I21 - Algorithmique élémentaire
TP 5 - Algorihtmes de tri et de recherche

Année universitaire 2022/23

Exercice 1. Tri bleu/blanc/rouge

Le but est de cet exercice est de visualiser le parcours du tri bleu/blanc/rouge vu en cours. Pour cela il faut:

- copier les fichiers `drapeau.py` et `bbr.py` dans votre répertoire de travail;
- exécuter la commande `python3 drapeau.py` depuis un terminal dont le répertoire courant est votre répertoire de travail;
- cliquer sur `restart` pour générer un nouveau tableau à trier. Le nombre de coups à atteindre est suggérer bien qu'il ne soit pas toujours optimal.

Compléter le fichier `bbr.py` afin de pouvoir utiliser le bouton `tri` qui tri automatiquement le tableau proposé. Le tableau `T` est composé uniquement des chaînes de caractères `"blue"`, `"red"` et `"white"`. On pourra essayer d'améliorer l'algorithme du cours afin de diminuer le nombre de permutations.

Exercice 2. Tri par morceau

À la base des algorithmes de tri les plus efficaces on retrouve toujours une idée simple consistant à trier indépendamment différents sous-tableaux du tableau principal puis à les fusionner.

Le but de cette exercice est de montrer que cette technique permet déjà d'améliorer la complexité des tris standards.

1. Écrire une fonction `fusionner(T1,T2)` qui prend en paramètres deux tableaux d'entiers triés et retourne un tableau `T3` triés contenant les éléments de `T1` et `T2` ainsi que le nombre de comparaisons d'élements effectuées. L'algorithme utilisé doit être de complexité $\Theta(n+m)$ où n et m sont les longueurs respectives des deux talbeaux. Par exemple:

```
>>> fusionner([1,4,6,8,10],[2,3,9,11])
 ([1, 2, 3, 4, 6, 8, 9, 10, 11], 8)
```

2. Écrire une fonction `tri_partiel(T,a,b)` qui tri le tableau `T` entre les indices `a` (inlus) et `b` (exclus) en utilisant la méthode du tri par insertion et retourne le nombre de comparaisons effectuées.

```
>>> T = [2,4,5,3,6,8,5,4,2,7]
>>> tri_partiel(T,2,8)
 12
>>> print(T)
 [2, 4, 3, 4, 5, 5, 6, 8, 2, 7]
```

3. Écrire une fonction `fusion_partielle(T,a,b)` qui fusionne les sous-tableaux `T[:a]` et `T[a:b]` du tableaux `T` suposées triées et recopie le résultat dans le sous-tableau `T[:b]`. La fonction doit de plus retourner le nombre de comparaisons effectuées.

```
>>> T = [2,4,5,3,6,8,5,4,2,7]
>>> fusion_partielle(T,3,6)
 4
>>> print(T)
 [2, 3, 4, 5, 6, 8, 5, 4, 2, 7]
```

4. Écrire une fonction `tri_morceau(T,m)` qui tri le tableau `T` de la façon suivante:
   - trier les m premiers éléments
   - tant qu'ils restent des éléments à trier
     - trier les m éléments suivants
     - fusionner ces m élémentst avec les éléments précédents

   La fonction devra de plus retourner le nombre de comparaisons effectuées.

5. On souhaite évaluer la complexité de cette en fonction de la valeur de m. On fixe la taille des tableaux à $n = 2^{12}$, pour chaque valeur de $m = 2^k, k \in \{1, 2, 3, \ldots, 12\}$ calculer le nombre moyen de comparaisons effectuées pour 10 tableaux d'entiers entre 1 et 1000 générés aléatoirement. En déduire la valeur optimale de m.

Exercice 3. Challenge: Tri alien

1. (**Niveau facile**) On dispose d'un dictionnaire composé de mots écrits dans un alphabet venu d'ailleurs. La seule information dont on dispose est l'alphabet en question, dont les symboles sont rangés par ordre croissant. Écrire une fonction `tri_alien(L,A)` qui tri la liste de mots `L` en fonction de l'ordre des symboles dans la chaine de caractères `A`. Par exemple:

```
>>> L = ['#!', '@!@', '!!^^!', '@#!!^', '!']
>>> A =  '@!#^'
>>> tri_alien(L, A)
>>> print(L)
 ['@!@', '@#!!^', '!', '!!^^!', '#!']
```

   (**Indication**: commencer par écrire une fonction `diff(ch1,ch2)` qui retourne le premier indice où les deux chaines diffèrent.)

2. (**Bonus: Niveau (très) difficile**) On dispose d'une liste de mots triés dans l'ordre lexicographique ainsi que de la liste des symboles du langage alien. Écrire une fonction `ordre_alien(L,A)` qui prend en paramètre la liste de mots triés `L` et l'alphabet `A` est retourne l'alphabet rangé dans l'ordre lexicographique. On fera l'hypothèse que la liste est suffisament grande pour qu'il ne puisse pas y avoir d'ambiguité sur l'ordre des symboles.

```
>>> L = ['@!@', '@#!!^', '@^^!!', '!', '!!^^!', '#!', '###',
         '##^', '^@!@^^', '^^@!@']
>>> S = '!#@^'
>>> A = ordre_dico(S,L)
>>> print(A)
 '@!#^'
```