itératif est construit avec des boucles. Pour prouver qu'il est correct, nous disposons de la notion d'invariant de boucle.

Définition
Un invariant d'une boucle est une propriété qui est vérifiée avant l'entrée dans une boucle, à chaque passage dans cette boucle et à la sortie de cette boucle.
On peut faire le lien avec les suites définies par récurrence du programme de mathématiques. Pour démontrer qu'une propriété est un invariant d'une boucle, on utilise un raisonnement semblable au raisonnement par récurrence.

- On commence par vérifier que la propriété est vraie avant l'entrée dans la boucle. Cette étape s'appelle l'initialisation.
- On prouve ensuite que si la propriété est vraie avant un passage dans la boucle, alors elle est vraie après ce passage. Cette étape s'appelle l'hérédité.
  On peut alors conclure, que la propriété est vraie à la sortie de la boucle.

Exemple :

```
m = 0
p = 0
tant que m < a
    m = m + 1
    p = p + b
fin du tant que
```

Nous allons montrer que la propriété $p = m \times b$ est un invariant de la boucle "tant que".

- Avant le premier passage dans la boucle, $m = 0$ et $p = 0$, donc l'égalité $p = m \times b$ est vraie.
- Supposons donc que $p = m \times b$ avant un passage dans la boucle.
  Les nouvelles valeurs de m et p après le passage, notées m' et p' vérifient :
  $m' = m + 1$ et $p' = p + b$
  Alors
  $p' = m \times b + b = (m + 1) \times b = m' \times b$
  Donc la propriété est vraie après ce passage dans la boucle.

Nous pouvons conclure qu'à la sortie de la boucle, $p = m \times b$. Et puisqu'à la sortie de la boucle, la variable m a pour valeur celle de a, nous avons finalement obtenu le produit $p = a \times b$.

Terminaison
Un algorithme ne doit toujours comporter qu'un nombre fini d'étapes. Afin de prouver la terminaison d'un algorithme itératif, (qui contient une boucle), nous utilisons la notion de variant. Nous parlons ici de boucles conditionnelles. Dans le cas de boucles non conditionnelles, le nombre d'étapes est déterminé.

Méthode
On choisit un variant, c'est-à-dire une expression, la plus simple étant une variable, telle que la suite formée par les valeurs de cette expression au cours des itérations converge en un nombre fini d'étapes vers une valeur satisfaisant la condition d'arrêt.
Considérons par exemple le code suivant où la valeur de la variable a est un nombre quelconque :

```
x = 0
while x ** 2 < a:
    x = x + 1
```

Si la valeur de a est négative ou nulle, il n'y a aucun passage dans la boucle.
Sinon, la suite des valeurs de la variable x, le variant choisi, est 0, 1, 2, ..., n, et n est certainement la première valeur supérieure ou égale à la racine carrée de a. Le nombre de passages dans la boucle est donc fini.

Revenons sur l'exemple du produit de deux nombres étudié plus haut.
Nous avons prouvé qu'en sortie de boucle, la valeur de p était le produit a x b. Mais nous n'avons pas prouvé la terminaison, c'est-à-dire que la sortie de boucle était effective après un nombre fini de passages. Pour cela, dans cet exemple, nous choisissons comme variant la variable m. Cette variable prend pour valeurs successives 0, 1,....., a et il y a donc exactement a passages dans la boucle, ce qui prouve la terminaison.

3.2 Coût

Un programme doit traiter une liste de 107 éléments puis une liste de 108 éléments.
Est-ce que le temps d'exécution du programme sera multiplié par 10 ?
Quel est le rapport entre le temps d'exécution et la taille de la liste ?

Ce sont des questions auxquelles il faut réfléchir quand on écrit un algorithme.

Les réponses sont variées et dépendent de l'algorithme et de la liste.
Pour une liste donnée, un programme peut être plus rapide qu'un autre, mais avec une autre liste, ce peut être le contraire.
Le même programme peut être plus rapide avec la liste la plus longue.

De plus, pour traiter un même problème, non seulement nous pouvons disposer de plusieurs algorithmes mais un même algorithme peut avoir un temps d'exécution différent selon le langage de programmation utilisé et suivant la machine sur laquelle le programme est exécuté.
L'étude n'est pas simple à réaliser et pour comparer deux algorithmes nous allons ici nous concentrer sur le nombre d'opérations à effectuer en essayant d'évaluer un ordre de grandeur de ce nombre en fonction de la taille des données.

Nous parlerons du coût d'un algorithme ou de sa complexité.
Ce coût pouvant être très différent pour une même taille de données, nous nous placerons dans le pire des cas, celui où le coût est le plus important.

Étudions quelques exemples simples , commençons par l'algorithme précédent :

```
m = 0
p = 0
tant que m < a
    m = m + 1
    p = p + b
fin du tant que
```

Les passages dans la boucle ont lieu pour les valeurs de m égales à 0, 1, 2,....., a - 1 si la valeur de la variable a est un entier naturel a. Nous avons donc exactement a passages dans la boucle.
À chaque passage, nous comptons deux additions ainsi que deux affectations. Nous pouvons donc dire que le nombre d'additions est *2a* ou que le nombre d'opérations, au sens large en comptant les affectations, est *4a*.
Nous dirons alors que le coût est proportionnel à a ou qu'il est *linéaire* .

Avec une boucle "tant que", le calcul peut être plus compliqué puisque le nombre de passages dans la boucle varie avec les cas pour une même taille de données.
Nous devons alors identifier le pire des cas, c'est-à-dire celui où le nombre de passages est maximal. Considérons une boucle "pour" où le nombre de passages dans la boucle est bien déterminé.

```
somme = 0
for i in range(1, n+1):
        somme = somme + i
```

Cet algorithme, ou ce programme, permet de calculer la somme des entiers de *1 à n* .
Il y a clairement n passages dans la boucle. À chaque passage nous avons une addition et une affectation, donc un total de n additions et n affectations. Nous pouvons affirmer que le coût est **linéaire**.

Complexité linéaire
Soit n la taille d'une donnée. Si le nombre d'opérations à effectuer peut s'écrire $an + \beta$, avec α et β réels, α > 0, nous disons que l'algorithme a un coût linéaire ou une complexité linéaire.

Complexité quadratique
Soit n la taille d'une donnée. Si le nombre d'opérations à effectuer peut s'écrire $an^2 + \beta n + \gamma$, avec α, β et γ réels α > 0, nous disons que l'algorithme a un coût quadratique ou une complexité quadratique. Dans les codes qui suivent, les pointillés sous-entendent un nombre fixe d'opérations.

- Premier cas : n est la taille de la donnée, k est un nombre fixé.
  ```
  for i in range(n):
      ...
      for j in range(k):
          ...
  ```
  Nous avons *n* passages dans la première boucle. À chaque passage, nous avons un nombre fixe d'opérations *q puis k* passages dans la seconde boucle avec un nombre fixe d'opérations *r* .
  Donc pour chaque valeur de i, nous avons *q + (k x r)* opérations un nombre fini indépendant de n. Le nombre total d'opérations est donc *n x (q + (k x r))* qui peut s'écrire an et la complexité est donc **linéaire**.

- Deuxième cas : n est la taille de la donnée.
  ```
  for i in range(n):
      ...
      for j in range(n):
          ...
  ```
  Nous avons *n* passages dans la première boucle. À chaque passage, nous avons un nombre fixe d'opérations *q puis n* passages dans la seconde boucle avec un nombre fixe d'opérations *r*.
  Donc pour chaque valeur de i, nous avons *q + (n x r)* opérations un nombre fini. Le nombre total d'opérations est donc *n x (q + (n x r)) = r x* n^2 *+ q x n* qui peut s'écrire $an^2 + \beta n + \gamma$ et la complexité est donc **quadratique**.

- Troisième cas : n est la taille de la donnée.
  ```
  for i in range(n):
      ...
      for j in range(i):
          ...
  ```
  Nous avons _n_passages dans la première boucle. À chaque passage, nous avons un nombre fixe d'opérations *q* puis *i* passages dans la seconde boucle avec un nombre fixe d'opérations *r*.
  Donc pour chaque valeur de i, nous avons *q + (i x r)* opérations un nombre fini. Les valeurs de i sont successivement 0,1,2,3,....,n-1.
  Le nombre total d'opérations est donc *q + (q + 1 x r) + (q + 2 x r) + ... + (q + (n-1) x r = n x q + r x (1+2+3+...+(n-1)) .*
  La somme des n premiers entiers est connue, donc le résultat est :
  $nq + r \times \frac{n(n-1)}{2} = \frac{r}{2}n^2 + (q - \frac{r}{2})n$
  qui peut s'écrire $an^2 + \beta n + \gamma$ et la complexité est donc **quadratique**.

4. Exercices

Pour chacune des fonctions suivantes, calculez leur complexité.

In [1]:
```
def convert(n):
    h = n // 3600
    m = (n - 3600*h) // 60
    s = n % 60
    return h,m,s

n = 102152
print("{} secondes = {} h () min {} sec.".format(n,convert(n)[0],convert(n)[1],convert(n)[2]))
```

```
102152 secondes = 28 h 22 min 32 sec.
```

Saisissez votre réponse ici :

Dans la fonction "convert", il y a 3 affectations (h, m et s) et 5 opérations (n//3600, 3600h, *n*-3600h, (n-3600*h)//60 et n%60)
La complexité de cette fonction est donc égale à 8.

In [2]:
```
def maFonction(n):
    if n%3 == 0:
        p = n/3 + 2
    else:
        p = n*2 + 1
    return p

maFonction(23)
```

Out[2]:
```
47
```

Saisissez votre réponse ici :

Il y a d'abord un test (if) dans lequel il y a une opération (n%3), ce qui nous fait pour le moment une complexité de 2.
Ensuite, quelle que soit l'issue du test, il y a 1 affectation (p) et 2 opérations (pour calculer p), soit une complexité augmentée de 3.

Au final, on obtient donc une complexité de 5.

In [3]:
```
def recherche(l,x):
    for i in range(len(l)):
        if l[i]==x:
            return i
    return -1

recherche("Bonjour.","j")
```

Out[3]:
```
3
```

Saisissez votre réponse ici :

La fonction recherche a pour but d'afficher le rang où se trouve le caractère "x" dans la chaîne de caractères "l", et affiche "-1" si ce dernier n'est pas trouvé. Au niveau de la boucle for, il y a 1 addition (sur i), une affectation (i) et une comparaison (test si i<len(l)) et dans la boucle for, il y a un test (if l[i]==x).

Au final, si _n_ est la longueur de la chaîne de caractères, il y a 4n opérations élémentaires, qui correspond à la complexité de la fonction recherche.

In [4]:
```
def somme(n):
    s = 0
    for i in range(n+1):
        s += i
    return s

somme(100)
```

Out[4]:
```
5050
```

Saisissez votre réponse ici :

Il y a : 1 affectation (s = 0); au niveau de la boucle for, 3 opérations élémentaires (affectation sur i, opération d'incrémentation sur i, test sur i); dans la boucle for, 2 opérations élémentaires (1 affectation sur s et une opération sur s).
Ainsi, il y a 5 opérations élémentaires dans la boucle, répétées n fois, plus la première (hors boucle).

La complexité de la fonction somme est donc égale à 5n+1

In [5]:
```
def mystere(n):
    m = 0
    for i in range(n):
        for j in range(i):
            m += i+j
    return m

mystere(100)
```

Out[5]:
```
490050
```

Saisissez votre réponse ici :

Il y a 1 affectation dès le début (m=0)
au niveau de la boucle sur i, 3 opérations élémentaires
au niveau de la boucle sur j, il y a 3 opérations élémentaires répétées i fois
dans la boucle sur j, il y a 3 opérations élémentaires (1 affectation sur m, une somme sur m et une somme de i et j)
Ainsi, à part la première affectation, il y a 27 opérations élémentaires pour chaque valeur de i possible.

Il y a :
1 valeur possible de j quand i=0 (j=0) 1 valeur possible de j quand i=1 (j=0) 2 valeurs possibles de j pour i=2 (j=0 et j=1) 3 valeurs possibles de j pour i=3 (j=0, j=1 et j=2) ... n-1 valeurs possibles de j pour i=n-1

La complexité est donc égale à : $1 + 1 + 27 \times (1+2+3+\cdots+(n-1)) = 2 + 27 \times \frac{n(n-1)}{2} = \frac{27}{2}n^2 - \frac{27}{2}n + 2$