L15 : Complexité d’un algorithme
Parcours d’un tableau

Un **algorithme** est une suite finie et claire d’opérations ou d'instructions permettant de résoudre une classe de problèmes. Un algorithme est donc écrit à partir de données et d’opérations qui s’enchaînent grâce à des structures de contrôle (boucles et conditions).

Historique : Le mot **algorithme** vient du nom d'un mathématicien perse **Al Khârizmi**, né en 780 et mort en 850 (IX^{e} siècle).

Lorsqu’on rédige une lettre, un article, une rédaction,..., etc., on vérifie l’orthographe, la conjugaison et la grammaire de ce texte. De la même manière, lorsqu’on a écrit un algorithme, on doit ensuite prendre la précaution de vérifier trois points :

- la . . . . : être sûr que cet algorithme se termine ;
- la . . . . : être sûr que cet algorithme donne la solution au problème posé ;
- la . . . .

1. Complexité d’un programme

Le calcul de la complexité d'un algorithme permet de mesurer sa performance. Il existe deux types de complexité :

- . . . . : permet de quantifier l'utilisation de la mémoire ;
- . . . . : permet de quantifier la vitesse d’exécution.

Dans ce cours, on s’intéresse à la complexité temporelle.

La complexité temporelle **mesure l’efficacité d’un programme en comptant le nombre d’opérations élémentaires** (affectations, calculs arithmétiques ou logiques, comparaisons …) à effectuer tout au long de cet algorithme.

On parle aussi de **coût en temps** de l’algorithme.

Remarque : cette efficacité est mesurée sur un algorithme et est donc indépendante de la puissance de l’ordinateur et du langage dans lequel cet algorithme est programmé.

Exemple : on considère l’algorithme suivant :

```
Entrée : un entier n
Traitement :  nombre ← 1
              pour i variant de 1 à n+1
                  nombre ← nombre * i
Sortie : nombre
```

On considère n = 5.

1. Qu'obtient-on à la sortie de cet algorithme ? . . . .

2. Quel est le nombre d'opérations effectuées par l'algorithme ?

ligne 1	. . . . affectation(s)
ligne 2	. . . .
lignes 3 - 4	répétitions . . . . fois de : . . . . . . opération(s) . . . . . . affectation(s)
ligne 5	. . . . affichage(s)

Total : . . . .

. . . .

On appelle **opérations élémentaires (ou opérations de base)** les instructions suivantes :

- affectation ;
- test de comparaison (`==` ; `<` ; `<=` …)
- opération de lecture (`input`) et écriture (`print`)
- opération arithmétique (y; `-`; `*`; `/`; `%`; `//`)

Le coût d'une opération élémentaire est égale à 1.

On appelle **opérations composées** les instructions suivantes :

- instruction conditionnelle **si Test alors C_1 sinon C_2**

  dans ce cas : Coût <= Coût du test + max(coût de C_1 et coût de C_2)

- Boucle **for** ou **while** :

  dans ce cas : Coût = nombre de répétitions □ coût du corps de la boucle

- **Appel d'une fonction** :

  dans ce cas : Coût = nombre d'opérations élémentaires engendrées par l'appel de cette fonction

La complexité en temps d'un algorithme sera exprimé par une **fonction, notée** □.

On notera ***n*** la taille d'une donnée. En général, on ne donne pas le nombre exact d'opérations élémentaires effectuées mais un ordre de grandeur de ce nombre d'opérations, que l'on notera ***O*(*n*)**. Selon cet ordre de grandeur, on définit plusieurs types de complexités :

- complexité de type **constant** :

  le nombre d'opérations à effectuer ne change pas lorsque le paramètre varie.

  L'ordre de grandeur est ***O*(1)**.
- complexité de type **linéaire :**

  le nombre d'opérations à effectuer $T(n)$ est de la forme $T(n) = a\,n + b$

  L'ordre de grandeur est ***O*(*n*)**.
- complexité de type **quadratique**

  le nombre d'opérations à effectuer $T(n)$ est de la forme $T(n) = a\,n^2 + b\,n + c$

  L'ordre de grandeur est $\boldsymbol{O(n^2)}$.

En général on s'intéresse à la performance de l'algorithme dans les situations où le problème prend le plus de temps. On parle alors de *complexité dans le pire cas*.

Exemple 1 : on considère l'algorithme suivant :

```
si n % 2 == 0 :
    n = n//2
sinon :
    n = n + 1
    n = n//2
```

Quel est le nombre d'opérations effectuées par l'algorithme ?

. . . .

. . . .

. . . .

. . . .

Exemple 2 : on considère l'algorithme suivant :

```
tant que i <= n
    somme = somme + i
    i = i + 1
```

Quel est le nombre d'opérations effectuées par l'algorithme ?

. . . .

. . . .

. . . .

. . . .

Exemple 3 :

On considère l'algorithme ci-dessous :

```
1   Entrées : une liste L[1..n] et un élément a
2   Traitement :  i ← 1
3                 tant que i < n et L[i] ≠ a faire
4                    i ← i + 1
5                 fin
6                 si i<n alors
7                    retourner Vrai
8                 sinon
9                    retourner Faux
10                fin
11   Sorties : un booléen
```

On définit L = [5, 1, 2, 4, 6, 7].

1. Qu'obtient-on à la sortie de cet algorithme ? . . . .

2.

a. Quel est le nombre d'opérations effectuées par l'algorithme de recherche avec les entrées L et 5 ?

ligne 1	. . . . affectation(s)
ligne 2	
ligne 3 à 10	répétitions au plus . . . . fois de : . . . . . . comparaison(s) . . . . . . affectation(s)
ligne 11	

b. En déduire la complexité de cet algorithme : . . . .

2. Parcours séquentiel d’un tableau

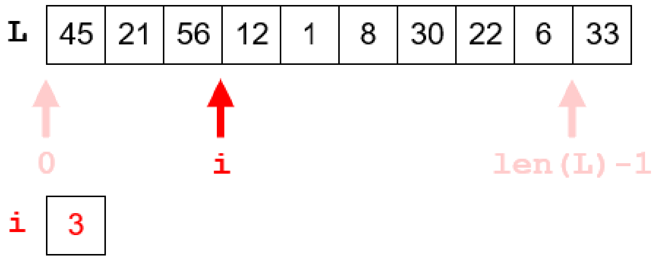
Pour la suite, nous utiliserons Python, et travaillerons sur une liste de nombres entiers générés aléatoirement :

```
from random import randint
L = [randint(1, 100) for i in range(10)]
```

En Python, les listes, les chaînes de caractères et les tuples sont des **itérateurs**. C’est-à-dire qu’ils contiennent dans leur structure les méthodes permettant de les parcourir.

a. Parcours par indice

On considère la liste L définie ci-dessous :



On peut parcourir un tableau en faisant évoluer un indice :

Programme 1 :

```
rang = 0                    # initialisation de rang
while rang < len(L):        # vérification
    print(L[rang])
    rang += 1               # incrémentation de rang
```

Programme 2 :

```
for rang in range(len(L)):
    print(L[rang])          # l’incrémentation de rang
                            se fait automatiquement
```

b. Parcours par valeur

En Python, on peut directement accéder à une valeur de la liste :

```
for valeur in L:
    print(valeur)
```

c. Parcours par indice et valeur

En Python, on peut également parcourir un liste afin d'avoir à la fois le rang et la valeur :

```
for rang,valeur in enumerate(L):
    print(rang,valeur)
```