Chapitre 18

# Introduction à l'algorithmique

En général on écrit un programme informatique pour résoudre un problème (comment déterminer le plus court chemin pour aller d'une ville à une autre, rechercher l'adresse d'une personne dans un annuaire, etc.). Pour ces problèmes il existe souvent plusieurs solutions. La science algorithmique permet de tester et de classer ces solutions.

## 1. Notion d'algorithme

### 1.1 Définition

Un **algorithme** est une suite finie d'opérations élémentaires obéissant à un enchainement déterminé et produisant un résultat souhaité.

C'est une « recette » qui, si on la suit parfaitement, doit aboutir à chaque fois au résultat.

Bien sûr, les opérations élémentaires dont il est question sont des opérations réalisables par un processeur (ajout, suppression, comparaison, etc.).

*Exemple* : On souhaite trier un tableau de valeurs par ordre croissant. L'algorithme de tri est donc la succession d'opérations élémentaire qui va produire le résultat (le tableau trié).

*Remarque* Un même algorithme peut être implémenté dans différents langages. Une fois traduit dans le langage informatique utilisé, on obtient un **programme**.

En général un algorithme travaille sur des données d'entrée et donne des données de sortie.

*Exemple* : Pour l'algorithme de tri précédent, la donnée d'entrée est un tableau non trié et la donnée de sortie le même tableau trié.

## 1.2 Propriétés et caractéristiques d'un algorithme

Un bon algorithme doit posséder deux propriétés essentielles :

- Il doit **terminer** : il ne doit donc pas « tourner en rond » ou continuer indéfiniment ; il doit fournir le résultat en un nombre **fini** d'opérations.
- Il doit être **correct** : c'est-à-dire qu'il doit produire le résultat attendu dans toutes les situations.

Enfin il possède deux caractéristiques :

- Son **coût en temps** : le nombre d'opérations nécessaires à son exécution (où la durée de son exécution).
- Son **coût en espace :** la quantité d'espace mémoire nécessaire à son exécution.

## 1.3 Coût en temps d'un algorithme

Dans le programme de spécialité, on s'intéresse uniquement au coût en temps.

En général, on ne calcule pas le coût exact d'un algorithme mais son ordre de grandeur ; on classe alors les différents types de coûts par familles.

Si on note n la taille des données d'entrée, le coût de l'algorithme est une fonction de n, noté O(f(n)).

***Remarque*** Pour comprendre la notion de complexité, on peut prendre l'exemple du tri d'un tableau. Si un algorithme met un temps donné pour trier un tableau de taille n, combien mettra-t-il de temps pour trier un tableau 10 fois plus grand ? 1000 fois plus grand ?

La réponse à cette question dépend de la complexité de l'algorithme.

Exemple : Voici quelques types de complexité ainsi que des ordres de grandeurs de temps d'exécution en fonction de n.

| Type de complexité | Notation | Temps pour n = 10 | Temps pour n = 1000 | Temps pour $n = 10^6$ |
|---|---|---|---|---|
| Constante | O(1) | 10 ns | 10 ns | 10 ns |
| Logarithmique | O(log(n)) | 10 ns | 30 ns | 60 ns |
| Linéaire | O(n) | 100 ns | 10 µs | 10 ms |
| quadratique | $O(n^2)$ | 1 µs | 10 ms | 2,8 heures |

# 2. Parcours séquentiel d'un tableau

Dans ce paragraphe, on s'intéresse à un tableau noté tab et contenant n éléments notés tab[i] pour $0 \leq i < n$.

On suppose que les éléments du tableau sont des nombres.

## 2.1 Recherche d'une occurrence

Dans cette partie, on cherche si le tableau contient un élément noté b.

### Solution

Le programme suivant, écrit en Python, répond à cette question ; il renvoie « True » si b est dans le tableau et « False » sinon.

```
1. def recherche_occurrence(tab,b) :
2.     p=False
3.     for i in range(len(tab)):
4.        if tab[i]==b:
5.           p=True
6.           return p
7.     return p
```

### Coût de l'algorithme

Le programme effectue n+1 affectations (une avant la boucle et n dans la boucle). Le coût est donc **linéaire** (proportionnel à n, c'est-à-dire à la taille du tableau) et noté O(n).

## 2.2 Recherche d'un extremum

On cherche maintenant un extremum du tableau ; par exemple la plus grande valeur du tableau.

### Solution

On écrit donc une fonction qui prend en paramètre un tableau, et renvoie la plus grande valeur contenue dans ce tableau.

```
1. def recherche_maximum(tab):
2.     max = tab[0]
3.     for i in range(len(tab)):
4.        if tab[i]>max:
5.           max=tab[i]
6.     return max
```

### Coût de l'algorithme

Le programme ici encore effectue n+1 affectations (une avant la boucle et n dans la boucle). Le coût est donc **linéaire** et noté O(n).

## 2.3 Calcul d'une moyenne

Pour terminer ce paragraphe, on calcule la moyenne des valeurs du tableau.

### Solution

On écrit donc une fonction qui prend en paramètre un tableau de valeurs, et renvoie la moyenne de ces valeurs.

```
1. def calcul_moyenne(tab):
2.     m = 0
3.     for i in range(len(tab)):
4.         m = m + tab[i]
5.     moyenne = m/len(tab)
6.     return moyenne
```

### Coût de l'algorithme

Le programme ici encore effectue n+1 affectations (une avant la boucle et n dans la boucle). Il effectue également n opérations arithmétiques Le coût est donc **linéaire** et noté O(n).

## 3. Deux algorithmes de tri d'un tableau

### 3.1 Tri par insertion

On se propose d'effectuer un tri croissant par insertion sur le tableau tab = [9,8,5,4,7,6].

Le tri par insertion est le plus naturel : il consiste à parcourir le tableau de la gauche vers la droite, et pour chaque élément, le classer dans la partie du tableau située à sa gauche.

### Algorithme

Voici un exemple d'algorithme de tri par insertion :

```
1. def tri_insertion(tab):
2.     for i in range(1,len(tab)):
3.         cle=tab[i]
4.         j=i-1
5.         while j >= 0 and tab[j] >cle:
6.             tab[j+1] = tab[j]
7.             j = j - 1
8.         tab[j+1]=cle
```

On peut résumer dans un tableau ce que fait l'algorithme pour chaque itération de la boucle for :

| Valeur de i | Tableau avant la boucle | Valeur de la clé | Tableau en fin de boucle |
|---|---|---|---|
| 1 | [9, 8, 5, 4, 7, 6] | 8 | [8, 9, 5, 4, 7, 6] |
| 2 | [8, 9, 5, 4, 7, 6] | 5 | [5, 8, 9, 4, 7, 6] |
| 3 | [5, 8, 9, 4, 7, 6] | 4 | [4, 5, 8, 9, 7, 6] |
| 4 | [4, 5, 8, 9, 7, 6] | 7 | [4, 5, 7, 8, 9, 6] |
| 5 | [4, 5, 7, 8, 9, 6] | 6 | [4, 5, 6, 7, 8, 9] |

***Remarque*** À la fin de chaque boucle, le tableau est trié de la case n° 0 à la case n° i (valeurs en rouge).

### Preuve de correction

La propriété p(i) : « Le tableau est trié jusqu'à la case n° 1 » est ce qu'on appelle un **invariant de boucle**. Cette propriété est vraie avant et après chaque itération.

Elle prouve que l'algorithme est correct car P(n-1) est vraie (ce qui correspond au tableau trié).

### Coût

Dans le pire des cas, (tableau initial trié par ordre décroissant), la boucle while effectue 2n opération. Chaque itération de la boucle for compte donc $2n + 3$ opérations, répétées $n - 1$ fois. Le coût est donc de $(n - 1)(2n + 3)$ soit de l'ordre de $n^2$ (noté $O(n^2)$). On dit que le coût est **quadratique** dans le pire des cas.

## 3.2 Tri par sélection

On trie à présent le même tableau, en utilisant une autre méthode, appelée tri par sélection. Il consiste à :

- rechercher le plus petit élément du tableau et l'échanger avec celui d'indice 0,
- rechercher le plus petit élément du tableau restant et l'échanger avec celui d'indice 1,
- continuer ainsi pour tous les éléments.

### Algorithme

Voici un algorithme de tri par sélection écrit en Python :

```
1. def tri_selection(tab):
2.     for i in range(0,len(tab)-1):
3.         mini = i
4.         for j in range(i+1, len(tab)):
5.             if tab[j] < tab[mini]:
6.                 mini = j
7.         temp = tab[i]
8.         tab[i] = tab[mini]
9.         tab[mini] = temp
```

Et voici un tableau qui résume ce que fait l'algorithme à chaque itération :

| Valeur de i | Tableau avant la boucle | Tableau en fin de boucle |
|---|---|---|
| 0 | [9, 8, 5, 4, 7, 6] | [4, 8, 5, 9, 7, 6] |
| 1 | [4, 8, 5, 9, 7, 6] | [4, 5, 8, 9, 7, 6] |
| 2 | [4, 5, 8, 9, 7, 6] | [4, 5, 6, 9, 7, 8] |
| 3 | [4, 5, 6, 9, 7, 8] | [4, 5, 6, 7, 9, 8] |
| 4 | [4, 5, 6, 7, 9, 8] | [4, 5, 6, 7, 8, 9] |

***Remarque*** À la fin de chaque boucle, le tableau est trié de la case n° 0 à la case n° i (valeurs en rouge).

### Preuve de correction

La propriété p(i) : « Le tableau est trié jusqu'à la case n° i – 1 » est un **invariant de boucle**. Cette propriété est vraie avant et après chaque itération.

P(n – 1) est vraie (ce qui correspond aux n – 1 premières valeurs triées). La dernière valeur est forcément la plus grande de toutes, donc le tableau est trié à la fin de la dernière boucle.

### Coût

La première boucle s'effectuer n – 2 fois et la seconde n – 1 – i fois.

Le coût est donc proportionnel à $n^2$, donc noté $O(n^2)$, soit un coût **quadratique**.

### *Compétences à acquérir*

- Écrire un algorithme de recherche d'une occurrence sur des valeurs de type quelconque.
- Écrire un algorithme de recherche d'un maximum, de calcul d'une moyenne.
- Écrire un algorithme de tri, par sélection ou insertion et savoir prouver leur correction.
- Estimer le coût d'un algorithme.

# Fiche méthodes

## Méthode 18.1

## Écrire un programme en langage naturel

Pour écrire un algorithme on utilise le langage naturel (« tant que », »si », etc.) qui permette ensuite de passer à un langage de programmation.

***Exemple*** : Écrire le programme Python qui correspond à l'algorithme suivant.

Variables :

- t : tableau d'entiers
- x : entier recherché dans t
- p : variable booléenne
- i : entier

```
Début
p ← Faux
I ← 1
tant que i ⇐ longueur(t) et p == Faux :
    si t[i] == x
    p ← Vrai
    fin si
    i ← i + 1
fin tant que
renvoyer la valeur de p
Fin
```

***Réponse*** :

```
1. def recherche_occurrence(t,x):
2.    p=False
3.    while i <= len(t) and p == False:
4.       if tab[i]==x:
5.          p=True
6.       i = i + 1
7.    return p
```

## Méthode 18.2

# Évaluer le coût d'un algorithme

Pour évaluer le coût d'un algorithme, il faut compter le nombre d'opérations élémentaire (affectations et opérations) effectuées et l'exprimer en fonction de la taille des données n.

Ensuite on examine la dépendance du coût à n.

- Si le coût est indépendant de n, il est constant.
- Si le coût est proportionnel à n, il est linéaire.
- Si le coût est proportionnel à $n^2$, il est quadratique.

***Exemple*** : Soit la fonction suivante qui calcule la somme des éléments numériques d'un tableau. Quel est sont coût ?

```
1. def somme_tableau(tab):
2.     s = 0
3.     for i in range(len(tab)):
4.         s = s + tab[i]
5.     return s
```

***Réponse*** : Il y a une boucle qui s'effectue n fois si le tableau contient n éléments. Le coût est proportionnel à n, donc linéaire.

## Méthode 18.3

# Prouver la correction d'un algorithme

La preuve de la correction d'un algorithme qui effectue une ou plusieurs boucles s'obtient en cherchant un **invariant de boucle**, c'est-à-dire une propriété qui est vraie avant et après l'exécution d'une boucle.

On la note en général P(i).

Si P(i) est vraie alors P(n) doit être vraie (n correspondant à la dernière boucle).

***Exemple*** : On considère la fonction définie à la méthode 18.2. Prouver la correction de ce code.

***Réponse*** : À la fin de la boucle n° i, la variable s contient la somme des i premiers termes du tableau. La propriété P(i) : « s contient la somme des termes du tableau d'indices inférieurs à i » est donc vraie.

À la fin de la boucle suivante, on a ajouté à s la valeur du terme suivant. La propriété P(i) reste donc vraie : c'est un invariant de boucle.

On peut donc conclure que P(n – 1) est vraie et que l'algorithme calcule bien la somme de tous les termes du tableau.