

IX – Preuve et complexité d'un algorithme

1/ Preuve d'un algorithme

Lorsque l'on utilise une boucle TANT QUE, les mêmes instructions sont répétées un certain nombre de fois et le temps d'exécution peut être arbitrairement long, voire infini dans le pire des cas. On ne rencontre pas ce genre de problèmes avec les boucles FOR puisque le nombre de répétitions est fixée dans celles-ci.

Exemple : l'algorithme ci-dessous ne s'arrête pas, on n'atteindra jamais l'affichage final.

```
n ← 0
TANT QUE n ≥ 0
    n ← n + 1
Afficher "C'est fini !"
```

Deux questions légitimes se posent ainsi lorsque l'on emploie une boucle.

- ❶ **Terminaison** : la boucle se termine-t-elle ? À quelles conditions ?
- ❷ **Validité** : qu'effectue précisément la boucle ? Obtient-on le résultat escompté ?

1.1) Terminaison

Principe : dans une boucle, on cherche à identifier la présence d'une expression à valeurs entières strictement décroissante et minorée, appelée le **Variant de boucle**.

Proposition

Un variant de boucle ne prend qu'un nombre fini de valeurs, donc la boucle se termine.

Exemple : prenons l'algorithme permettant d'effectuer la division euclidienne d'un entier a par un entier b non nul en utilisant uniquement les opérations élémentaires.

Dresser son tableau d'exécution pour $a=15$ et $b=4$.

Le *Variant de Boucle* est ici le reste r , qui décroît strictement à chaque étape de b et qui est minoré par 0.

De ce fait, le nombre d'opérations est fini.

```
a, b : entiers
q ← 0
r ← a
TANT QUE r ≥ b
    r ← r-b
    q ← q+1
Afficher q, r
```

Remarque : si l'on a une variable à valeurs entières strictement croissante et majorée (comme un compteur), il suffit de considérer son opposé pour obtenir un variant de boucle.

1.2) Validité

Principe : lorsqu'une boucle se termine, il faut qu'elle ait réalisé la tâche voulue. Pour s'en assurer, on utilise un **Invariant de boucle**. C'est une proposition qui équivaut au résultat recherché en fin de boucle et qui vérifie les conditions suivantes :

- elle est vraie avant l'entrée dans la boucle ;
- si elle est vraie au début d'une itération, elle reste vraie à la fin de cette itération.

Sous ces conditions, elle est encore vraie en fin de boucle, ce qui prouve le résultat.

Exemple : reprenons l'exemple précédent. On voudrait avoir $a = bq + r$ en fin de boucle. Vérifions alors que la proposition $\mathcal{P} : \langle a = bq + r \rangle$ est bien un invariant de boucle.

Remarques :

- ❶ À la fin de l'algorithme, on obtient $a = bq + r$, avec $0 \leq r \leq b - 1$.
Par le plus grand des hasards, c'est exactement ce que l'on voulait !
- ❷ Le principe de récurrence permet de s'assurer que la proposition $\mathcal{P}$ demeure vraie tout au long de l'algorithme, et en particulier à la fin.

2/ Complexité d'un algorithme

2.1) Généralités

En général, plusieurs algorithmes différents permettent de résoudre un problème donné. Si le nombre d'opérations et la taille des données sont très importants, deux paramètres deviennent prépondérants :

- ❶ le temps d'exécution ou **complexité temporelle** ;
- ❷ l'occupation de la mémoire ou **complexité spatiale**.

L'efficacité de l'algorithme dépend alors de ces deux paramètres, puisque l'on veut qu'il soit à la fois rapide et peu gourmand en ressources. De nos jours, le coût de la mémoire est faible, si bien que l'on cherche à améliorer avant tout la complexité temporelle.

Définition La complexité d'un problème donné est, par convention, la complexité de l'algorithme le plus efficace permettant de le résoudre.

Les calculs de complexité vont ainsi nous permettre de choisir les algorithmes les plus adaptés aux situations rencontrées. Par contre, on ne va pas compter le temps d'exécution en secondes, car il dépend de la machine utilisée. On choisit d'utiliser à la place des unités de temps abstraites, correspondant aux nombres d'opérations élémentaires effectuées.

2.2) Calculs de complexité

Définition On appelle *opérations élémentaires* (abrégé en oe) :

- les opérations mathématiques élémentaires ;
- les opérations logiques (calculs sur les booléens, comparaisons, tests) ;
- les affectations de variables ;
- les incrémentations de compteurs ;
- les affichages et les retours de valeurs

Par convention, chacune de ces opérations consomme une unité de temps.

Remarque : on simplifie quelque peu le problème car en réalité, toutes ces opérations n'ont pas la même complexité. Ainsi, la multiplication de deux entiers fait intervenir plusieurs additions (autant que le nombre de bits du plus petit entier).

Exemple : considérons la fonction ci-contre.

- On initialise f et i : 2 oe
- Lors de chaque itération, on incrémente i et le compare à n , on calcule $f*i$ et l'affecte à f : 4 oe
- On renvoie f : 1 oe

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

Ainsi, le calcul de $n!$ avec cette fonction va nécessiter $4n + 3$ opérations élémentaires. On dira que la complexité a pour ordre de grandeur n , ce qui se note $\mathcal{O}(n)$.

Définition Le *paramètre de complexité* d'un algorithme est le paramètre qui fait varier son temps d'exécution.

Exemples : pour la fonction précédente, c'est n . Pour un algorithme manipulant les termes d'une liste, cela sera la taille de la liste.

En général, pour un même paramètre de complexité, le nombre d'opérations élémentaires peut dépendre de la valeur prise par les entrées.

Exemple : considérons la recherche d'un élément précis dans une liste de taille n .

- S'il n'y est pas : on passe en revue les n éléments.
- S'il est en k -ième position : on ne passe en revue que k éléments.

Par conséquent, on sera amenés à étudier les quantités suivantes :

- 1) la **complexité dans le pire des cas**, qui permet d'avoir une majoration du nombre d'oe.
- 2) la **complexité moyenne**, intéressante si l'on répète l'algorithme avec des entrées variées.

2.3) Différentes complexités

Considérons un algorithme de paramètre de complexité n . Voici les ordres de grandeur de complexité usuels :

$\mathcal{O}(1)$	temps constant
$\mathcal{O}(\ln n)$	temps logarithmique
$\mathcal{O}(n)$	temps linéaire
$\mathcal{O}(n \ln n)$	temps linéarithmique
$\mathcal{O}(n^p)$	temps polynomial
$\mathcal{O}(p^n)$	temps exponentiel

Proposition

Quand une boucle contient (au plus) un nombre fixé d'opérations élémentaires, sa complexité totale est en $\mathcal{O}(N)$, où N est le nombre de passages dans la boucle.

Remarque : on utilisera ce résultat presque systématiquement lors des calculs de complexité. L'enjeu sera alors de déterminer le nombre d'itérations des boucles, en particulier en présence de boucles imbriquées.

Exemples : on considère dans la suite des listes de nombre de longueur n .

❶ La recherche des extrema d'une liste est en $\mathcal{O}(n)$.

❷ La recherche « basique » d'un élément dans une liste est en $\mathcal{O}(n)$.

Pour ces deux algorithmes, on passe en revue les n termes de la liste les uns après les autres, ce qui produit n étapes contenant chacune un nombre fixé d'opérations.

❸ La recherche dichotomique dans une liste triée est en $\mathcal{O}(\ln n)$.

Cet algorithme consiste à couper la liste en 2 à chaque étape, en ne conservant qu'une des deux moitiés. On s'arrête quand il ne nous reste qu'une liste à un élément. De ce fait, le nombre d'étapes k vérifie $2^k \simeq n$ soit $k \simeq \frac{\ln n}{\ln 2}$.

❹ Le crible d'Eratosthène est en $\mathcal{O}(n \ln n)$

On prend les nombres premiers successifs (situés en tête de liste) en supprimant à chaque fois tous leurs multiples de la liste. Par exemple, lorsqu'on a affaire au nombre k , on supprime $\frac{n}{k}$

éléments. On peut majorer grossièrement la complexité par $\sum_{k=2}^n \frac{n}{k} = n \times \sum_{k=2}^n \frac{1}{k} \simeq n \ln n$.

Exercices

1 / Preuve d'un algorithme

Les algorithmes proposés dans cette section sont systématiquement basés sur des boucles TANT QUE, afin d'étudier leur terminaison et leur validité. Certains peuvent être réécrits avec des boucles POUR, ce qui est bien évidemment préférable « dans la vraie vie ».

Exercice 1.

- 1) Écrire une fonction qui prend en argument deux entiers a et b et qui renvoie – en utilisant uniquement des opérations élémentaires – le quotient et le reste de la division euclidienne de a par b , sous forme de couple.
- 2) Ajouter à un test permettant de savoir et d'afficher si l'invariant de boucle est vérifié à chaque itération.

Exercice 2.

Pour calculer le terme d'indice N d'une suite $(u_n)_{n \in \mathbb{N}}$ définie par $u_0 = a$ et $u_{n+1} = f(n, u_n)$, on utilise l'algorithme ci-contre.

- 1) Justifier qu'il se termine en exhibant un **variant de boucle**.
- 2) Justifier qu'il se termine bien en prouvant que « $u = u_i$ » est un **invariant de boucle**.
- 3) Appliquer ceci au calcul de $N!$

```
i ← 0
u ← a
TANT QUE i < N
    u ← f(i,u)
    i ← i+1
Afficher u
```

Exercice 3.

Soient $a \in \mathbb{R}$ et $n \in \mathbb{N}$. L'algorithme ci-contre permet d'accélérer le calcul de a^n (pour les curieux, il est basé sur la décomposition de n en base 2).

- 1) Appliquer cet algorithme pour $a = 2$ et $n = 13$. On dressera un tableau d'exécution permettant de voir le nombre d'étapes nécessaires.
- 2) Trouver un **variant de boucle**.
- 3) Vérifier que la proposition « $s^N \times r = a^n$ » est un **invariant de boucle**.

```
s ← a
N ← n
r ← 1
TANT QUE N > 0
    SI N est pair ALORS
        s ← s2
        N ← N/2
    SINON
        r ← rs
        N ← N - 1
Afficher r
```

Exercice 4.

L'algorithme ci-contre permet de trouver le minimum d'une liste de valeurs réelles L.

- 1) Exécuter cet algorithme pour $L=[5,8,1,2]$.
On dressera un tableau d'exécution
- 2) Trouver un **variant de boucle**.
- 3) Trouver un **invariant de boucle**.

Indication : lors de l'exécution de l'algorithme, à la fin de l'étape i , que représente m en fonction de i ?

```

n ← len(L)
i ← 0
m ← L[0]
TANT QUE i < n-1
    i ← i+1
    SI L[i] < m ALORS
        m ← L[i]
Afficher m

```

Exercice 5.

L'algorithme ci-contre permet de trouver l'écriture en base 2 d'un entier décimal N.

- 1) Exécuter cet algorithme pour $n=15$.
On dressera un tableau d'exécution
- 2) Trouver un **variant de boucle**.
- 3) Montrer que « $N = n \times 2^{\text{len}(ch)} + \text{int}('0b'+ch)$ » est un **invariant de boucle**.

```

ch ← ''
n ← N
TANT QUE n > 0
    ch ← str(n%2) + ch
    n ← n // 2
Afficher ch

```

2/ Complexité d'un algorithme

Exercice 6.

Montrer les complexités des algorithmes des exercices 1,2 et 4 sont linéaires.

Exercice 7.

- 1) Montrer que les complexités des algorithmes des exercices 3 et 5 sont logarithmiques.
- 2) Quelle est la complexité du calcul « naïf » de a^n ? Commentaires ?

Exercice 8.

Une fonction trie les éléments d'une liste de longueur n et les range dans une nouvelle liste dans l'ordre croissant.

- 1) Quelle est sa complexité ?
- 2) Comment pourrait-on créer cette fonction, en s'appuyant sur un algorithme rencontré précédemment ?