Programmation Python
TP n° 2 : Listes et dictionnaires

Objectifs d'apprentissage

Dans ce TP, vous acquerrez une expérience pratique dans :

— La manipulation des listes et des tableaux associatifs (dict) ;

— L'utilisation de structures de données pour résoudre des questions algorithmiques.

N'oubliez pas de continuer à typer, tester et documenter toutes vos réponses !

Exercice 1 : Égalités

Faites l'exercice suivant d'abord sur papier. Vérifiez ensuite vos réponses avec Python.

1. Que affichera le code suivant ? Expliquer.

```python
students = [["Adam", 20], ["Adam", 14]]
students[1][1] = 20
x1 = students[0] == students[1]
x2 = students[0] is students[1]
x3 = id(students[0]) == id(students[1])
print(f"{x1=}, {x2=}, {x3=}")
```

2. Que affichera le code suivant ? Expliquer.

```python
l1 = [1, [2,3,4], (7,8,9)]
l2 = [x for x in l1]
l1[1].append(5)
l1[2] += (10,11)
print(l2)
```

Exercice 2 : Parenthèses équilibrées

1. Écrire une fonction qui vérifie en $O(n)$ si une chaîne de caractères composée uniquement des caractères (et) est *équilibrée*, c'est-à-dire que chaque parenthèse ouvrante a une parenthèse fermante correspondante, et que les parenthèses sont correctement imbriquées.
2. Étendre votre fonction pour qu'elle puisse prendre en compte les chaînes de caractères qui contiennent en plus les symboles $\{, \}, [,]$. Pour qu'une chaîne soit équilibrée, le type de parenthèse fermante doit correspondre à une parenthèse ouvrante du même type et au même niveau d'imbriquement.

Exercice 3 : Palindromes et files d'attente à double extrémité

Cet exercice montre deux façons d'écrire une fonction qui vérifie si une chaîne de caractères est un *palindrome*, c'est-à-dire une chaîne qui est égale à son propre inverse.

1. Implementer une solution facile : parcourir la chaîne et comparer les caractère à la position i comptant du début et de la fin.
2. Implementer une autre solution, en utilisant le type `deque` (file d'attente à double extrémité) du module `collections` (doc) : Initialiser la chaîne de caractères comme un `deque`, et, à chaque étape, retirer un caractère du début et de la fin et vérifier qu'ils sont égaux.
3. Y a-t-il une différence de complexité ? Faire de tests pour comparer les temps d'exécution des solutions.

Exercice 4 : Scrabble

L'objectif de cet exercice est de simuler le processus de recherche des mots de valeur maximale dans le jeu Scrabble. On définit d'abord `s = "|AEIOULNSTR|DG|BCMP|FHVWY|K||JX||QZ"`. Chaque lettre majuscule apparaît une fois dans `s`. On définit le *score* d'une lettre comme le nombre d'occurrences du caractère `|` à gauche de cette lettre. Par exemple, A, L et R ont un score de 1, D et G ont un score de 2, et Q et Z ont un score de 10.

1. Étant donné la chaîne `s` ci-dessus, écrire une expression de compréhension qui définit un dictionnaire `scores` dont les clés sont les lettres majuscules et les valeurs sont les scores de ces lettres. *Indication.* Utiliser les fonctions `split` et `enumerate`.
2. Écrire une fonction `score_word(s)` qui renvoie, étant donné un mot s en lettres majuscules, la somme des scores des lettres du mot.
3. Écrire une fonction `random_letters(n)` qui prend en paramètre un nombre entier n et qui renvoie une chaîne de n lettres majuscules tirés au hasard de l'alphabet. Note : la variable `ascii_uppercase` du module `string` peut être utile.
4. Écrire une fonction `words(s)` qui prend en paramètre une chaîne de caractères s et qui renvoie la liste de tous les mots que l'on peut former en utilisant les caractères dans s. Chaque occurrence d'un caractère peut être utilisé au plus une fois : si s = HELLO, on peut former OLLH mais pas LEE. *Indication.* Utiliser `permutations` du module `itertools`.
5. Le fichier `wordlist.txt` contient une liste de dix mille mots. Importer cette liste dans une variable `legal_words`. *Indication.* Pour ouvrir et lire un fichier, utiliser les fonctions `open` et `readlines`. Par exemple :

```
with open("wordlist.txt", "r") as f:
    for l in f.readlines():
        print(l)
```

   Notez que chaque ligne se termine par un caractère `\n`, que vous pouvez supprimer d'une chaîne en lui appliquant la fonction `rstrip`.
6. Écrire une fonction `find_best_word1(s, ws)` qui prend en paramètre une chaîne de caractères s, génère d'abord `words(s)` et cherche ensuite dans cette liste le mot à valeur maximale qui apparaît aussi dans `ws`. Testez `find_best_word1(random_letters(7), legal_words)` et expliquez pourquoi cette implémentation n'est pas optimale.
7. Écrire une fonction `is_possible(w,s)` qui prend en paramètre deux chaînes de caractères w et s et qui renvoie un booléen qui indique si le mot w peut être formé en utilisant les caractères dans s. *Indication.* Utiliser `Counter` du module `collections`.
8. Écrire une fonction `find_best_word2(s, ws)` plus efficace. Testez votre implémentation avec `find_best_word1(random_letters(50),legal_words)`.
9. (*) Optionnel, à faire après les autres exercices de ce TP : cherchez la structure de données "trie" et implementez des methodes plus efficaces. Lisez l'article "The world's fastest Scrabble program".

Exercice 5 : Nombre de chemins dans un treillis et programmation dynamique

Dans un treillis de hauteur h et de largeur w, entiers naturels, on voudrait calculer le nombre $c(h, w)$ de plus courts chemins allant du coin supérieur gauche au coin inférieur droit.

1. Ecrire une relation de récurrence pour $c(h, w)$.
2. Implementer une fonction qui calcule c par récurrence. Quelle est la complexité en temps et en espace ?
3. Améliorer la complexité en temps en utilisant la *mémoïsation* : créer un cache de taille $(h+1) \times (w+1)$, qui contiendra à l'indice i, j la valeur de $c(i, j)$. Quelle est la complexité en temps et en espace ?

Exercice 6 : Planification de tâches

Le but de cet exercice est de trouver une planification de tâches qui optimise une valeur.
On supposera données n tâches possibles sous la forme de trois listes S, E et V de même longueur n, où pour chaque $0 \leqslant i < n$, la tâche i dure du temps S_i à E_i et a pour valeur V_i. Un sous-ensemble $P \subseteq \{0, \ldots, n-1\}$ de tâches est un *plan possible* si les intervalles $[S_i, E_i[$ et $[S_j, E_j[$ sont disjoints pour chaque $i \neq j \in P$.

1. Ecrire une fonction qui trouve la valeur maximale de $\sum_{i \in P} V_i$, pour P un plan possible. Analyser la complexité de votre implémentation.
2. Générer un jeu de tests. Quelles sont les limites des valeurs d'entrée pour que la fonction donne une réponse dans un temps raisonnable (pas plus de 5 secondes) ?
3. Améliorer votre implémentation de sorte qu'elle donne une réponse dans un délai raisonnable pour des cas de tests sous les contraintes suivantes : $1 \leqslant n \leqslant 5 \times 10^4$, $1 \leqslant V_i \leqslant 10^4$ et $0 \leqslant S_i < E_i < 10^9$. (Tests en ligne)