Chapitre 1

Parcours d'une structure séquentielle

UN SCIENTIFIQUE

Au III^e siècle avant notre ère, Alexandrie était le centre intellectuel du Monde. Une immense bibliothèque contenait toutes les connaissances de l'époque. **Ératosthène** (env. 276 av. J.-C. - env. 194 av. J.-C.), son conservateur, brillait dans toutes les disciplines, en particulier la géographie et les mathématiques. Son calcul du périmètre de la Terre s'est révélé presque exact et son nom reste attaché à la méthode du crible pour rechercher les nombres premiers.

Un peu d'histoire

L'algorithme conçu par Ératosthène pour la recherche des nombres premiers, connu sous le nom de crible, est sans doute le plus ancien de forme séquentielle. Pour obtenir, par exemple, tous les nombres premiers inférieurs à 1000, on les écrit tous à partir de 2. On laisse 2 et on raye tous ses multiples ; le plus petit nombre non rayé est alors 3 ; on le laisse et on raye tous ses multiples. On prolonge cette opération jusqu'à atteindre la partie entière de la racine carrée de 1000, soit 31. Les nombres non rayés sont les nombres premiers.

Depuis, des algorithmes de recherches de nombres premiers se sont multipliés comme celui conçu par le mathématicien et informaticien Derrick Lehmer. Ces recherches ont longtemps été considérées comme des curiosités sans application pratique. L'utilisation d'immenses nombres premiers en cryptographie leur a donné une importance cruciale.

Les boucles permettent l'exploration des éléments des tableaux unidimensionnels. Elles sont présentes dans de nombreux algorithmes étudiés au lycée. Avec elles, il est possible de déterminer la présence d'un élément particulier, d'effectuer un calcul sur les éléments.

OBJECTIFS

- Connaître quelques algorithmes classiques utilisant un parcours séquentiel :
  - être capable de lire et de modifier des éléments d'un tableau ;
  - savoir calculer une moyenne ;
  - maîtriser la recherche d'un extrémum et d'une valeur particulière ;
  - comprendre comment compter des éléments vérifiant une propriété.
- Comprendre la distinction entre un coût constant et un coût linéaire :
  - savoir évaluer le nombre d'opérations effectuées dans un programme ;
  - comprendre l'importance de la notion de coût ou de complexité.
- Découvrir l'intérêt des dictionnaires en Python :
  - comprendre le rôle d'un dictionnaire ;
  - connaître l'apport en terme de coût dans l'efficacité d'un programme.

Cours

Boucles

Un programme est composé d'une suite d'instructions, exécutées l'une après l'autre dans l'ordre où elles sont écrites, contenant des définitions de variables et de fonctions, des affectations, des boucles, des instructions conditionnelles, qui utilisent des expressions pouvant être des résultats d'appels de fonctions.

On distingue deux types de boucles :

- les boucles conditionnelles :

```
while condition:
    instructions
```

- les boucles non conditionnelles :

```
for elt in sequence:
    instructions
```

Avec une boucle non conditionnelle, le bloc d'instructions est répété n fois, n étant la longueur de la séquence (une liste, un tuple, une chaîne de caractères). La variable `elt` prend successivement la valeur de l'un des n éléments de la séquence. Cette variable `elt` peut être utilisée dans le bloc d'instructions ou pas.

Exemple :

```
for i in range(n):
    instructions
```

Avec le code qui suit, le bloc `instructions` est exécuté de la même manière :

```
i = 0
while i < n:
    instructions
    i = i + 1
```

Une variable `i` est créée. Cette variable n'est pas détruite après l'exécution de la boucle. Avec la boucle `for`, elle prend successivement les valeurs 0, 1, ..., `n`−1, avec la boucle `while` elle prend les valeurs 0, 1, ..., `n`.

■ Outils

■ Compteurs

Lorsqu'on utilise une boucle `while` on peut souhaiter compter le nombre de passages dans la boucle. De manière générale, on peut avoir besoin de compter le nombre d'apparitions d'un certain fait. On utilise alors une variable que l'on peut appeler *compteur*. Cette variable est initialisée à 0 et peut être incrémentée d'une unité à chaque passage dans la boucle.

Exemple avec une boucle conditionnelle sans test

Le paramètre `n` est un entier naturel. On compte le nombre de divisions euclidiennes successives de `n` par 2, jusqu'à arriver à un quotient nul. On obtient donc le nombre de chiffres dans l'écriture binaire de `n` si `n` est non nul.

```
def taille(n):
    cpt = 0
    while n > 0:
        cpt = cpt + 1
        n = n // 2
    return cpt
```

Avec une boucle inconditionnelle et un test

Le paramètre `n` est un entier naturel non nul. Le compteur est incrémenté quand `n` est divisible par `d`. On compte donc le nombre de diviseurs de `n` qui est le résultat renvoyé par la fonction.

```
def diviseurs(n):
    cpt = 0
    for d in range(1, n + 1):
        if n % d == 0:
            cpt = cpt + 1
    return cpt
```

■ Accumulateurs

Un accumulateur est semblable à un compteur mais il peut être incrémenté d'une valeur différente de 1 ou décrémenté.

```
def somme_pairs(liste):
    acc = 0
    for x in liste:
        if x % 2 == 0:
            acc = acc + x
    return acc
```

La boucle contient un test. Nous supposons que `liste` est une liste de nombres. La fonction renvoie la somme des nombres pairs contenus dans la liste.

L'exemple suivant qui concerne le calcul d'une moyenne est sans test.

```
def moyenne(liste):
    acc = 0
    for x in liste:
        acc = acc + x
    return acc / len(liste)
```

La liste est supposée non vide. Si n est sa longueur, nous disons que *le coût est linéaire en n*, car nous opérons n additions et n affectations effectuées dans la boucle. Nous supposons, et c'est le cas, que l'obtention de la longueur par `len(liste)` a un coût constant (une seule opération).

■ Recherche de valeurs

Définition : une *occurrence* est l'apparition d'un fait, par exemple la présence d'un mot dans un texte. Lorsqu'on cherche dans un conteneur une occurrence d'un élément, on cherche si une place est occupée par cet élément dans le conteneur. Une place est représentée par son indice.

■ Recherche d'une occurrence

Il s'agit de rechercher de manière séquentielle la présence d'une valeur dans un tableau. Cela signifie que la valeur cherchée est successivement comparées à toutes les valeurs du tableau. Cette méthode est aussi appelée *méthode par balayage* ou *recherche linéaire* (en anglais, *linear search*). Un tableau peut être ici une liste, un p-uplet ou une chaîne de caractères. On cherche donc une valeur précise dans une liste ou un p-uplet ou un caractère précis dans une chaîne.

L'algorithme s'arrête dès que l'élément est trouvé ou si la fin du tableau est atteinte.

Un algorithme de recherche d'un élément `x` dans un tableau `t` de longueur `n` utilise, de manière générale, une boucle conditionnelle :

```
i = 0
tant que i < n et x différent de t[i]
    i = i + 1
fin tant que
si i < n
    renvoyer i
```

Cet algorithme est l'occasion d'aborder la notion de *coût*, notion qui est précisée au chapitre 10. Disons que *le coût en temps d'un algorithme est déterminé par le nombre d'opérations élémentaires, (affectations, comparaisons, opérations arithmétiques simples), exécutées par l'algorithme.*

Nous commençons par compter le nombre maximum de comparaisons effectuées.

Si l'élément recherché n'est pas dans le tableau, il est nécessaire de parcourir tout le tableau, donc d'effectuer n itérations, pour examiner chaque élément du tableau avec $2n$ comparaisons (comparaisons entre `i` et `n`, et entre `x` et `t[i]`. Nous avons alors pour la boucle un total de n affectations et n additions, donc de $4n$ opérations. Nous disons que *le coût est linéaire en n*.

Lorsqu'on parcourt une liste ou une chaîne, on parle de parcours séquentiel. Dans la notion de séquence nous avons la notion d'ordre. Les éléments ont chacun une place numérotée par un indice.

Nous pouvons envisager plusieurs variantes qu'il faut maîtriser : soit on souhaite simplement chercher si une valeur est présente, soit on cherche un indice d'occurrence, soit on cherche tous les indices d'occurrence. La suite propose quelques exemples.

- **Recherche de la présence d'une valeur avec une boucle `while`**

```
def recherche1(tab, val):
    presence = False
    i = 0
    while i < len(tab) and not presence:
        if tab[i] == val:
            presence = True
        i = i + 1
    return presence
```

Une deuxième forme :

```
def recherche2(tab, val):
    i = 0
    while i < len(tab)-1 and tab[i] != val:
        i = i + 1
    return tab[i] == val
```

Avec une boucle `for` et une sortie de boucle anticipée :

```
def recherche3(tab, val):
    for i in range(len(tab)):
        if tab[i] == val:
            return True
    return False
```

Une fonction semblable à la précédente mais qui n'utilise pas les indices :

```
def recherche4(tab, val):
    for elt in tab:
        if elt == val:
            return True
    return False
```

Avec la fonction qui suit, c'est Python qui fait tout le travail.

```
def recherche5(tab, val):
    return val in tab
```

- **Recherche de la première occurrence**
  Avec une boucle `while` :

```
def recherche6(tab, val):
    indice = 0
    while indice < len(tab) and tab[indice] != val:
        indice = indice + 1
    return indice # renvoie len(tab) si échec
```

Avec une boucle `for` et une sortie de boucle anticipée :

```
def recherche7(tab, val):
    for indice in range(len(tab)):
        if tab[indice] == val:
            return indice
    return len(tab) # ou indice + 1
```

- **Recherche de la dernière occurrence**
  Pour la recherche de la dernière occurrence, on parcourt tout le tableau avec une boucle `for` :

```
def recherche8(tab, val):
    indice = len(tab)
    for i in range(len(tab)):
        if tab[i] == val:
            indice = i
    return indice
```

On pourrait aussi chercher la première occurrence à partir de la fin. Avec une petite modification de la fonction `recherche7`, on obtient la fonction `recherche9` ci-dessous :

```
def recherche9(tab, val):
    for i in range(len(tab)):
        indice = len(tab) - 1 - i
        if tab[indice] == val:
            return indice
    return len(tab)
```

■ Recherche d'un extrémum

Nous disposons d'un ensemble de nombres dans lequel nous cherchons un extrémum. On peut chercher un maximum, un minimum, ou les deux.

- **Algorithme de recherche du maximum**

Si la liste est non vide, on suppose que le maximum est le premier élément, puis on parcourt la liste et chaque fois qu'on rencontre un élément plus grand que le maximum provisoire, on dit que c'est le nouveau maximum provisoire.

Nous procédons à la recherche du maximum à l'aide d'un parcours séquentiel dans une liste de nombres non vide.

```
def maximum(liste):
    maxi = liste[0]
    for i in range(1, len(liste)):
        if liste[i] > maxi:
            maxi = liste[i]
    return maxi
```

Évaluons le nombre de comparaisons et le nombre d'affectations effectuées afin d'obtenir le coût en temps de l'algorithme. Si n est la longueur de la liste, nous avons dans tous les cas $n-1$ comparaisons et au maximum, (nous disons *dans le cas le moins favorable* ou *dans le pire des cas*), n affectations, (une avant d'entrer dans la boucle), donc un total de $2n-1$ opérations. Nous disons que le coût ou la complexité de l'algorithme est linéaire en fonction de la taille de la liste.

Il est évident que la recherche d'un minimum dans une liste de nombres s'effectue de manière similaire. Il suffit de remplacer le signe `>` par le signe `<` dans la comparaison.

Un problème semblable est de chercher les deux plus grands éléments. On pourrait commencer par chercher le maximum avec $n-1$ comparaisons puis recommencer pour chercher le deuxième maximum avec $n-2$ comparaisons.

La méthode appliquée ci-dessous est différente, c'est celle qui est appliquée à la recherche d'un maximum : les deux premiers éléments constituent le couple cherché, puis on parcourt la liste pour remplacer éventuellement le plus petit de ces deux éléments. On suppose que la liste contient au moins deux éléments et que les éléments sont tous distincts.

```
def maxima2(liste):
    maxi1, maxi2 = liste[0], liste[1]
    if maxi1 < maxi2:
        maxi1, maxi2 = maxi2, maxi1
    for i in range(2, len(liste)):
        x = liste[i]
        if x < maxi1:
            if x > maxi2:
                maxi2 = x
        else:
            maxi1, maxi2 = x, maxi1
    return maxi1, maxi2
```