

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

```
1 d = {}                # dictionnaire vide
2 d[7] = 2              # associer la valeur 2 a la cle 7
3 d[7] = 5              # remplacer la valeur associee a 7
4 7 in d                # True : tester la presence d'une cle
5 d[7]                  # 5 : acceder a la valeur associee
6 d.get(9, 0)           # 0 : valeur par défaut 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 :

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

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

Question 2.

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

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 :

```

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

```

Question 4.

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

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 boucles¹. On n'impose pas de contrainte sur le nombre de boucles.

```

1 >>> sans_repetition_naif([4, 2, 4, 5, 2, 3, 1])
2 (2, 7)
3 >>> sans_repetition_naif([8, 8, 8])
4 (0, 1)
5 >>> sans_repetition_naif([])
6 (0, 0)

```

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.

```

1 >>> sans_repetition([4, 2, 4, 5, 2, 3, 1])
2 (2, 7)
3 >>> sans_repetition([1, 2, 2, 1])
4 (0, 2)
5 >>> sans_repetition([])
6 (0, 0)

```

1. traduction : pas de slicing. Ça coûte cher !

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.

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

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

Question 8.

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

```
1 >>> sommes_prefixes([1, 2, -1, 1, 0, 3])
2 [0, 1, 3, 2, 3, 3, 6]
3 >>> sommes_prefixes([])
4 [0]
```

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 .

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.

```
1 >>> compte_segments_naif([1, 2, -1, 1, 0, 3], 3)
2 6
3 >>> compte_segments_naif([0, 0], 0)
4 3
5 >>> compte_segments([], 0)
6 0
```

Question 11.

Écrire la fonction `compte_segments(L, c)` qui renvoie, en une seule boucle, le nombre de segments non vides de somme c , en temps $O(n)$ en moyenne. Calculer les sommes préfixes au fil du parcours, sans construire leur liste. Initialiser le dictionnaire par $\{0: 1\}$. Expliquer cette initialisation et l'ordre des recherches et mises à jour.

```
1 >>> compte_segments([1, 2, -1, 1, 0, 3], 3)
2 6
3 >>> compte_segments([0, 0], 0)
4 3
5 >>> compte_segments([], 0)
6 0
```

Question 12. Approfondissement

Écrire une fonction `plus_long_segment_somme(L, c)` qui renvoie un couple (a, b) décrivant un segment non vide de somme c et de longueur maximale, ou `None` s'il n'en existe pas. En cas d'égalité, choisir le plus petit début. Quelle information faut-il associer à chaque somme préfixe ? On attend encore un temps $O(n)$ en moyenne.

```
1 >>> plus_long_segment_somme([1, 2, -1, 1, 0, 3], 3)
2 (0, 5)
3 >>> plus_long_segment_somme([0, 0], 0)
4 (0, 2)
5 >>> plus_long_segment_somme([2, 4], 3)
6 >>> plus_long_segment_somme([], 0)
```