Chapitre 4 : Analyse d'un programme

Les problèmes de sécurité (données personnelles, systèmes sensibles comme un hôpital) et de sûreté (éviter des pannes aux conséquences parfois dramatiques), sont devenus des questions très importantes dans notre société de plus en plus connectée. Il est ainsi primordial que les programmes fonctionnent correctement.
Un programme peut être exécuté seul ou utilisé par un autre programme et dans ce cas, il est une donnée de cet autre programme. Il doit produire en sortie le résultat attendu d'après la spécification quelles que soient les données en entrées pour ne pas provoquer des erreurs en chaîne qui pourraient être catastrophiques.

1. Validité d'un algorithme

Lorsqu'on écrit un algorithme, il est nécessaire de s'assurer que :
- l'algorithme fournit un résultat après un nombre fini d'étapes (problème de terminaison)
- que ce résultat correspond à celui attendu (problème de correction).

Si ces deux conditions sont bien respectées alors l'algorithme peut être considéré comme valide. Si, lorsqu'il termine, l'algorithme donne la réponse attendue, on parle de correction partielle. Si la terminaison est assurée dans tous les cas et que la réponse est totale, on parle de correction totale.
Nous allons introduire deux notions permettant de vérifier la validité d'un algorithme :
- La notion de variant de boucle permettra de prouver la terminaison
- La notion d'invariant permettra quant à elle de prouver la correction

1.1. Terminaison d'un algorithme

Dans un algorithme non récursif (algorithme itératif) correctement structuré, et sans erreur de programmation, les seules structures pouvant amener une non terminaison d'un algorithme sont les boucles conditionnelles. Nous n'évoquerons pas dans ce cours la terminaison et la correction d'un algorithme récursif (étude se ramenant le plus souvent à une récurrence).

Pour l'étude de la terminaison, partons de l'exemple de l'algorithme d'Euclide.

Entrée : a,b : entiers positifs
Sortie : d : entier

tant que $b>0$ **faire**
    $(a,b) \leftarrow (b, a \bmod b)$
fin tant que
renvoyer a

Remarques :
- l'algorithme d'Euclide permet de déterminer le PGCD de deux entiers a et b, noté $a \wedge b$
- $a \bmod b$ signifie qu'on prend le reste de la division euclidienne de a par b
- l'algorithme d'Euclide utilise le fait que si on a :

$$a = b \times q + r$$

alors $a \wedge b = b \wedge r$

Exercice : appliquer l'algorithme d'Euclide au couple $(164;36)$. Que dire de l'évolution de la quantité b au fil des itérations successives ?

Corrigé :

La preuve mathématique classique de la terminaison de l'algorithme d'Euclide passe par le principe de descente infinie : les restes successifs (c'est-à-dire les valeurs successives de b) forment une suite strictement décroissante d'entiers positifs. Nécessairement, il arrivera une étape pour laquelle la condition d'entrée dans la boucle ($b>0$) ne sera plus validée ce qui mettra fin au programme.

D'une manière générale, nous pouvons dire qu'afin de prouver la terminaison d'un algorithme, on exhibe un variant de boucle c'est-à-dire une variable dont les valeurs prises au cours des itérations constituent une suite qui converge en un nombre fini d'étapes vers une valeur satisfaisant à la condition d'arrêt de la boucle. Ce variant v doit être tel que :

- $v \in \mathbb{N}$;
- v décroît strictement à chaque passage dans la boucle (si on note v_e et v_f les valeurs en entrée et sortie d'un passage dans la boucle, alors on a $v_e > v_f$).

Le principe de descente infinie permet alors d'affirmer que :

- si, pour une boucle donnée, on peut exhiber un variant de boucle, alors le nombre de passages dans la boucle est finie.
- si, pour un algorithme donné, on peut exhiber, pour toute boucle de l'algorithme, un variant de boucle, alors l'algorithme s'arrête en temps fini.

Exercice : considérons l'exemple du programme suivant :

```
def puissance(a, n):
    """
    a -- entier strictement positif
    n -- entier naturel
    renvoie le nombre a exposant n (a**n)
    """
```

```
A = a
N = n
p = 1

while N > 0:
    if N%2 == 0:
        A = A*A
        N = N//2
    else:
        p = p * A
        N = N - 1
return p
```

Montrer que cette fonction se termine en explicitant un variant.

1.2.Correction d'un algorithme

La terminaison d'un algorithme n'assure pas que le résultat obtenu est bien le résultat attendu, c'est-à-dire que l'algorithme répond bien à la question posée initialement. L'étude de la validité du résultat qu'on obtient fait l'objet de ce paragraphe. On parle de l'étude de la correction de l'algorithme.

Reprenons l'exemple de l'algorithme d'Euclide. La preuve mathématique classique de la correction de l'algorithme repose sur la constatation suivante : si r est le reste de la division euclidienne de a par b, alors $a \wedge b = b \wedge r$. La preuve mathématique de ce point ne pose pas de problème.

On constate alors que la quantité $w = a \wedge b$ prend toujours la même valeur à l'entrée d'une boucle (correspondant aussi à la valeur à la sortie de la boucle précédente). C'est le fait d'avoir une valeur constante qui nous assure que la valeur initiale recherchée $a \wedge b$, est égale à $a \wedge b$ pour les valeurs finales obtenues pour a et b. Or, en sortie de boucle $b = 0$, donc $a \wedge b = a$. Ainsi, le PGCD des valeurs initiales de a et b est égal à la valeur finale de a.

De façon générale, l'étude de la correction d'un algorithme se fera ainsi par la recherche d'une quantité invariante d'un passage à l'autre dans la boucle.

On appelle invariant une propriété liée aux variables $(x_1,...,x_k)$ du programme telle que :

- la propriété est vraie avant l'entrée dans la boucle ;
- La propriété est vraie après chaque passage dans la boucle.

Dès lors qu'on exhibe un invariant, montrer la correction d'un algorithme se résume en trois étapes :

- initialisation : montrer que l'invariant est vrai avant la première itération.
- conservation : montrer que si l'invariant est vrai avant une itération, il reste vrai avant l'itération suivante.
- terminaison : déduire de l'invariant final la propriété voulue.

Exercice : considérons l'exemple du programme suivant qui retourne le maximum d'un tableau :

```
def max(tab):
    """
    tab -- liste de valeurs numériques
    renvoie la valeur maximale contenue dans tab
    """
    max=tab[0]
    for i in range(1,len(tab)):
        if tab[i]>max:
            max=tab[i]
    return max
```

Prouver la correction de la fonction `max`.

Retenons de cette étude que non seulement, l'étude de la terminaison et de la correction d'un algorithme permet de prouver sa validité, mais que par ailleurs, cette étude permet de détecter des erreurs dans l'algorithme le cas échéant.

2. Complexité d'un programme

Lorsqu'un algorithme est correct, il doit encore, avant d'être programmé et exécuté, satisfaire à deux impératifs en terme 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

D'une manière générale, on parle de complexité d'un programme ou de son coût.

2.1.Complexité en espace

Le but de l'étude de la complexité en mémoire, que nous aborderons très peu ici, est d'étudier l'encombrement en mémoire du fait de l'exécution d'un algorithme. Cette encombrement peut provenir de mises en mémoire explicites, ou d'empilements implicites de données et d'instructions, par exemple dans le cadre d'un algorithme récursif.
Une mauvaise complexité en mémoire ralentit le programme (car la mise en mémoire doit se faire loin du processeur), et peut aller jusqu'à la saturation de la mémoire, cas dans lequel l'algorithme ne peut terminer son exécution.

2.2.Temps d'exécution

Étudier la complexité temporelle consiste à évaluer le temps d'exécution d'un algorithme en fonction de la taille des données en entrée. Or, le temps d'exécution d'un programme dépend de la machine, du langage de programmation et de l'algorithme. Toutefois, pour des entrées de grandes tailles, l'algorithme est la cause déterminante du temps d'exécution. C'est la part de l'algorithme qui est évaluée lorsqu'on s'intéresse à l'étude de la complexité temporelle.

La prise en compte du coût d'un algorithme est très importante comme l'illustre l'exemple suivant : on cherche à trier une liste. On dispose de deux programmes fondés chacun sur deux algorithmes exacts :
- le tri par insertion
- le tri fusion

On suppose que les deux programmes sont :
- écrits en langage Python
- exécutés sur la même machine

Avec chacun des deux programmes, trions par ordre croissant une liste aléatoire de N éléments pour N allant de 100 à 10^8.

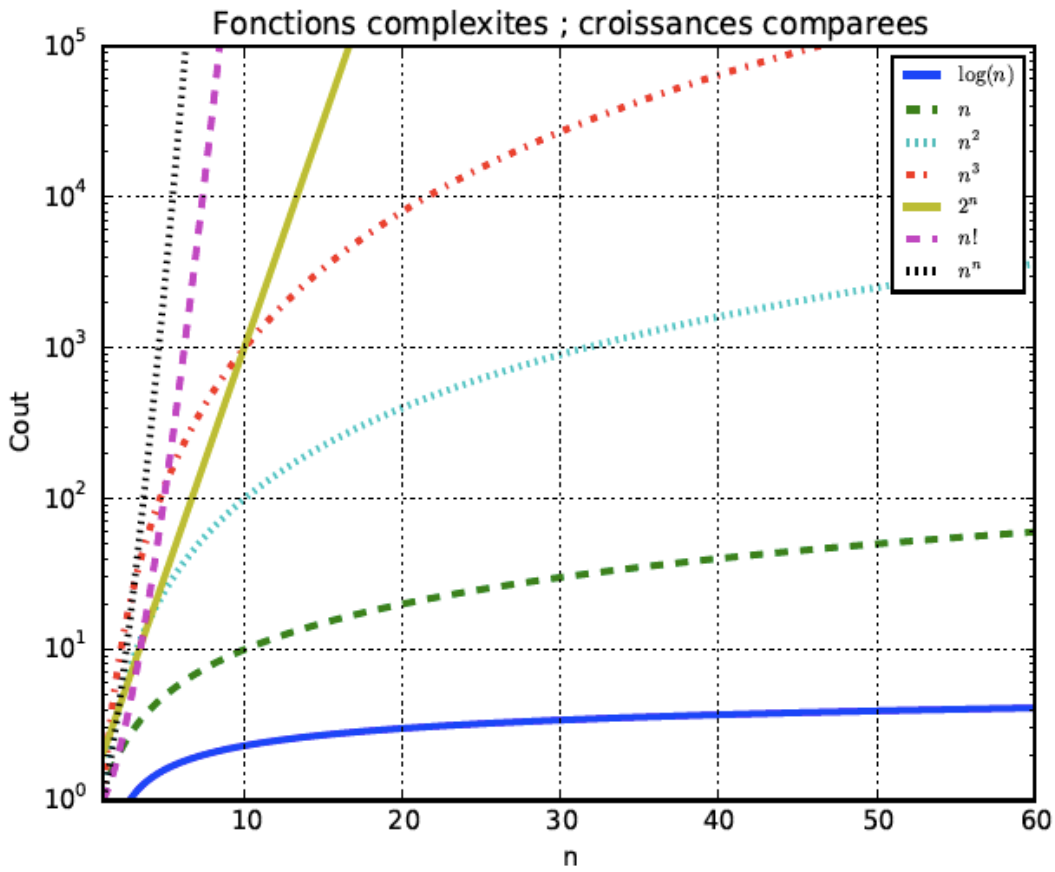
N	10^2	10^3	10^4	10^5	10^6	10^7	10^8
Tri Fusion	$0{,}3\ ms$	$0{,}9\ ms$	$5{,}5\ ms$	$20\ ms$	$215\ ms$	$2{,}3\ s$	$25{,}8\ s$
Tri Insertion	$1{,}8\ ms$	$67\ ms$	$5{,}7\ s$	$11\ \text{min}$	$19\ h$	$66\ jour$	$21\ ans$

2.3.Niveaux de complexité

Soient deux suites (u_n) et (v_n). On dit que $u_n = \Theta(v_n)$ s'il existe $m, M > 0$ tels que $m \leq \frac{u_n}{v_n} \leq M$ à partir d'un certain rang. Ainsi, (u_n) et (v_n) restent dans un rapport comparable.

Ainsi, pour évaluer la complexité temporelle d'un programme, on pourra estimer son coup, $C(n)$, en fonction de n qui représentera la taille des données en entrée et le comparer à des suites connues. Les différents comportements de référence sont les suivants :

- Coût constant : $C(n)=\Theta(1)$
- Coût logarithmique : $C(n)=\Theta(\ln(n))$
- Coût linéaire : $C(n)=\Theta(n)$
- Coût quasi-linéaire : $C(n)=\Theta(n\ln(n))$
- Coût quadratique : $C(n)=\Theta(n^2)$
- Coût cubique : $C(n)=\Theta(n^3)$
- Coût polynomial : $C(n)=\Theta(n^\alpha), \alpha>0$
- Coût exponentiel : $C(n)=\Theta(x^n), x>1$.



Pour information, voici les temps approximatifs de calcul pour un processeur effectuant un milliard d'opérations par seconde, et pour une donnée de taille $n=10^6$, suivant les coûts :

- $\Theta(1)=1\ ns$
- $\Theta(\ln(n))=15\ ns$
- $\Theta(n\ln(n))=15\ ms$
- $\Theta(n^2)=15$ min
- $\Theta(n^3)=30\ ans$
- $\Theta(2^n)=10^{300000}$ milliards d'années

Ainsi, si on est amené à traiter des grandes données, l'amélioration des complexités est un enjeu important !

En pratique, se rendre compte de l'évolution de la complexité n'est par très dur : pour des données de taille suffisamment grande, en doublant la taille des données :
- on n'augmente par le temps de calcul pour un algorithme en temps constant
- on augmente le temps de calcul d'une constante (indépendante de n) pour un coût logarithmique
- on double le temps de calcul pour un algorithme linéaire
- on quadruple le temps de calcul pour un algorithme quadratique
- on multiplie le temps de calcul par 8 pour un algorithme cubique

2.4.Coût d'un algorithme

Chaque opération effectuée nécessite un temps de traitement. Les opérations élémentaires effectuées au cours d'un algorithme sont :
- l'affectation
- les comparaisons
- les opérations sur des données numériques.

Évaluer le coût $C(n)$ d'un algorithme revient alors à sommer tous les temps d'exécution des différentes opérations effectuées lors de l'exécution d'un algorithme. On utilise pour cela une hypothèse simplificatrice : le modèle à coût fixe. Ce dernier consiste à considérer que le temps d'exécution est le même pour toutes les opérations (coût fixe) et que cette durée ne dépend pas de la taille des objets (notamment des entiers), et que les opérations sur les flottants sont de durée similaire aux opérations semblables sur les entiers.

Toutefois, il serait trop fastidieux d'exprimer de cette manière $C(n)$ pour chaque algorithme.
On raisonnera plutôt en s'appuyant sur les propriétés ci-après.
- Si deux blocs d'instructions successifs ont une complexité en $\Theta(u_n)$ alors la complexité totale est en $\Theta(u_n)$.
- Si on répète u_n fois un bloc d'instructions de complexité en $\Theta(v_n)$ alors la complexité totale est en $\Theta(u_n v_n)$. Si u_n a une valeur constante alors la complexité totale est en $\Theta(v_n)$.
- Si deux blocs successifs ont une complexité en $\Theta(u_n)$ pour le premier et en $\Theta(v_n)$ pour le second alors la complexité totale est en $\Theta(\max(u_n, v_n))$.
- La complexité d'une boucle non-conditionnelle correspondant à n passages dans la boucle est $\Theta(n)$.
- La complexité d'une boucle conditionnelle ne peut être connue de manière exacte. On devra considérer la complexité dans le meilleur ou le pire des cas.

Quelques exemples :
- Premier cas : n est la taille de la donnée et k un nombre fixé

```
for i in range(n):
      …
      for j in range(k):
            …
```

Il y a n passages dans la boucle externe. Pour chaque passage dans la boucle externe, on a un nombre fixe q d'opérations et k passages dans la boucle interne. Soit r le nombre fixe

d'opérations de la boucle interne. Pour chaque passage dans la boucle externe, on a donc $q+k\times r=\alpha$ opérations. Le nombre total d'opérations est donc $\alpha\times n$ et la complexité est donc linéaire.
On peut simplifier le raisonnement en affirmant que la boucle interne est en $\Theta(k)$ (donc à coût constant) donc l'ensemble des deux boucles imbriquées est en $\Theta(n\times k)=\Theta(n)$ soit un coût linéaire.

- Deuxième cas : n est la taille de la donnée

```
for i in range(n):
      …
      for j in range(n):
            …
```

Il y a n passages dans la boucle externe. Pour chaque passage dans la boucle externe, on a un nombre fixe q d'opérations et n passages dans la boucle interne. Soit r le nombre fixe d'opérations de la boucle interne. Pour chaque passage dans la boucle externe, on a donc $q+n\times r=\alpha$ opérations. Le nombre total d'opérations est donc $\alpha\times n=q\times n+n^2\times r$ et la complexité est donc quadratique.
On peut simplifier le raisonnement en affirmant que la boucle interne est en $\Theta(n)$ (donc à coût linéaire) donc l'ensemble des deux boucles imbriquées est en $\Theta(n\times n)=\Theta(n^2)$ soit un coût linéaire.

- Troisième cas : n est la taille de la donnée :

```
for i in range(n):
      …
      for j in range(i):
            …
```

Il y a n passages dans la boucle externe. Pour chaque passage dans la boucle externe, on a un nombre fixe q d'opérations et i passages dans la boucle interne. Soit r le nombre fixe d'opérations de la boucle interne. Pour chaque passage dans la boucle externe, on a donc $q+i\times r=\alpha$ opérations. Le nombre total d'opérations est donc $\sum_{i=0}^{n-1}(q+i\times r)=n\times q+\frac{(n-1)\times r}{2}\times n=n\times\left(q-\frac{r}{2}\right)+n^2\times\frac{r}{2}$ et la complexité est donc quadratique.

3. Tests d'un programme

On peut construire un jeu de tests afin de s'assurer que le programme fonctionne correctement. Cela consiste à définir un ensemble de données qui seront utilisées afin de vérifier que le programme retourne bien les résultats attendus avec ces données.

On peut bien entendu utiliser la fonction `print` pour afficher quelques réponses et vérifier directement si elles sont correctes. Toutefois, il sera plus pratique d'avoir recours aux assertions.

On peut distinguer différents types de tests :

- Tester quelques cas simples typiques (pour une utilisation basique du programme)
- Tester des valeurs extrêmes, des cas limites, des cas interdits
- Tester un nombre important de données (choisies de manière aléatoire par exemple)
- Tester des cas qui pourraient nécessiter un temps d'exécution important afin de pouvoir évaluer l'efficacité du programme.

Considérons l'exemple de la fonction permute ci-dessous avec sa spécification et ses commentaires.

```
def permute(liste):
    """ liste est de type list
        la fonction permute le premier et le dernier élément
        et renvoie une nouvelle liste
        permute [1,2,3,4] en [4,2,3,1]"""
    copie=liste[:] # une copie superficielle de la liste
    n=len(copie)
    copie[0],copie[n-1]=copie[n-1],copie[0] # permutation des valeurs
    return copie
```

On associe un jeu de tests en considérant quelques cas typiques :

- Une simple liste de nombres
- Une liste dont les éléments sont des listes
- Une liste qui contient deux éléments ou un élément unique
- Une liste vide

```
assert permute([1,2,3,4])==[4,2,3,1]
assert permute([[1,2],[3,4],[5,6]])==[[5,6],[3,4],[1,2]]
assert permute([1,2])==[2,1]
assert permute([1])==[1]
assert permute([])==[]
```

En exécutant le programme, nous remarquons que la fonction effectue bien ce qui est prévu même pour une liste ne contenant qu'un seul élément. Par contre, le cas d'une liste vide n'est pas traité et une erreur et interrompt le programme.

Ceci peut être corrigé de la manière suivante :

```
def permute(liste):
    """ liste est de type list
        la fonction permute le premier et le dernier élément
        et renvoie une nouvelle liste
        permute [1,2,3,4] en [4,2,3,1]"""
    if liste == []:
        return []
    copie=liste[:] # une copie superficielle de la liste
    n=len(copie)
    copie[0],copie[n-1]=copie[n-1],copie[0] # permutation des valeurs
    return copie
```