TD 4 : récursivité et algorithmes de tri

Les sujets de TD ne sont ni relevés, ni notés. Ils sont volontairement conçus pour ne pas pouvoir être traités en une seule séance ; n'hésitez cependant pas, si vous le souhaitez, à tenter de les finir chez vous.

Rappel : respectez les conventions de style (PEP 8 et *Zen of Python*), écrivez des tests, documentez votre code.

Introduction

Un *tri* est une fonction prenant en entrée une liste de données munies d'une relation d'ordre et permettant d'obtenir en sortie une liste contenant les mêmes valeur, ordonnées.

La fonction peut, au choix, modifier la liste initiale (ce que l'on appelle un tri *en place*) ou renvoyer une nouvelle liste sans toucher à la première.

Il existe de très nombreux algorithmes de tri, pouvant être plus ou moins adaptés à chaque situation. Pour évaluer un algorithme, plusieurs critères peuvent être intéressants, par exemple :

- La complexité temporelle dans le pire des cas, souvent simple à calculer.
- La complexité temporelle moyenne, c'est-à-dire l'espérance de la complexité si la liste en entrée a été tirée aléatoirement, la loi de probabilités dépendant du problème posé.
- La complexité spatiale (quantité de mémoire consommée) moyenne et dans le pire des cas. Elle est souvent liée au fait que le tri se fasse en place ou non.
- Le comportement sur les listes presque triées : certains algorithmes feront très peu d'étapes si la liste en entrée est déjà presque triée, tandis que d'autres se seront pas plus efficaces.
- La stabilité. Cette propriété est vérifiée si deux objets de la liste de départ possédant la même clef de tri (donc pouvant être permutés une fois la liste triée) seront dans le même ordre au départ et à l'arrivée.

1. Vous avez vu en première année le *tri à bulles* et le *tri par insertion*. Quelle est leur complexité dans le pire des cas ? S'exécutent-ils en place ? Comment se comportent-ils sur des listes presque triées ? Sont-ils stables ?

   De nombreux algorithmes de tri reposent sur une méthode de conception appelée « diviser pour régner » : un problème initial est *divisé* en deux problèmes plus simples, chacun de ces problèmes est *résolu* séparément, puis les solutions des deux sous-problèmes sont *combinées* pour obtenir la solution du problème initial. La résolution des sous-problèmes peut elle-même nécessiter de les diviser à nouveau, jusqu'à tomber sur un problème suffisamment simple pour que l'on sache le résoudre. La recherche par dichotomie est l'un des exemples les plus simples d'algorithme conçus selon la méthode « diviser pour régner ».

   Un algorithme basé sur la méthode de « diviser pour régner » aura typiquement la structure suivante :

   ```
   def resoudre(probleme):
       if (est_un_cas_simple(probleme)):
           return solution_cas_simple(probleme)
       sous_probleme1, sous_probleme2 = diviser(probleme)
       sol_probleme1 = resoudre(sous_probleme1)
       sol_probleme2 = resoudre(sous_probleme2)
       return combiner(sol_probleme1, sol_probleme2)
   ```

   Ici, il est important de voir que l'on a défini une fonction *récursive* : dans certains cas, elle peut se rappeler elle-même.

2. Programmer deux fonctions récursives `fact` et `fibo` qui, pour un paramètre n, renvoient respectivement $n!$ et le u_n où $(u_n)_{n\in\mathbf{N}}$ est la suite de Fibonacci, définie par $u_0 = u_1 = 1$ et $\forall n \in \mathbf{N}, u_{n+2} = u_{n+1} + u_n$.

Programmation récursive du tri rapide

Pour le tri rapide, les opérations à mener sur la liste L sont :

0. Si L est de longueur 1, terminer. Sinon,
1. Choisir un élément quelconque de la liste, appelé *pivot* (π).
2. Séparer la liste entre les éléments plus petits que le pivot (L_-) et ceux plus grands (L_+). Modifier L pour mettre dans l'ordre L_-, π et L_+.
3. Trier séparément L_- et L_+.

Le tri se faisant en place, à l'étape 3, L_+ et L_+ sont en fait des sous-listes de L. On appliquera donc notre fonction de tri rapide à L *entre un indice de début et un indice de fin.*

3. Écrire une fonction `partitionner` prenant comme paramètres une liste `L` et un indice de pivot `p`. Cette fonction doit modifier `L` en une liste `L'` de sorte que `L'` soit une permutation de `L` contenant d'abord les éléments plus petits que `L[p]`, puis `L[p]`, et enfin les éléments plus grands que `L[p]`. Elle doit renvoyer l'indice où se situe `L[p]` dans `L'`.
   Par exemple, pour
   ```
   L = [42, 14, 3, 7, 12, 7, -2, 15, 1]
   p = 4
   ```
   `L` doit être modifiée en
   ```
   L' = [3, 7, 7, -2, 1, 12, 42, 14, 15]
   ```
   et la valeur de retour doit être `5`.

4. Modifier la fonction `partitionner` pour qu'elle prenne en plus deux paramètres `d` (début) et `f` (fin), `p` devant être compris entre `d` et `f` (strictement pour `f`). La modification ne doit alors plus être faite que sur `L[d:f]`.
   Par exemple, pour
   ```
   L = [1, 0, 3, 27, 42, 14, 3, 7, 12, 7, -2, 15, 1, 13, -11, 78]
   d = 4, p = 8, f = 13
   ```
   on doit avoir après exécution :
   ```
   L' = [1, 0, 3, 27, 3, 7, 7, -2, 1, 12, 42, 14, 15, 13, -11, 78]
   ```
   et une valeur de retour égale à `9`.

5. Écrire maintenant une fonction `aux_qsort` prenant comme paramètres `L` la liste à trier, `d` l'indice de début de la zone de la liste sur laquelle effectuer le tri et `f` l'indice de fin. On choisira arbitrairement pour pivot `p=d`.
   En déduire une fonction `qsort` prenant pour unique paramètre la liste `L` et la modifiant de sorte qu'elle soit triée selon l'algorithme du tri rapide.

Comparaison avec d'autres tris classiques

6. Implémenter le tri à bulles et le tri par insertion, modifiant également les listes en place.

   L'on souhaite maintenant évaluer ces trois algorithmes de tri. Le critère intéressant est le nombre de comparaisons effectuées pour trier la liste. Pour pouvoir les compter, l'on propose d'utiliser le morceau de code suivant. Notez que la signification de ce code est totalement hors-programme, nous allons juste voir comment l'utiliser dans le cas présent :

```
class Counter:
    nb = 0
    def lt(self, x, y):
        self.nb += 1
        return (x < y)
```

Un exemple d'utilisation est donné par :

```
c = Counter()
d = Counter()
print(c.nb)
print(d.nb)
print(c.lt(1, 2))
print(c.lt(0, -1))
print(d.lt(23, 11))
print(c.lt(42, 17))
print(c.nb)
print(d.nb)
```

Qui devrait afficher dans l'ordre `0`, `0`, `True`, `False`, `False`, `False`, `3`, `1`.
En fait, `c = Counter()` définit une nouvelle variable complexe, qui contient une sous-variable `nb` et une sous-fonction `lt`. La première est un compteur, la deuxième renvoie le résultat de la comparaison entre ses deux paramètres (*lower than*) et incrémente le compteur.

7. Modifier les trois fonctions de tri écrites plus haut pour qu'elles renvoient le nombre de comparaisons effectuées. L'on devra parfois modifier les sous-fonctions pour qu'elles prennent le `Counter` comme paramètre supplémentaire.

8. Le script partagé contient une fonction générant une liste d'entiers aléatoire de longueur `n`, avec un taux de doublons de `doub` et un un taux de désordre de `des` (ces taux étant compris entre 0 et 1). Tracer l'évolution du nombre de comparaisons pour trier une liste en fonction de `n` pour chacun des algorithmes de tri sur le même graphe. Tracer plusieurs graphes correspondant à différentes valeurs de `des` et `doubl`. L'on pourra tracer les courbes dans un repère semi-logarithmique si cela se justifie.

9. Écrire une fonction `qsort_randp` qui, au lieu de choisir systématiquement le pivot `p=d`, sélectionne aléatoirement `p` selon une loi uniforme sur `range(d, f)`. L'ajouter aux courbes précédentes.

   La complexité dans le pire des cas *du tri rapide est en $O(n^2)$ mais, pour un choix de pivot aléatoire, sa complexité moyenne passe en $O(n \log n)$, ce qui est optimal comme complexité moyenne sur une entrée aléatoire. Le choix d'un pivot optimisant la complexité moyenne dépend de ce que l'on sait des données en entrée, mais il s'agit d'un problème assez complexe.*

Pour aller plus loin

10. Réécrire la fonction `qsort` *sans appel récursif*, c'est-à-dire sans que le code de l'une des fonctions appelées ne contienne son propre nom. L'on pourra pour cela utiliser une *pile* stockant les indices de début et de fin des opérations à effectuer, c'est-à-dire une liste initialement vide (`stack = []`) à laquelle on ajoute (`stack.append`) ou retire (`stack.pop`) un dernier élément.

    L'on peut toujours réécrire une fonction récursive pour en faire disparaître la récursivité à l'aide de piles, mais les fonctions obtenues sont souvent plus compliquées.

11. Implémenter l'algorithme du *tri fusion* (*mergesort*). Cet algorithme, basé sur la méthode *diviser pour régner*, fonctionne de la façon suivante :

    Division Découper la liste en deux sous listes de même taille (ou presque).

    Résolution Trier les deux sous-listes récursivement.

    Combinaison Fusionner les listes triées en une nouvelle liste en parcourant *en même temps* les deux sous-listes triées et en ajoutant dans la nouvelle alternativement le premier élément de l'une et de l'autre des sous-listes.

    Le tri fusion a une complexité moyenne et une complexité dans le pire des cas en $O(n \log n)$*. Il n'est pas fait* en place*, mais la mémoire consommée est au plus quatre fois la taille de la liste initiale. Si l'on gère correctement l'étape* combinaison*, il est possible d'assurer la stabilité du tri. Il n'est pas efficace sur des listes déjà triées.*