Informatique PCSI
Prérequis TP 1 : parcours séquentiel

1 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
```

Le nombre `maxi1` est le maximum, `maxi2` est le second maximum. Suivant les cas, le nombre total de comparaisons, en comptant celle effectuée avant la boucle, est compris entre $n-1$ et $2n-3$. Si la première comparaison échoue à chaque fois, il n'y a que $n-1$ comparaisons, si elle réussie à chaque fois, il y en a $2n-3$.

Il est possible de diminuer le nombre de comparaisons pour cette recherche. En effet lorsqu'on a effectué $n-1$ comparaisons pour trouver le maximum, on est sûr que le second maximum fait partie des éléments qui ont été comparés au maximum. L'algorithme est nettement moins simple à écrire.

2 Comptage

Dictionnaires

Dans une liste, un n-uplet ou une chaîne de caractères, les éléments sont ordonnés. On les repère par leur indice. Si n est la longueur, à chaque entier de 0 à $n-1$ correspond un élément.

- **Définition**

Un dictionnaire, objet de type dict, est une association entre des clés et des valeurs. Les clés sont des objets non mutables, les valeurs des objets quelconques. Ces clés ne sont pas ordonnées. On accède à une clé en temps constant selon un processus qu'il n'est pas nécessaire de décrire ici. On accède à une valeur par sa clé.

Pour créer un dictionnaire, on écrit entre des accolades les couples `clé: valeur`, une clé et la valeur sont séparées par le signe deux-points, les couples sont séparés par des virgules.

Exemple : `jours = {"dimanche": 1, "lundi": 2, "mardi": 3, "mercredi": 4, "jeudi": 5, "vendredi": 6, "samedi": 7}`.

- **Manipulation**

La construction par compréhension existe pour les dictionnaires.

```
>>> d = {x: x**2 for x in range(1, 5)}
>>> d
{1: 1, 2: 4, 3: 9, 4: 16}
```

- **Accès aux clés**

La méthodes `keys` permet d'obtenir les différentes clés.

```
>>> jours.keys()
dict_keys(['dimanche', 'lundi', 'mardi', 'mercredi', 'jeudi',
           'vendredi', 'samedi'])
```

La méthode `items` permet d'obtenir l'ensemble des couples (clé, valeur).

```
>>> jours.items()
dict_items([('dimanche', 1), ('lundi', 2), ('mardi', 3),
('mercredi', 4), ('jeudi', 5), ('vendredi', 6), ('samedi', 7)])
```

Le mot `in` permet de tester l'appartenance d'une clé à un dictionnaire, pas d'une valeur.

```
>>> 4 in jours
False
>>> "dimanche" in jours
True
>>>
```

Donc, avec `for elt in dic`, où `dic` est un dictionnaire, la variable d'itération `elt` est une clé.

```
>>> cle = []
>>> for elt in jours:
        cle.append(elt)
```

La variable `cle` a pour valeur `['dimanche', 'lundi', 'mardi', 'mercredi', 'jeudi', 'vendredi', 'samedi']`.
Mais il est aussi possible d'itérer sur les couples clés-valeurs en utilisant la méthode `items`.

```
>>> for cle, val in jours.items():
        print(cle, val)

dimanche 1
lundi 2
mardi 3
mercredi 4
jeudi 5
vendredi 6
samedi 7
```

L'accès à une valeur s'obtient comme avec les listes, les tuples ou les chaînes. Mais il faut préciser la clé à la place de l'indice. Par exemple, `jours['dimanche']` nous donne la valeur 1.

- **Insertion, modification**

Comme pour les listes, il est possible de modifier une valeur par affectation, par exemple avec une instruction comme `jours['dimanche'] = 0`. Pour insérer un élément, on écrit une instruction comme `jours['dimanche2'] = 8`. On ajoute un deuxième dimanche à la semaine.

Si la clé `c` existe, l'instruction `d[c] = val` modifie la valeur associée à la clé `c`. Si la clé `c` n'existe pas, elle est créée et la valeur `val` lui est associée.

- **Nombre d'éléments :** comme avec les listes, les n-uplets et les chaînes, la fonction `len` renvoie la longueur d'un dictionnaire qui est à la fois le nombre de clés et le nombre de couples.

- **Copie**

Les comportements sont similaires à ceux rencontrés avec les listes.

```
>>> d1 = {'one': 1, 'two': 2, 'three': 3}
>>> d2 = d1
>>> d2['four'] = 4
>>> d1
{'one': 1, 'two': 2, 'three': 3, 'four': 4}
```

Dans le code précédent, `d1` et `d2` sont deux noms différents liés au même dictionnaire.
Avec le code qui suit, le nom `d2` est lié à un nouveau dictionnaire.

```
>>> d1 = {'one': 1, 'two': 2, 'three': 3}
>>> d2 = dict(d1)
>>> d2['one'] = 0
>>> d1
{'one': 1, 'two': 2, 'three': 3}
```

Nous pouvons aussi utiliser la méthode `copy` en écrivant `d2 = d1.copy()`. Mais attention, si les valeurs sont des listes, les questions posées sur les copies de listes sont à nouveau présentes.

Dans le doute, pour obtenir une vraie copie, il est plus sûr d'utiliser la fonction `deepcopy` du module `copy` qui se comporte comme avec les listes.

2.1 Programmes

Dans un objet de type list, les éléments sont ordonnés. Ils sont indexés par une suite d'entiers 0, 1, 2, etc. Avec le type dict, les éléments sont indexés par des clés qui peuvent être de n'importe quel type *non mutable*, par exemple int, float, tuple. L'utilisation d'un dictionnaire est décrite à la fin du livre en annexe.

Si les éléments à compter sont des entiers naturels, on peut utiliser une liste. On considère alors que les indices de la liste représentent ces entiers et les valeurs de la liste leur nombre. Soit le maximum est donné, soit on le calcule avec la fonction `maximum`. Le langage Python propose aussi une fonction `max`.

```
def compte(entiers):
    m = max(entiers) #ou maximum(entiers)
    cpts = [0 for i in range(m+1)]
    for n in entiers:
        cpts[n] = cpts[n] + 1
    return cpts
```

Une condition pour que la méthode soit efficace est que la dispersion des éléments ne soit pas trop importante afin de ne pas avoir une liste trop longue dont les éléments sont en majorité nuls et ne servent à rien. Si c'est le cas, une solution est d'utiliser un dictionnaire. Les valeurs des éléments du tableau sont les clés du dictionnaire. Cette méthode permet en plus de traiter les cas où les éléments à compter ne sont pas des entiers naturels.

```
def compte(liste):
    d = {}
    for elt in liste:
        if elt in d:
            d[elt] = d[elt] + 1
        else:
            d[elt] = 1
    return d
```