ITC – cours n°2
Algorithmes de parcours séquentiel

On présente dans ce cours la structure algorithmique de base pour interroger ou modifier une collection fournie, par exemple :
- quel est le majorant / minorant ?
- quelle est la position du majorant / minorant ?
- quelle est la somme, la moyenne, l'écart-type ?
- quel est le nombre d'occurrences d'une valeur donnée ?
- modifier toutes les valeurs paires ...

I. Premier exemple

On propose l'algorithme suivant pour chercher le minorant dans une liste :

```
def minimum(liste):
    # Détermine le minimum dans une liste de nombres de cardinal >= 1
    n = len(liste)
    xmin = liste[0]
    for i in range(n):  # on parcourt tous les indices valides
        # xmin est le minorant de la sous-liste liste[0:i]  (1)
        if liste[i] < xmin:  # on conserve la propriété (1)
            xmin = liste[i]
        # xmin est le minorant de la sous-liste liste[0:i+1]  (1)
    return xmin
```

On analyse cet algorithme de la façon suivante :
- l'élément principal est la boucle qui permet de parcourir l'ensemble des éléments de la liste : à chaque tour, on a accès à une variable i qui représente un indice valide de la liste, donc au i^{e} élément de la liste ; les opérations de cette boucle seront répétées pour chaque élément de la liste ;
  - dans la boucle, on s'assure que, si le nouvel élément `liste[i]` est plus petit que le minimum trouvé jusque-là $x_{\min}$, alors $x_{\min}$ deviendra égal à `liste[i]` : ainsi on s'assure que $x_{\min}$ reste le plus petit élément recontré jusqu'ici ;
- une fois tout le parcours fini, $x_{\min}$ est donc par construction le plus petit élément rencontré dans la liste ;
- la question de choisir une valeur initiale pour $x_{\min}$ doit être posée avant la boucle : ici, on choisit la première valeur disponible dans la liste, afin de s'assurer que c'est la plus petite rencontrée en début de parcours.

II. Algorithme de parcours

2.1. Structure générale

D'une manière générale, un algorithme de parcours séquentiel va suivre la même structure :

```
def parcours(liste):
    # initialisation
    n = len(liste)
    # ...
    for i in range(n):  # parcours des éléments
        # traitements à effectuer
    # fin de la boucle
    return résultat
```

- l'**initialisation** va préparer l'ensemble des variables utiles pour le parcours et le traitement nécessaire ;
- la **boucle de parcours** va permettre d'appliquer le même traitement à chaque élément de la collection, afin d'effectuer le traitement voulu ; en général, on propage une propriété qui reste vraie à travers toute la collection (dans le cas de l'exemple, la propriété que $x_{\min}$ reste l'élément le plus petit de la sous-liste `liste[0:i]`) ;
- une fois la collection parcourue, on peut construire (ou on a déjà) le résultat grâce à la propriété restée vraie sur la liste complète.

2.2. Choix du parcours : indices vs valeurs

La syntaxe python permet de parcourir directement les valeurs comprises dans la liste : par exemple, on peut refaire la fonction `minimum` de la façon suivante :

```
def minimum(liste):
    xmin = liste[0]
    for x in liste:  # on parcourt toutes les valeurs
        # xmin est le minorant de la sous-liste liste[0:i]  (1)
        if x < xmin:  # on conserve la propriété (1)
            xmin = x
        # xmin est le minorant de la sous-liste liste[0:i+1]  (1)
    return xmin
```

Ainsi, au tour de boucle numéro $i \in [\![0; n[\![$, x est une variable qui correspond à `liste[i]` . L'avantage est que la manipulation des variables est plus naturelle ; l'inconvénient est que **l'on n'a pas accès à l'indice correspondant à la valeur** x. Dans le cas de la recherche du minorant, cela ne pose pas de problème, mais si on veut l'indice correspondant, ou si on veut modifier la liste, cela est impossible. Prenons l'exemple d'une fonction qui multiplie les éléments par -1 :

```
def opposé(liste):
    n = len(liste)
    for i in range(n):
        liste[i] = -liste[i]
```

conduit au comportement attendu

```
In [1]: l_test = [2, 1, 4]
In [2]: opposé(l_test)
In [3]: l_test
Out[1]: [-2, -1, -4]
```

tandis que

```
def opposé(liste):
    for x in liste:
        x = -x
```

conduit à

```
In [1]: l_test = [2, 1, 4]
In [2]: opposé(l_test)
In [3]: l_test
Out[1]: [2, 1, 4]
```

car à chaque tour de boucle, x est une étiquette qui pointe sur la i^{e} valeur de la liste ; si on change x, on fait pointer **cette étiquette** vers une autre valeur, mais on n'a pas fait pointer **l'étiquette** `liste[i]` vers cette nouvelle valeur : la liste est inchangée.

Les deux méthodes de parcours sont donc à distinguer et à utiliser avec discernement.

2.3. Cas courant : accumulateurs

Lorsqu'on souhaite calculer une propriété de **l'ensemble** de la collection, on utilise souvent une variable appelée **accumulateur** dans laquelle est stockée un résultat partiel sur une sous-liste. Par exemple :

```
def somme(liste):
    val = 0  # accumulateur
    for x in liste:  # tou de boucle n°i
        # val vaut sum(liste[0:i])
        val += x
        # val vaut sum(liste[0:i+1])
    return val
```

on constate qu'à chaque tour de boucle, on **accumule** une somme partielle dans la variable `var` :

$$val = \sum_{j=0}^{i} \ell_j$$

ce qui garantit bien qu'elle contient la somme des valeurs de la liste une fois la boucle finie.

Les accumulateurs permettent souvent de calculer des grandeurs qui s'expriment mathématiquement avec les signes $\sum$ ou $\prod$.

III. Considérations plus avancées

3.1. Complexité : le « temps de calcul »

Une des questions les plus importantes en informatique est le temps de calcul pris par un algorithme. Il ne s'agit pas de déterminer le temps d'exécution en secondes de la fonction (cela dépend de la machine utilisée, du langage de programmation, de la taille des entrées...tout ceci n'est pas connu à ce stade), mais plutôt de déterminer **comment le temps de calcul évolue lorsque la taille des entrées devient très grande**. Pour ce faire, on va considérer que toutes les opérations suivantes coûtent un temps constant (indépendant de la taille des entrées) :

- l'affectation, comparaison, lecture d'un élément d'une liste ;
- les branchements conditionnels `if` ;
- le fait de « boucler » un tour de for ou de while ;
- l'instruction `return` .

On les appelle **opérations élémentaires**.

Ainsi, si on reprend le cas de la recherche d'un minimum, en notant $a, b, c \dots$ ces temps d'exécution constants, en notant n la taille de la liste :

```
def minimum(liste):
    xmin = liste[0]      # a
    for x in liste:      # b (n fois)
        if x < xmin:     #   c (n fois)
            xmin = x     #   d (entre 0 et n fois)
    return xmin          # e
```

Le nombre total d'opération est donc, dans le pire cas (on effectue toujours les opérations du if) :

$$C_n = a + (b + c + d) \times n + e = An + B$$

et donc, quand n devient très grand, le terme proportionnel à n domine. On notera

$$\boxed{C_n = \mathcal{O}(n)}$$

qui se lit « C_n est un grand o de n » et signifie

$$C_n \underset{n\to\infty}{\sim} An \quad \text{avec} \quad A \text{ constant}$$

Cela nous montre que si on double la taille de la liste, on double le temps de calcul. Un tel algorithme est dit **à complexité linéaire**.

3.2. Fin paresseuse

Certains algorithmes permettent d'avoir la réponse avant d'avoir parcouru toute la collection. Par exemple, si on souhaite savoir si une liste contient 0 :

```
def présence_0(liste):
    trouvé = False
    for x in liste:
        # trouvé est vrai si 0 in liste[0:i]
        if x == 0:
            trouvé = True
        # trouvé est vrai si 0 in [0:i+1]
    return trouvé
```

Notons que les opérations élémentaires sont répétées n fois. Cependant, dès que l'on trouve un élément de la liste valant 0, on peut arrêter le parcours : la réponse est `True`. On peut donc arrêter la boucle en avance et renvoyer la réponse sans faire tous les tours :

```
def présence_0(liste):
    trouvé = False
    for x in liste:
        if x == 0:
            trouvé = True
            break
    return trouvé
# autre proposition
def présence_0(liste):
    for x in liste:
        if x == 0:
            return True
    return False
```

La première utilise `break` pour éviter les tours excessifs, et permet de conserver la structure usuelle d'un parcours. La deuxième exploite le fait que la fonction s'arrête à l'instant où l'instruction `return` est rencontrée.

Il faut s'assurer que, si 0 n'est pas présent, **toute** la liste est parcourue.

Finalement, si un tel algorithme renvoie une réponse booléenne, il y a souvent possibilité de finir l'algorithme sans parcourir toute la liste pour une des deux réponses. Comme les opérateurs `or` et `and`, on dira que cet algorithme est paresseux.

Comme dans le pire cas, le nombre d'opérations élémentaires reste proportionnel à n, on dit tout de même que la complexité est $\mathcal{O}(n)$.

ITC – cours n°2

Exercices

1 – Écrire une fonction `maximum` qui calcule et renvoie le maximum d'une liste.

2 – Écrire une fonction `min_max` qui calcule et renvoie simultanément le minimum et le maximum d'une liste.

3 – Écrire une fonction `deux_maxima` qui calcule et renvoie les deux valeurs maximum d'une liste (on supposera que toutes les valeurs sont deux à deux distinctes).

4 – Écrire une fonction `pos_maximum` qui calcule et renvoie l'indice auquel se situe le maximum d'une liste.

5 – Écrire une fonction `moyenne` qui calcule et renvoie la moyenne des éléments d'une liste.

6 – Écrire une fonction `écarttype` qui calcule et renvoie l'écart-type des éléments d'une liste. On pourra réutiliser la fonction `moyenne`.

7 – Écrire une fonction `produit` qui calcule et renvoie le produit des éléments d'une liste.

8 – Écrire une fonction `pos_pair` qui renvoie une liste contenant les indices auxquels se situent les nombres pairs dans une liste fournie en entrée. Par exemple,

```
In [1]: pos_pair([0, 2, 3, 1, 7, 8, 6])
Out[1]: [0, 1, 5, 6]
```

9 – Écrire une fonction `occurrences` qui prend en paramètres un entier `val` et une liste, et qui renvoie le nombre d'occurrences de `val` dans la liste.

10 – Écrire une fonction `est_triée` qui prend en paramètre une liste et vérifie si la liste est triée :

```
In [1]: est_triée([0, 2, 3, 1, 7, 8, 6])
Out[1]: False
In [2]: est_triée([-4, -2, 3, 5, 7, 8, 13])
Out[2]: True
```