
MPSI - Informatique Tronc Commun

Cours 6 : Comment étudier un algorithme ?

I Généralités

1 - Définition (Problème algorithmique) Un problème algorithmique consiste en

- La description des **entrées** du problème
- La description des **sorties** attendues

Les hypothèses que l'on fait sur les entrées sont appelées des **pré-conditions** et les propriétés qu'on demande sur la sortie du problème sont des **post-conditions**.

Exemples de problèmes algorithmiques

2 - Exemple (Le problème du tri)

Entrée : Une liste l .

Sortie : Une permutation triée de l .

3 - Exemple (Le problème de la parité)

Entrée : Un entier n positif.

Sortie : 0 si n est pair, et 1 si n est impair.

4 - Définition (Commentaires de documentation) Au tout début d'une fonction, on peut utiliser une chaîne de caractères entre 3 doubles guillemets pour préciser quel est le problème algorithmique résolu par la fonction. Cela s'appelle un **commentaire de documentation**.

5 - Définition (Algorithme) On appelle **algorithme** toute description précise d'étapes de calcul à suivre pour résoudre un problème algorithmique.

6 - Remarque Pour un problème algorithmique donné, il peut y avoir *plusieurs algorithmes différents* qui le résolvent.

7 - Les questions à se poser sur n'importe quel algorithme Quand on étudie un algorithme, il y a des questions qu'il convient toujours de se poser :

1. **Est-ce que l'algorithme termine ?** C'est à dire : Est-ce que l'exécution s'arrête, quelle que soit l'entrée ? On parle de **terminaison** de l'algorithme.
2. **Est-ce que l'algorithme est correct ?** C'est à dire : Est-ce que l'algorithme répond bien au problème posé ? On parle de **correction** de l'algorithme.
3. **Quel est le coût de l'algorithme ?**

Il a différents coûts auxquels on pourrait s'intéresser :

- Temps d'exécution
- Espace mémoire utilisé

- Consommation électrique
- Réchauffement de l'ordinateur

mais en ITC nous ne regarderons, en général, que le temps d'exécution.

Le temps d'exécution ne sera pas le même sur toutes les entrées, donc quand on étudie le temps d'exécution d'un algorithme, on pourrait s'intéresser à plusieurs choses :

- Quel est le temps *maximal* pris par l'algorithme ?
- Quel est le temps *moyen* pris par l'algorithme ?

En ITC, nous ne regarderons, en général, que le temps maximal d'exécution. En plus, nous ne regarderons pas le temps exact (qui peut dépendre de beaucoup de paramètres difficiles à contrôler, tels que "Quels autres programmes s'exécutent au même temps sur l'ordinateur ?"). À la place, nous choisirons certaines opérations, et nous regarderont, l'*ordre de grandeur* du nombre de fois que ces opérations ont lieu, en *fonction de la taille de l'entrée*. On parle alors de **complexité temporelle asymptotique dans le cas le pire** de l'algorithme. Souvent on dira juste **complexité** au lieu de *complexité temporelle asymptotique dans le cas le pire*.

II Terminaison des algorithmes

8 - Exemple de fonction qui ne termine pas

```
def affiche_nombres() :  
    n = 0  
    while True:  
        print(n)  
        n = n + 1
```

9 - Remarque Avec les fonctionnalités de Python qu'on a vu jusqu'ici, la seule raison pour laquelle l'exécution d'un programme pourrait ne pas terminer, est s'il contient une boucle **while** qui ne termine pas. Pour montrer qu'un programme termine, il convient alors de montrer que *chacune* de ses boucles **while** termine. Pour montrer qu'une boucle **while** termine, la méthode au programme est d'exhiber un **variant**.

Un **variant** est une quantité entière, qui change à chaque itération de la boucle, et qui est

- soit décroissante et minorée
- soit croissante et majorée

10 - Exemple n°1 de variant Soit la fonction suivante, qui résout le problème algorithmique de la parité :

```
def parite_inefficace(n) :  
    while n >= 2:  
        n = n - 2  
    return n
```

Pour cette fonction, n est un variant (strictement décroissant à chaque itération de la boucle, et minoré par 2), et permet donc de prouver que la boucle termine. En effet, si on pose n_i = valeur de n au début de la $i^{\text{ème}}$ itération de la boucle, la suite des n_i est une suite strictement décroissante d'entiers plus grands que 2. Hors il n'y a qu'un nombre fini de tels entiers, donc cette suite est finie, donc la boucle s'arrête à un moment donné.

11 - Exemple de variant n°2 On considère maintenant le problème du logarithme entier suivant :

Entrée : Un nombre x strictement supérieur à 1.

Sortie : Le plus petit entier n tel que $2^n \geq x$.

Une fonction qui résout ce problème algorithmique est la suivante :

```
def log_entier_inefficace(x):
    n = 0
    while (2**n < x):
        n = n + 1
    return n
```

Pour prouver que la boucle **while** de cette fonction termine, on peut utiliser le variant 2^n , qui est strictement croissant d'une itération de la boucle sur l'autre, et qui est majoré par $2x$.

12 - Remarque : on peut coder plus efficacement les exemples Bien sûr, ces exemples ont été choisis pour illustrer de façon simple la notion de variant, mais ils ne correspondent pas, en l'état, à des fonctions qu'on écrirait vraiment en Python.

En effet, les deux fonctions suivantes résolvent les mêmes problèmes algorithmiques, de façon plus efficace :

```
def parite(n) :
    return n%2

from math import log, ceil

def log_entier(x):
    resultat_a_virgule = log(x)/log(2)
    return ceil(resultat_a_virgule)
```

III Correction des algorithmes

13 - Définition (Propriété invariante / Invariant) Quand on veut prouver qu'un programme est correct, et que le programme contient des boucles (que ce soit des boucles **for** ou des boucles **while**), on utilise une **propriété invariante** de la boucle. Une **propriété invariante** d'une boucle est une propriété qui est vraie à chaque itération de la boucle. Une propriété invariante est parfois aussi appelée un **invariant**.

14 - Exemple de propriété invariante Considérons la fonction suivante, qui cherche le maximum dans une liste d'entiers non-vide.

```
def recherche_max(liste):
    """
    Entrée
    -----
    Une liste non-vide d'entiers
    Sortie
    -----
    Le plus grand élément de la liste
    """
    candidat = liste[0]
    for i in range(len(liste)):
        if liste[i] > candidat:
            candidat = liste[i]
    return candidat
```

Indiquons les différentes itérations de la boucle à partir de 0. Appelons candidat_i la valeur de la variable candidat à la fin de l'itération d'indice i . La propriété qui est invariante par exécution de la boucle, et qui va nous permettre de prouver la correction de la fonction est :

« Il existe un indice $j \in \llbracket 0, i \rrbracket$ tel que $\text{candidat}_i = \text{liste}[j]$;
et pour tout indice $k \in \llbracket 0, i \rrbracket$, $\text{candidat}_i \geq \text{liste}[k]$ ».

Dit autrement, après chaque itération, la variable candidat contient l'un des éléments de la partie déjà parcourue de la liste, qui est plus grand que tous les éléments de la partie déjà parcourue de la liste.

Prouvons cette propriété invariante par récurrence finie sur i .

Initialisation : À la fin de la 0^{ième} itération de la boucle, candidat contient $\text{liste}[0]$ qui est bien un élément de la liste (on prend $j = 0$). De plus, le seul k dans $\llbracket 0, 0 \rrbracket$ est 0 et on a bien $\text{candidat}_0 \geq \text{liste}[0]$ car $\text{candidat}_0 = \text{liste}[0]$.

Hérédité : Supposons la propriété vraie à la fin de la $i^{\text{ème}}$ itération et montrons-là à la fin de la $(i + 1)^{\text{ème}}$ itération. On distingue deux cas :

- Si $\text{candidat}_i \geq \text{liste}[i + 1]$ alors on a $\text{candidat}_{i+1} = \text{candidat}_i$. Par hypothèse de récurrence, il existe un indice $j \in \llbracket 0, i \rrbracket$ tel que $\text{candidat}_i = \text{liste}[j]$. Or $\llbracket 0, i \rrbracket \subset \llbracket 0, i + 1 \rrbracket$ donc on peut garder le même j . De plus, par hypothèse de récurrence on a pour tout indice $k \in \llbracket 0, i \rrbracket$, $\text{candidat}_i \geq \text{liste}[k]$ et par hypothèse de cas on a $\text{candidat}_i \geq \text{liste}[i + 1]$, donc on a pour tout $k \in \llbracket 0, i + 1 \rrbracket$, $\text{candidat}_{i+1} \geq \text{liste}[k]$, ce qui conclut ce cas de l'hérédité.
- Si $\text{candidat}_i < \text{liste}[i + 1]$, alors on a $\text{candidat}_{i+1} = \text{liste}[i + 1]$. Pour le j on prend $i + 1$. Prouvons maintenant le "pour tout". Soit $k \in \llbracket 0, i + 1 \rrbracket$. Si $k < i + 1$, l'hypothèse de récurrence nous donne $\text{candidat}_i \geq \text{liste}[k]$. Or, la condition du **if** nous garantit que $\text{candidat}_{i+1} \geq \text{candidat}_i$ donc $\text{candidat}_{i+1} \geq \text{liste}[k]$. Si $k = i + 1$, nous avons l'égalité, donc l'inégalité, ce qui conclut notre preuve par récurrence.

Correction de l'algorithme, en utilisant la propriété prouvée par récurrence Lorsque la boucle **for** a fini de s'exécuter, on a donc que la variable candidat contient l'un des éléments de la liste, et est plus grand que tous les éléments de la liste. C'est donc le maximum de la liste. $\square$

15 - Remarque : Formuler sa propriété invariante... au début, ou à la fin de la boucle ? Quand on pose un invariant, on doit choisir si on est en train de le formuler au début, ou à la fin du corps de la boucle ; ce qui change sera une question de décalage d'indice dans vos formules : i ou $i - 1$. Ici on a choisi d'exprimer notre invariant en fonction de l'état **à la fin de la boucle**. Cela permet d'avoir des formules mathématiques plus proches du code écrit.

16 - Remarque : « Préservation » au lieu de « Hérédité » Quand on prouve une propriété invariante d'une boucle, souvent on n'explicite pas qu'il s'agit d'une preuve par récurrence, et on parle parfois de **préservation** de la propriété invariante, au lieu de parler d'*hérédité* de la propriété prouvée par récurrence. Mais les choses à prouver restent les mêmes.