4 | Preuve et efficacité d'un algorithme

Savoir si son algorithme est "satisfaisant" est une des préoccupations principales du programmeur. Derrière ce mot se cachent en fait plusieurs questions fondamentales en algorithmique :

- L'algorithme s'arrête-t-il bien ou tourne-t-il indéfiniment ? (**terminaison** de l'algorithme)
- L'algorithme renvoie-t-il bien le résultat que l'on attend ? (**correction** de l'algorithme)
- Selon la taille des données, s'exécute-t-il en un temps raisonnable ? On parle d'**efficacité,** quantifiée par un calcul de **complexité** de l'algorithme.

1 Terminaison d'un algorithme

On dit qu'**un algorithme termine** s'il s'arrête au bout d'un nombre fini d'étapes. On comprend donc que ce problème se pose essentiellement dès que le code contient des boucles (boucle *for*, boucle *while* ou appel récursif) et que connaître le nombre d'instructions du programme permet de démontrer immédiatement sa **terminaison**.

Pour montrer qu'un algorithme se termine, on montre qu'il existe une **quantité** v, appelée **variant**, qui vérifie les conditions suivantes :

- En entrée et en sortie de boucle, v ne prend que des valeurs entières et positives
- v décroît strictement à chaque itération.

Comme une **suite strictement décroissante d'entiers est nécessairement finie**, alors la simple existence de v permet de prouver que l'algorithme se termine effectivement (c'est-à-dire qu'il y a un nombre fini d'itération). On appelle cette quantité un **variant** (ou un **convergent**) **de boucle**.

1.1 Cas de la boucle for

Dans le cas d'une boucle for, un variant simple est obtenu à partir du compteur de la boucle. En effet, si la structure de la boucle est du type `for i in range(a,b)` alors on peut définir le variant $\boldsymbol{v = b - i}$ dont les valeurs successives à chaque itération forment bien une suite strictement décroissante d'entiers positifs.

L'algorithme s'arrête donc en un nombre fini d'opérations $N = (b-1) - a + 1$ soit $= b - a$.

Remarque : on rappelle que l'instruction `range(a,b)` créée un itérateur de valeurs comprises entre a et `b-1`.

1.2 Cas de la boucle while

Dans le cas de la boucle while, le choix du variant n'est pas toujours aussi immédiat, tout simplement parce qu'il n'est pas toujours clair que la boucle termine. Etudions cela avec 2 exemples classiques et un troisième plus délicat :

1.2.1 Exemple 1 : Division euclidienne

```
>>> def diveucl(a,b) :          # b > 0
>>>     r,q = a,0
>>>     while r >= b :
>>>             q = q + 1
>>>             r = r – b
>>>     return (q,r)
```

Conseil : si le convergent de boucle n'est pas évident, détailler l'exécution du script pour les premières itérations avec le cas général ou des valeurs simples (mais néanmoins significatives) et chercher une quantité décroissante.
Ainsi, si on appelle par exemple la fonction `div_eucl(10,3)` :

Etat initial	r=10	q=0
Après la 1ère itération	r=7	q=1
Après la 2ème itération	r=4	q=2
Après la 3ème itération	r=1	q=3

On voit ici que $\boldsymbol{v = r}$ semble être un convergent de boucle satisfaisant mais que $\boldsymbol{v = r - b}$ pourrait également convenir.

Il est standard de noter r=(r_i) le variant, puisqu'il s'agit en fait d'une suite, l'indice indiquant le numéro de l'itération.

a) On constate bien ici que b étant un entier strictement positif alors $r_i = r_{i-1} - b$ reste positive tout au long de l'algorithme puisque la condition d'entrée de boucle impose $r_{i-1} - b \geq 0$.
b) De plus, à chaque itération la variable r diminue de la valeur $b > 0$, la suite (r_i) est donc strictement décroissante.

1.2.2 Exemple 2 : Recherche d'un élément dans une liste

```
>>> def rechercheElmt (L,x):
>>>     trouve = False
>>>     pos=0
>>>     while pos < len(L) and not trouve:
>>>             if x == L[pos]:
>>>                     trouve = True
>>>             else:
>>>                     pos+=1
>>>     return trouve
```

Ici, la grandeur qui décroît strictement à chaque itération est len(L)-pos. En effet, tant que l'algorithme ne se termine pas, pos est incrémenté. La condition d'arrêt est len(L)-pos <=0 ou trouve

= True. Si l'élément n'est pas dans la liste, la quantité décroissante finit invariablement par vérifier la condition d'arrêt.

Noter que dans cet exemple, on manipule une boucle while avec un compteur et un critère d'arrêt lié au compteur, le variant est donc facile à trouver. Ici, une boucle for aurait aussi bien fait l'affaire.

1.2.3 Exemple 3(bonus)

```
>>> def f(b) :
>>>     a = 0
>>>     while b > 0 :
>>>             if a == 0 :
>>>                     b = b – 2
>>>                     a = 1
>>>             else :
>>>                     b = b + 1
>>>                     a = 0
>>>             print ( (a, b) )
```

Exécution avec f(4) :

Etat initial	a = 0	b = 4
Après la 1ère itération	a = 1	b = 2
Après la 2ème itération	a = 0	b = 3
Après la 3ème itération	a = 1	b = 1
Après la 4ème itération	a = 0	b = 2
Après la 5ème itération	a = 1	b = 0

Dans cet exemple le choix du variant de boucle n'est pas aussi immédiat que dans le 1er exemple. On essaie de trouver une quantité v_i définie à partir des valeurs a_i et b_i des variables a et b après la $i^{ème}$ itération.
La suite (v_i) doit être une suite strictement décroissante d'entiers positifs.

On se propose de choisir, pour l'itération numéro i, le variant $\boldsymbol{v_i = 3a_i + 2b_i}$.

a) $\forall i \geq 0$, v_i est un entier naturel comme somme de 2 entiers naturels.
b) A l'entrée de la $i^{ème}$ itération :
   - Si $a_{i-1} = 0$ alors à la fin de l'itération on a : $b_i = b_{i-1} - 2$ et $a_i = 1$
     donc $v_i = 3a_i + 2b_i = 3 + 2b_{i-1} - 4 = 2b_{i-1} - 1$ et $v_{i-1} = 3a_{i-1} + 2b_{i-1} = 2b_{i-1}$
     d'où : $v_i < v_{i-1}$
   - Si $a_{i-1} = 1$ alors à la fin de l'itération on a : $b_i = b_{i-1} + 1$ et $a_i = 0$
     Donc $v_i = 3a_i + 2b_i = 2b_{i-1} + 2$ et $v_{i-1} = 3a_{i-1} + 2b_{i-1} = 3 + 2b_{i-1}$
     d'où : $v_i < v_{i-1}$

Finalement (v_i) est une suite strictement décroissante d'entiers naturels, nécessairement finie, et son existence prouve que la boucle se termine.

2 Correction d'un algorithme

Une fois qu'on sait qu'un algorithme se termine, il est nécessaire de vérifier que **cet algorithme renvoie bien le résultat escompté**. Cette démonstration de la validité d'un algorithme constitue ce qu'on appelle sa **correction**.
Comme avec la terminaison, les difficultés qui peuvent être rencontrées sont posées par les boucles ou les appels récursifs. Dans le cas où l'algorithme ne contient qu'une suite d'instructions, il suffit de les exécuter pour vérifier, une fois arrivé à la fin, le contenu des variables.

La preuve de la correction d'un algorithme passe par l'existence d'une propriété, appelée **invariant de boucle**. Il s'agit d'une propriété qui est vraie en entrée de boucle et reste vraie à chaque itération. En sortie, elle permet d'en déduire que le programme renvoie bien le résultat attendu. L'invariant vérifie donc les conditions suivantes :

- La propriété est vérifiée avant d'entrer dans la boucle.
- Si la propriété est vérifiée avant la $n^{ième}$ itération alors celle-ci doit rester vraie après l'itération.
- Après la dernière itération, en sortie de boucle, la propriété doit permettre de déduire que la boucle produit le résultat attendu.

2.1 Cas de la boucle for

Dans le cas d'une boucle for la démonstration se fait en 3 étapes, reprenant les conditions énoncées précédemment.
Prenons l'exemple d'une boucle `for i in range(a,b+1)` :

- On note $P(k)$ la propriété après la $k^{ième}$ itération.
- ***Initialisation :*** on montre que $P(a)$ est vérifiée.
- ***Transmission/Hérédité :*** soit $i \in [a, b-1]$ pour lequel $P(i)$ est vraie. On montre alors que $P(i+1)$ l'est aussi.
- ***Sortie :*** on déduit de $P(b)$, qui est vraie, que la boucle renvoie bien le résultat attendu.

Exemple : puissance de 2

```
>>> def f(n) :
>>>     p = 1
>>>     for i in range (1, n+1) :
>>>             p = p * 2
>>>     return(p)
```

On choisit comme invariant de boucle la propriété $P(i)$, $p_i = 2^i$ après la $i^{ème}$ itération.

On suppose $n > 0$ sinon la boucle ne s'effectue pas mais on pose que $p_0 = p = 1$.

Initialisation : Après la 1ère itération, pour $i = 1$ on a : $p_1 = p * 2 = 2^1$. Donc $P(1)$ est vraie.

Transmission : On admet que $P(i)$ est vraie après la $i^{ème}$ itération.
Après la $(i+1)^{ème}$ itération on a $p_{i+1} = p_i * 2 = 2^i \times 2 = 2^{i+1}$ donc $P(i+1)$ est vraie également.

Sortie : $P(i)$ est vraie pour chaque $i \in [1, n-1]$ donc $p_n = 2^n$ est vraie en sortie de boucle.

2.2 Cas de la bouche while

Dans le cas d'une boucle while, la boucle ne termine pas de la même façon qu'une boucle for: la propriété en sortie de boucle $P(n)$ est vérifiée, mais un ultime test sur $P(n+1)$ est effectué pour sortir de la boucle. Il est alors souvent nécessaire d'**utiliser à la fois la propriété vraie $P(n)$ et celle qui est fausse pour la première fois** pour pouvoir déduire le résultat attendu.
Le choix de l'invariant de boucle est crucial, car s'il a été correctement choisi, la démonstration vient ensuite d'elle-même. Toute la difficulté est donc de trouver un invariant de boucle judicieux.

Exemple : La factorielle, n!, que l'on code code de manière diffrente de d'habitude.

```
>>> def f(n) :
>>>     p, q = 1, n
>>>     while q > 0 :
>>>             p = p *q
>>>             q = q -1
>>>     return (p)
```

Le choix de l'invariant de boucle n'est ici pas immédiat car on ne peut pas, après chaque itération, choisir l'invariant $p_i = i$! (on voit bien que cela est faux et qu'à chaque itération on n'a pas $p_i = i$!).
Il nous faut donc trouver une propriété qui reste vraie à chaque itération. Dressons ici le tableau des valeurs pour les premières itérations ($i = 0$ correspond aux variables avant la 1ère itération) afin de se faire une idée :

$i = 0$	$p_0 = 1$	$q_0 = n$
$i = 1$	$p_1 = n$	$q_1 = n - 1$
$i = 2$	$p_2 = n(n-1)$	$q_2 = n - 2$
$i = 3$	$p_3 = n(n-1)(n-2)$	$q_3 = n - 3$

On constate qu'à la fin de chaque itération, la "quantité" $p_i \times q_i!$ reste toujours constante et égale à n !.
Il semblerait donc qu'on puisse choisir comme invariant de boucle la propriété $P(i)$: $p_i \times q_i! = n!$

Initialisation *:* Pour $i = 0$, on a $p_0 = 1$ et $q_0 = n$ soit $p_0 \times q_0! = n!$ donc $P(0)$ est vraie.
Transmission *:* Pour $i \in \mathbb{N}$, on admet que $P(i)$ est vraie.
Après la $i^{ème}$ itération, on a donc : $p_{i+1} = p_i \times q_i$
et $q_{i+1} = q_i - 1$
donc $p_{i+1} \times q_{i+1}! = p_i \times q_i \times (q_i - 1)! = p_i \times q_i! = n!$
Donc $P(i+1)$ est vraie.
Sortie *:* Il existe un certain rang $i \in \mathbb{N}$ tel que la condition de boucle $q > 0$ ne soit plus vérifiée (pour l'affirmer, il aurait fallu étudier la terminaison avant, q étant un variant de boucle trivial).
La première fois que cette condition n'est pas vérifiée, on a $q = 0$ et $p = \frac{n!}{0!} = n!$

L'invariant de boucle choisi prouve qu'en sortie de boucle, la variable p contient bien $n!$

3 Jeu de tests

Construire un jeu de tests consiste à définir un ensemble de données qui vont être utilisées pour **vérifier que le programme produit bien les résultats attendus** avec ses données. L'objectif n'est pas de prouver qu'il n'y a aucune erreur mais d'en trouver pour corriger ces défauts. Ces vérifications peuvent s'effectuer de plusieurs manières. On peut utiliser la fonction **print** pour afficher quelques réponses et vérifier visuellement qu'elles sont correctes. Pour un nombre conséquent de tests, il est plus pratique d'utiliser des **assertions**. Un message d'erreur est affiché seulement pour les cas qui posent problème.

Dans la construction d'un jeu de tests, on distingue plusieurs 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.

Pour effectuer ces tests, on **partitionne de domaine d'entrée**. On utilise ensuite une donnée représentante de chaque partie, pour effectuer un test en vérifiant la sortie.
Globalement, la construction des tests est liée à la spécification de l'algorithme testé ainsi qu'aux conditions sur les arguments et les résultats. L'étude directe du code permet également de compléter ces tests.

4 Complexité d'un algorithme

L'exécution d'un programme mobilise un certain nombre de ressources, dont les deux principales sont **le temps de calcul** du processeur et **l'occupation de la mémoire**. La conception d'un programme doit donc **limiter ces besoins** autant que possible.
Il est souvent très difficile de connaître précisément l'efficacité d'un algorithme car cela dépend de nombreux paramètres physiques liés à l'ordinateur (processeur, mémoire, etc.) autant que de paramètres liés au langage de programmation utilisé (ainsi qu'au compilateur et à l'interpréteur).

Ainsi, on se contentera bien souvent d'estimer le coût d'un algorithme, en temps ou en taille-mémoire, en déterminant ce qu'on appelle sa complexité.

On parle de **complexité temporelle** lorsque l'on s'intéresse au temps d'exécution et de **complexité spatiale** pour l'espace mémoire.

4.1 Notion de coût d'un algorithme

On part du postulat suivant (**modèle à coût fixe**), acceptable et généralement admis, bien que pas tout à fait exact :

1) Le coût d'un algorithme est le nombre d'opérations élémentaires significatives effectuées.
2) Les opérations élémentaires (ayant un coût constant de 1) sont :
   - Les opérations mathématiques usuelles et logiques ($+, -, \times, \div, and, or, not$)
   - La lecture de valeur dans une variable
   - L'affectation de valeur à une variable
   - La réalisation d'un test, d'une évaluation ou d'une comparaison

3) Le temps d'exécution est proportionnel à la taille des données traitées

Remarque :

Pour une boucle **`for i in range(n)`** le coût est de n opérations.

Pour une boucle **`while`**, comme le nombre d'itérations n'est pas toujours simple à déterminer, on cherchera plutôt à le majorer (voir 4.2).

Exemple III.1.1

```
def f(n):
    r, i = 0, 1
    while i <= n :
        r = r + i
        i = i + 1
    print(r)
```

Lecture : *2n+3n+1*
Affectation : *3 + 2n*
Test : *n + 1*
Affichage : 1
Addition : 2n
→ environ 10n opérations

Exemple III.1.2

```
def f(n,p):
    for i in range(n) :
        for j in range(p) :
            print(i*j,end="")
        print()
```

Boucles sur i : n fois
Boucles sur j : p fois

→ environ $n \times p$ opérations

Exemple III.1.3

```
def f(n):
    for i in range(n) :
        for j in range(i+1) :
            print("*",end="")
        print()
```

Boucles sur i : n fois
Boucles sur j : i+1 fois

$\rightarrow environ \sum_{i=0}^{n-1}(i+1)$

Soit environ $\frac{n(n+1)}{2}$ opérations

4.2 Complexité temporelle

On comprend avec les exemples précédents que calculer le coût d'un algorithme peut vite devenir fastidieux voire impossible. De plus, il n'est souvent pas utile de connaître exactement le coût de l'algorithme : ce qui intéresse l'informaticien c'est le **comportement de cet algorithme** avec le nombre et la taille des données.

On recherche donc la fonction mathématique $f(n)$ traduisant au mieux le **comportement asymptotique de l'algorithme**, c'est-à-dire lorsque le nombre et la taille des données deviennent importants.

Comme là encore il n'est pas toujours simple de trouver cette fonction on se ramène très souvent à étudier 3 cas :

- le cas le plus favorable (**complexité dans le meilleur** des cas, souvent peu utile)
- le pire des cas (**complexité dans le pire cas**, très souvent celle que l'on étudie)
- le cas moyen (**complexité en moyenne**, parfois délicate à gérer et qui relève plus du travail du mathématicien)

Dans l'étude du pire cas, le problème revient à chercher une fonction $f(n)$ majorant le coût $C(n)$ de l'algorithme.
On pourra dire que la complexité d'un algorithme est en $f(n)$ lorsque son coût $C(n)$ est borné par $f(n)$, on note alors $C(n) = O\big(f(n)\big)$ (se lit : « grand O de $f(n)$ »). L'étude de la complexité d'un algorithme dans le pire cas revient alors à chercher la meilleure fonction $f(n)$ traduisant le comportement asymptotique de l'algorithme.

Ordres de grandeurs des temps d'exécution avec un processeur effectuant 1 milliard d'opérations par seconde et des données de taille $n = 10^6$ (Python en réalise en fait beaucoup moins, plus proche de la dizaine de millions/s) :

Complexités	**Temps indicatif**	**Quelques exemples typiques**
coût constant en $O(1)$	$1\ ns$	● Echanger 2 éléments d'une liste ● Script qui ne dépend pas de la taille des données
coût logarithmique en $O(ln\ (n))$	$15\ ns$	● Recherches par dichotomie
coût linéaire en $O(n)$	$1\ ms$	● Recherche séquentielle ● Recherche d'un maximum/minimum
coût quasi-linéaire en $O(n\ ln(n))$	$15\ ms$	● Algorithmes de tris efficaces
coût quadratique en $O(n^2)$	$15\ min$	● Recherche dans une matrice, calcul matriciel ● Algorithmes de tris naïfs
coût cubique en $O(n^3)$	$30\ ans$	
coût polynomial en $O(n^\alpha)\quad \alpha > 3$	$30\ millions\ d'années$	
coût exponentiel en $O(x^n)\quad x > 1$	$plus\ de\ 10^{300000}$ milliards d'années	● Résolution naïve du problème du sac à dos
coût factoriel en $O(n!)$		● Résolution naïve du problème du voyageur

Exemple III.2.1

```
def f(n):
    for i in range(n) :
        for j in range(i+1) :
            print("*",end="")
        print()
```

Environ $\frac{n(n+1)}{2}$ opérations

donc C(n)=O(n^2)

Exemple III.2.2

```
def f(x,n):
    r = 1
    while n > 0 :
        if n%2 != 0 :
            r = r * x
        x = x**2
        n = n//2
    return(r)
```

On a : $2^k \leq n_i \leq 2^k + 1$

Dans le pire cas, on a donc à chaque itération :

n_{i+1}=⌊n_i/2⌋ soit k+1 passages avec $k \leq \lfloor \frac{\ln n}{\ln 2} \rfloor \leq \frac{\ln n}{\ln 2}$

Donc C(n)=O(ln n)

4.3 Complexité spatiale

Le but de la complexité spatiale est de donner une estimation de la place en mémoire nécessaire pour qu'un algorithme donné renvoie un résultat. De la même manière que la complexité temporelle, elle s'exprime sous la forme d'un $O(f(n))$.

Nous ne détaillerons pas plus cette partie car elle n'a aucun intérêt tant que la quantité de mémoire vive nécessaire à la réalisation d'un programme ne dépasse pas la capacité de la machine.

Remarque :

Le problème peut se poser avec les structures de piles ou récursives qui peuvent être amenées à stocker en mémoire de grandes quantités de données. Dans les autres cas (dans les sujets de concours notamment), l'étude de cette efficacité est inutile ou permet juste de prouver que le programme est mal écrit.

Table des matières