TP : Utilisation des dictionnaires

L'objectif de ce TP est d'utiliser des dictionnaires pour éviter des recherches répétées. On ne s'intéresse pas à leur implémentation. Les listes manipulées contiennent des entiers, éventuellement négatifs. On note n la longueur de la liste étudiée. Les fonctions ne doivent pas modifier la liste reçue en argument.

Pour les analyses de complexité, on considère les opérations sur les entiers en temps constant. On suppose également que les recherches, insertions et mises à jour dans un dictionnaire coûtent $O(1)$ en moyenne. Les complexités obtenues avec les dictionnaires seront donc des complexités moyennes.

Quelques opérations utiles

```
d = {}                  # dictionnaire vide
d[7] = 2                # associer la valeur 2 a la cle 7
d[7] = 5                # remplacer la valeur associee a 7
7 in d                  # True : tester la presence d'une cle
d[7]                    # 5 : acceder a la valeur associee
d.get(9, 0)             # 0 : valeur par defaut si la cle est absente
```

L'appel `d.get(9, 0)` ne modifie pas le dictionnaire. L'accès `d[9]` soulèverait ici une exception `KeyError`.

1 Deux nombres de somme donnée

On reçoit une liste L et un entier c. On cherche deux indices $i < j$ tels que $L[i] + L[j] = c$. Une même case ne peut pas être utilisée deux fois, mais deux cases peuvent contenir la même valeur.

Question 1.

Écrire une fonction `deux_somme_naif(L, c)` qui examine les couples d'indices et renvoie un couple convenable, ou `None` s'il n'en existe pas. Si plusieurs couples conviennent, on accepte n'importe lequel. Exemples de retours possibles :

```
>>> deux_somme_naif([4, 7, 1, 9], 10)
(2, 3)
>>> deux_somme_naif([5], 10)
>>> deux_somme_naif([5, 5], 10)
(0, 1)
```

Dans les exemples, l'absence d'affichage après un appel signifie que la fonction renvoie `None`.

Solution. Code

```
def deux_somme_naif(L, c):
    for j in range(len(L)):
        for i in range(j):
            if L[i] + L[j] == c:
                return (i, j)
    return None
```

□

Question 2.

Donner la complexité temporelle dans le pire cas et l'espace supplémentaire utilisé par cette fonction.

Solution. Lorsque aucun couple ne convient, on teste $n(n-1)/2$ couples. La complexité temporelle est $\Theta(n^2)$ dans le pire cas, et l'espace supplémentaire est $O(1)$. □

Question 3.

Écrire une fonction `deux_somme(L, c)` qui parcourt la liste une seule fois. Utiliser un dictionnaire associant à chaque valeur déjà rencontrée un indice où elle apparaît. Exemples de retours possibles :

```
>>> deux_somme([4, 7, 1, 9], 10)
(2,3)
>>> deux_somme([5], 10)
>>> deux_somme([5, 5], 10)
(0, 1)
>>> deux_somme([-3, 8, 2], 5)
(0, 1)
```

Solution. On cherche $c - L[j]$ parmi les valeurs rencontrées avant l'indice j.

```
def deux_somme(L, c):
    vus = {}
    for j in range(len(L)):
        complement = c - L[j]
        if complement in vus:
            return (vus[complement], j)
        vus[L[j]] = j
    return None
```

Cette fonction renvoie le couple (i, j) convenable dont la borne supérieure (j) est minimum. □

Question 4.

Justifier la correction de l'algorithme, puis donner ses complexités temporelle et spatiale.

Solution. Pourquoi faut-il effectuer la recherche avant l'insertion de $L[j]$ dans le dictionnaire ? Une insertion préalable pourrait conduire à utiliser deux fois la même case : la liste contenant seulement 5, avec $c = 10$, en est un exemple.

Montrons que ce qui est renvoyé convient : Avant l'itération au tour j, le dictionnaire contient exactement les valeurs rencontrées aux indices strictement inférieurs à j, avec un indice valide pour chacune. Tout couple renvoyé convient donc et possède deux indices distincts. Cet invariant est préservé par l'insertion.

Réciproquement, montrons que la méthode trouve toujours une solution s'il y en a une. Si un couple (i, j) avec $i < j$ convient, la valeur $L[i]$ (qui vaut $c - L[i]$) est présente dans le dictionnaire lors du traitement de j : un couple sera trouvé au plus tard à cette étape. La complexité temporelle est $O(n)$ en moyenne et l'espace supplémentaire est $O(n)$. □

2 Plus long segment sans répétition

Soit L une liste de n entiers.

Un segment de L est une suite de cases consécutives. On le décrit par un couple (a, b) avec $0 \leq a \leq b \leq n$: il correspond à `L[a:b]` et sa longueur vaut $b - a$. L'indice b est donc exclu. Le segment correspondant à $(0, 0)$ est vide. On cherche un segment de longueur maximale dont les valeurs sont toutes distinctes. En cas d'égalité, on renverra celui dont l'indice de début est le plus petit.

Question 5.

Écrire une fonction `sans_repetition_naif(L)`. Pour chaque indice de départ, parcourir les valeurs suivantes en les mémorisant dans un dictionnaire, jusqu'à la première répétition. Ne pas construire de tranches de la liste dans les boucles [1]. On n'impose pas de contrainte sur le nombre de boucles.

```
>>> sans_repetition_naif([4, 2, 4, 5, 2, 3, 1])
(2, 7)
>>> sans_repetition_naif([8, 8, 8])
(0, 1)
>>> sans_repetition_naif([])
(0, 0)
```

Solution. Code

```
def sans_repetition_naif(L):
    meilleur = (0, 0)
    for a in range(len(L)):
        vus = {}
        b = a
        while b < len(L) and L[b] not in vus:
            vus[L[b]] = True
            b += 1
        if b - a > meilleur[1] - meilleur[0]:
            meilleur = (a, b)
    return meilleur
```

Sur une liste de valeurs toutes distinctes, on examine $n + (n-1) + \cdots + 1$ valeurs : le coût moyen est $\Theta(n^2)$ pour cette famille d'entrées. L'espace supplémentaire est $O(n)$. □

On veut maintenant parcourir la liste une seule fois (UNE SEULE BOUCLE). On mémorise l'indice `a` du début du segment étudié, le meilleur segment rencontré `best` et, dans un dictionnaire, la dernière position connue de chaque valeur.

On utilise l'invariant suivant pour l'indice courant j : avant de traiter $L[j]$, le segment `L[a:j]` ne contient aucune répétition et a est le plus petit possible. Tout l'enjeu est de savoir quand mettre à jour `a` et `best`.

Question 6.

Écrire la fonction `sans_repetition(L)` selon cette méthode. Elle doit renvoyer les mêmes résultats que la version précédente.

```
>>> sans_repetition([4, 2, 4, 5, 2, 3, 1])
(2, 7)
>>> sans_repetition([1, 2, 2, 1])
(0, 2)
>>> sans_repetition([])
(0, 0)
```

Solution. Code

```
def sans_repetition(L):
    dernier = {}
    a = 0
    meilleur = (0, 0)
    for j in range(len(L)):
        x = L[j]
```

1. taduction : pas de slicing. Ca coûte cher !

```
        if x in dernier:
            a = max(a, dernier[x] + 1)
        dernier[x] = j
        if j + 1 - a > meilleur[1] - meilleur[0]:
            meilleur = (a, j + 1)
    return meilleur
```

□

Question 7.

Justifier qu'après le traitement de l'indice j, a est le plus petit indice tel que `L[a:j+1]` soit sans répétition. En déduire la correction de la fonction et ses complexités.

Solution. On procède par récurrence sur les tours de boucles. Le temps $j = -1$ est celui d'avant la boucle.

Cas de base Au départ, $j = -1$ et $a = 0$. $L[0 : (-1) + 1]$ est une liste vide donc sans répétion. Et a est effectivement le plus petit indice possible puisqu'il vaut 0.

Hérédité On suppose l'invariant vérifié à la fin du tour $j - 1$. S'il y a un tour j, a est le plus petit début admissible pour le segment se terminant juste avant j.

Si $L[j]$ n'est pas déjà dans une position entre entre a et $j - 1$, alors $L[a : j + 1]$ est sans répétition. On ne peut pas prendre a plus petit car cela introduirait une répétition avant le dernier élément du segment (par HR a est minimum pour que $L[a : j]$ soit sans répétition). Et donc l'invariant est vérifié.

Si $L[j]$ est déjà dans une position p entre entre a et $j - 1$, alors, si on veut éviter une répétition de $L[j]$ en intégrant la position j, il faut et il suffit de positionner a après p. Ainsi, après la mise à jour de a, $L[a : j + 1]$ est sans répétition et on ne peut pas prendre a plus petit.

On examine ainsi le plus long segment sans répétition pour chaque extrémité droite. Le meilleur de ces segments est un optimum global. La mise à jour se fait seulement en cas d'amélioration stricte : elle conserve donc le premier optimum rencontré, c'est à dire celui dont le début est le plus petit.

Le coût temporel est $O(n)$ en moyenne et l'espace supplémentaire $O(n)$.

□

3 Segments de somme donnée

On cherche maintenant des segments **non vides** dont la somme vaut un entier c. Les entiers de la liste peuvent être négatifs ou nuls. On définit les sommes préfixes par

$$S_0 = 0, \qquad S_j = \sum_{k=0}^{j-1} L[k] \quad (1 \leq j \leq n).$$

Question 8.

Écrire la fonction `sommes_prefixes(L)` qui renvoie la liste $[S_0, S_1, \ldots, S_n]$, en temps linéaire.

```
>>> sommes_prefixes([1, 2, -1, 1, 0, 3])
[0, 1, 3, 2, 3, 3, 6]
>>> sommes_prefixes([])
[0]
```

Solution. Code

```
def sommes_prefixes(L):
    S = [0]
    for x in L:
        S.append(S[-1] + x)
    return S
```

La fonction coûte $\Theta(n)$ en temps et utilise $\Theta(n)$ cases supplémentaires, pour $n \geq 1$. □

Question 9.

Exprimer la somme de `L[i:j]` en fonction de S_i et S_j. En déduire la valeur à rechercher parmi les sommes préfixes précédentes lorsqu'on connaît S_j et qu'on veut une somme c.

Solution. La somme du segment vaut $S_j - S_i$. On cherche donc les indices $i < j$ tels que $S_i = S_j - c$. Une même somme préfixe peut apparaître à plusieurs indices, chacun donnant un segment différent. Par exemple, pour $[0, 0]$ et $c = 0$, il existe trois segments non vides. Un simple test de présence ne suffit pas à les compter. □

Question 10.

Écrire la fonction `compte_segments_naif(L, c)` qui renvoie le nombre de segments non vides de somme c en utilisant la fonction précédente. On ne cherche pas une complexité optimale.

```
>>> compte_segments_naif([1, 2, -1, 1, 0, 3], 3)
6
>>> compte_segments_naif([0, 0], 0)
3
>>> compte_segments([], 0)
0
```

Solution. Code

```
def compte_segments_naif (L , c ) :
    S = sommes_prefixes(L)
    nb = 0
    for j in range(len(S)):
        for i in range(j):
            if S[j] - S[i] == c:
                nb+=1
    return nb
```

□