TP D'INFORMATIQUE 10
Programmation et complexité

1 Recherche par clé

Dans cet exercice, on étudie la possibilité de représenter des associations clé-valeur par une liste, plutôt que de passer par les dictionnaires préexistant en Python. On travaille ainsi sur des listes de couples `(c,v)`, où la valeur `v` est associée à la clé `c`. Par exemple, la liste

```
L = [('PSI', 263), ('MPSI', 273), ('PCSI', 264), ('PC*', 266), ('MP*', 267)]
```

associe la valeur `263` à la clé `'PSI'`.

1. Écrire une fonction `valeur_associee` prenant en argument une telle liste de couples `L` et une clé `x`, et renvoyant une valeur associée à `x`. La fonction renverra `None` si aucune valeur n'est associée à `x`
2. Déterminer la complexité de cette fonction, dans le meilleur et dans le pire cas, en fonction de la longueur de `L`.
3. Écrire une fonction `valeur_associee_dicho` renvoyant le même résultat, mais en exploitant le principe de la recherche dichotomique, en supposant que la liste est triée selon les clés. Il faut modifier l'exemple initial pour que les clés soient triées dans l'ordre du dictionnaire :
   `L = [('MP*', 267), ('MPSI', 273), ('PC*', 266), ('PCSI', 264), ('PSI', 263)]`
4. On note n la longueur de la liste, et g et d les indices des bornes gauche et droite de l'intervalle de recherche lors de la recherche dichotomique. Montrer par récurrence sur k que s'il y a une k-ième itération, alors à son début on a :
$$d - g \leq \frac{n}{2^{k-1}}$$
5. En déduire que s'il y a une k-ième itération avec $k = \lfloor \log_2(n) \rfloor + 2$, alors c'est la dernière itération.
6. En déduire la complexité dans le pire cas de la fonction `valeur_associee_dicho`. Quelle est sa complexité dans le meilleur cas ?

2 Tri fusion

On rappelle le principe du tri fusion :
- si la liste est vide ou admet un seul élément alors on s'arrête : elle est déjà triée ;
- on divise la liste en deux moitiés, que l'on trie récursivement ;
- on les fusionne en une liste triée en interclassant leurs éléments.

1. Écrire une fonction `fusion` qui prend en argument deux listes `L1` et `L2` **supposées triées** et retourne la fusion triée des deux listes. Cette fonction devra être de coût linéaire en la somme des longueurs des listes. En particulier, on ne pourra pas utiliser une fonction de tri préalablement écrite. On justifiera que la complexité demandée est respectée.
   Indication : Stocker en mémoire les deux indices `i1` et `i2` correspondant au parcours de chaque liste, et à chaque étape ajouter la plus petite des deux valeurs `L1[i1]` et `L2[i2]` au résultat.
2. En déduire une fonction récursive `tri_fusion` implémentant le tri fusion.
3. Déterminer la complexité de la fonction `tri_fusion`.

3 Tri rapide

On rappelle le principe du tri rapide (dans sa version non en place) : pour trier une liste de longueur supérieure à 2 :

- on choisit un élément, appelé **pivot**, par exemple le premier élément,
- On répartit les éléments en 3 listes, selon qu'ils soient strictement inférieurs, égaux ou strictement supérieurs au pivot,
- on trie récursivement la première et la dernière liste,
- on concatène dans le bon ordre les 3 listes.

1. Écrire une fonction `repartition` implémentant la deuxième étape.
2. Déterminer la complexité de cette fonction.
3. Écrire une fonction implémentant le tri rapide.
4. On considère une liste dont tous les éléments sont distincts. Déterminer la complexité du tri rapide, dans l'hypothèse où les appels récursifs se font toujours sur des listes maximalement déséquilibrées.
5. Déterminer la complexité du tri rapide dans le cas où les appels récursifs se font toujours sur des listes maximalement équilibrées.
6. En déduire la complexité dans le meilleur et dans le pire cas du tri rapide.
7. Quel est la complexité dans le meilleur cas si on retire l'hypothèse que tous les éléments sont distincts ?

4 Recherche de mot dans un texte

1. Écrire une fonction `recherche_mot` prenant en argument une chaîne de caractères `texte` et une chaîne de caractères `mot`, et renvoyant un indice `i` tel que `mot` apparait dans `texte` à partir de l'indice `i`. Si le mot n'apparait pas dans le texte, la fonction renverra `None`.

   Par exemple :
   - `recherche_mot('hello world!', 'world'` doit renvoyer `6`.
   - `recherche_mot('hello world!', 'artichaud'` doit renvoyer `None`
2. Déterminer la complexité de la fonction précédente, dans le meilleur et dans le pire cas, en fonction de la longueur du texte et de la longueur du mot.

5 Inclusion ensembliste

1. Écrire une fonction `est_incluse` prenant en argument deux listes et testant si chaque élément de la première liste apparait dans la seconde.
2. Déterminer la complexité dans le pire cas de la fonction précédente, en fonction des longueurs des deux listes.
3. On fait l'hypothèse que les deux listes sont de taille comparable. Proposer une variante de `est_incluse` améliorant la complexité dans le pire cas. On pourra utiliser un tri fusion.