Feuille de TP n° 8 - Tris

Objectifs de la séance

- Présenter plusieurs méthodes pour trier une liste sur les éléments d'un ensemble ordonné.
- Tester ces programmes, réfléchir au jeu de tests, évaluer la complexité théorique et voir la cohérence avec des tests chronométriques.

Afin de tester les algorithmes de tris étudiés dans ce TP, on désire construire des listes d'entiers ordonnées ou non et de longueurs variables.

Par la suite, on utilisera le générateur de nombre aléatoire `randint` de la librairie Python `random`. On pourra consulter l'aide avec la commande `help(random.randint)`.

Exercice 1.
1. Construire, trois listes `L10`, `L1000` et `L10000` contenant les entiers $[N-1, N-2 \ldots, 0]$ avec $N \in \{10, 1000, 10000\}$ respectivement.
2. Ecrire une fonction `gen_liste(n,N)` permettant de générer aléatoirement une liste de taille n et dont les éléments sont des entiers compris entre 0 et N.
3. Construire, trois listes `A10`, `A1000` et `A10000` de taille $N \in \{10, 1000, 10000\}$ respectivement, qui contiennent des entiers choisis chacun aléatoirement entre 0 et $N-1$.

Il est possible de mesurer le temps d'exécution d'une fonction en utilisant la fonction `time` de la librairie `time`. Pour éviter de polluer le code d'une fonction dont on veut mesurer le temps d'exécution, il est préférable d'exécuter à part ces lignes de commandes :

```
t1 = time()
fonctionaevaluer(args)
print(time() - t1)
```

On présente dans cette section des algorithmes de tris en place. Dans chaque cas, le programme correspondant reçoit une liste (ou un tableau) sur lequel il agit, mais ne retourne rien. De plus, il n'utilise pas de tableau auxiliaires. Ce sont donc des algorithmes à effet de bord.

Exercice 2. Écrire une fonction `echange(L,i,j)` permettant de procéder à l'échange des éléments `L[i]` et `L[j]` dans la liste `L`.
Contrainte : Cette fonction ne doit contenir qu'une seule ligne d'instructions en dehors de la ligne d'introduction et de `return` (`NONE` ici).

Exercice 3 (**Tri par sélection**)**.** Dans le tri par sélection, on cherche le plus petit élément du tableau entré en argument et on le place en position initiale (par exemple par un échange), puis on recommence avec la recherche du plus petit élément parmi ceux restants etc... Voici le pseudo-code de l'algorithme :

Algorithme 1 — Le tri par sélection

Entrées : une liste d'éléments à ordonner
Sorties : rien, mais la liste est ordonnée lorsque l'on sort de la procédure

```
pour i de 0 à |L|-1 faire
    Trouver la position j du minimum des |L| − i derniers éléments du tableau ;
    L[i] ↔ L[j] ;
fin
```

1. Écrire une fonction `select(L)` permettant de trier la liste `L` par cet algorithme. Cette fonction ne renvoie rien (`return` sans argument).
2. Tester le programme sur les listes précédemment définies.
3. En fonction de la longueur n de la liste, quel est le nombre d'échanges d'éléments dans le meilleur des cas ? Dans le pire des cas ? Il s'agit de la *complexité dans le meilleur et le pire des cas.*
4. En testant sur plusieurs listes $A1000$ et $A10000$ avec la fonction `time`, vérifier que la complexité moyenne de cet algorithme est un $O(n^2)$.

Exercice 4 (**Tri par insertion**). Cette fois, l'idée consiste à trier les éléments de la liste à mesure de la lecture de celle-ci. On suppose qu'après la k-ième itération, les k premiers éléments de la liste sont reclassés dans l'ordre. La $k+1$-ième itération consiste à insérer à sa place le $(k+1)$-ième élément parmi les précédents déjà triés.
On demande que cette insertion soit obtenue par échanges successifs avec les termes déjà positionnés (du terme d'indice k jusqu'au terme d'indice 1 éventuellement), de sorte que ce $(k+1)$-ième élément trouve sa place dans le début de liste ordonnée. Ainsi, la recherche de la position et l'insertion peuvent être menées en parallèle.

Algorithme 2 — Le tri par insertion

```
    Entrées : une liste d'éléments à ordonner
    Sorties : rien, mais la liste est ordonnée
1   pour i de 0 à |L| - 1 faire
2   |   pour j de i à 1 faire
3   |   |   si L[j] < L[j − 1] alors
4   |   |   |   L[j − 1] ↔ L[j] ;
5   |   |   fin
6   |   fin
7   fin
```

1. Écrire une fonction `insere(L)` qui prend en argument une liste L, trie cette liste et ne retourne rien.
2. Tester le programme sur les listes précédemment définies, et regarder le temps mis pour trier les listes $A1000$, $A10000$, $L1000$, $L10000$.
3. Quel est le nombre d'échanges d'éléments dans le meilleur des cas ? Dans le pire des cas ?

Remarque : Une version plus performante de l'algorithme consiste à rechercher la place d'insertion avant d'insérer directement le $(k+1)$-ième élément à sa place. Pour cela, on utilise un algorithme optimisé pour déterminer l'indice d'insertion du $(k+1)$-ième élément, c'est-à-dire via une recherche dichotomique entre $L[0]$ et $L[k-1]$.

Exercice 5 (**Tri à bulles**). Voici le pseudo-code de l'algorithme :

Algorithme 3 — Le tri à bulles

```
    Entrées : une liste d'éléments à ordonner
    Sorties : rien, mais la liste est ordonnée
1   pour i de 0 à |L| - 1 faire
2   |   pour j de 0 à |L| - 2 -i faire
3   |   |   si L[j] > L[j + 1] alors
4   |   |   |   L[j + 1] ↔ L[j] ;
5   |   |   fin
6   |   fin
7   fin
```

1. Écrire la fonction `tribulles` prenant en argument une liste d'entiers (ou de réels...) et qui trie la liste dans le corps de la fonction.
2. Quel est le nombre d'échanges d'éléments dans le meilleur des cas ? Dans le pire des cas ?
3. Quel est, expérimentalement, le type de dépendance entre ce temps mesuré et la longueur de la liste lorsque celle-ci est triée à l'envers (le pire des cas, donc) ?
4. Pour une liste `L` de taille n, on dit que deux indices $0 \leqslant i < j < n$ forment une inversion de la liste si `L[i] > L[j]`. Écrire une fonction `compte_inversions(L)` permettant de dénombrer les inversions dans une liste.
5. On veut vérifier expérimentalement que le nombre d'échanges effectués dans le tri à bulles correspond exactement au nombre d'inversions de la liste. Copier le tri à bulles dans une nouvelle fonction `echanges_bulles(L)` de sorte que la fonction modifie une copie de la liste et non la liste initiale et qu'elle renvoie le nombre d'échanges effectués durant le tri.
6. Tester cela sur des listes de différentes tailles.

Le **tri rapide** (Quicksort) est un algorithme de tri très efficace basé sur un principe de "diviser pour mieux régner".
La fonction quicksort est une fonction récursive qui fonctionne comme suit :

1. Si la liste `L` est de taille inférieure ou égale à 1, renvoyer la liste `L`.
2. Choisir un élément pivot dans la liste `L` (on prend un élément tiré au hasard, ou le premier élément).
3. Séparer la liste privée en trois sous-listes `L1`, `L2` et `L3` comprenant respectivement les éléments inférieurs au pivot, les éléments supérieurs au pivot, et les éléments égaux au pivot.
4. Creer une liste `LT` qui est la concaténation de `quicksort(L1)`, de `L3` et de `quicksort(L2)`.
5. Renvoyer `LT`

Exercice 6 (**Tri rapide**)**.**

1. Créer une fonction `Separation` qui prend en entrée une liste `L` et une valeur k, et qui renvoie trois listes `L1`, `L2` et `L3`, contenant tous les éléments de `L` inférieurs à `k`, tous ceux strictement supérieurs à `k` d'autre part, et enfin tous ceux égaux à k.
2. Appliquer l'algorithme de tri rapide à la liste `L = [8, 3, 9, 6, 5]`, en choisissant à chaque fois un élément au hasard comme pivot. On représentera l'arbre de tous les appels récursifs nécessaires.
3. Ecrire une fonction `quicksort` qui effectue le tri rapide d'une liste.
4. Tester l'algorithme sur les listes définies précédemment.

Remarque : La complexité en moyenne du tri rapide est $O(n\ln(n))$. Il est ainsi plus efficace que le tri par sélection et que le tri à bulles.
Cependant, le pire des cas pour le tri rapide se produit lorsque la liste est déjà triée ou presque triée, entraînant une complexité de $O(n^2)$.