Algorithmique avancée

—o000o– Activité TP projet –o000o—

Cette activité s'intéresse à la découverte et à la programmation d'algorithmes de tri classiques qui permettent d'organiser et d'indexer efficacement des collections d'objets pour en faciliter la recherche ultérieure. On s'intéresse dans l'ensemble des exercices à des algorithmes de tri en ordre croissant. Il est évident que ces algorithmes peuvent être utilisés de manière similaire pour trier par ordre décroissant. Le but de l'exercice consiste à :

- implémenter les algorithmes sur la base des structures de données vues en cours ; vous pouvez amener d'autres structures de données plus efficaces, mais il faudra les définir clairement
- découvrir la méthodologie scientifique qui repose, pour les sciences expérimentales, en : la formulation d'une hypothèse, la mise en place d'un protocole de test, l'observation et l'analyse critique des résultats, qui mène à la formulation d'une nouvelle hypothèse. Plus précisément, il est attendu que vous essayiez de mettre en place des protocoles de tests comme indiqué dans la section 5 "Expérimentations".

Ce travail est **totalement optionnel** et ne sera considéré dans votre note que si vous le rendez. Vous pouvez travailler seul ou en binôme. Dans ce cas, merci de préciser qui a fait quoi pour chaque tâche réalisée.

1 Tri par insertion

Il s'agit du tri le plus simple, que nous utilisons de manière intuitive dans notre vie quotidienne : pour classer des cartes dans une main ou des copies dans une pile. On suppose par la suite que les valeurs à trier sont contenues dans une liste chaînée. Le principe consiste pour chaque valeur de la liste, à l'insérer à la bonne position dans la liste résultat. Initialement la liste résultat est vide et le premier élément de la liste initiale est donc inséré sans aucun test. Ensuite, chaque élément est inséré à sa position en fonction de l'ordre choisi (ici croissant).

Indications

1. Avant de commencer, écrire une fonction qui prend en argument un entier positif `n` et retourne une liste chaînée contenant `n` valeurs choisies aléatoirement entre 1 et 100. Cette fonction nous servira en salle machine pour générer les listes à trier et évaluer la performance des algorithmes de tri.

2. Écrire maintenant une fonction qui prend en argument une liste `L` contenant des entiers et triée par ordre croissant ainsi qu'une valeur entière et qui retourne la liste initiale augmentée de la nouvelle valeur et toujours triée par ordre croissant. Proposer une version itérative et une version récursive de cette fonction d'insertion.

3. Écrire maintenant la fonction qui réalise le tri par insertion. Cette fonction prend en argument une liste de nombres et retourne une liste contenant les valeurs de la liste initiale triées par ordre croissant. Pour ce faire, la fonction doit parcourir la liste initiale et insérer à l'aide de la fonction de la question 2, chaque valeur dans la liste résultat (qui est initialement vide).

2 Tri à bulles

Le tri à bulles détermine à chaque itération i, le $i$ème élément le plus grand et le place en fin de liste, juste avant le $i+1$ème plus grand. Le principe de l'algorithme consiste donc à parcourir l'ensemble des couples de valeurs successives et à faire remonter la plus grande des deux en procédant par des échanges successifs lorsque cela est nécessaire. Cette méthode peut être imagée par la remontée de bulles d'air à la surface de l'eau d'où son nom. Pour réaliser un tel algorithme, nous allons découper le problème en deux fonctions principales qui font l'objet des questions suivantes.

Indications

1. Ce type de tri peut demander de parcourir les listes dans les deux sens (selon l'implémentation). Dans ce cas, une liste doublement chaînée peut apporter une solution.

2. Pour construire la liste résultat, nous allons utiliser la technique dite des "accumulateurs" : nous allons ajouter un argument à la fonction contenant son résultat temporaire. Celui-ci sera transmis d'un appel récursif au suivant en faisant évoluer sa valeur de manière coordonnée avec le parcours de la liste initiale. Écrire une fonction qui prend en argument la liste chaînée des valeurs initiales ainsi qu'une liste "résultat" des "i" éléments les plus grands déjà triés. Votre fonction devra réaliser le traitement suivant :

   - si la liste initiale est vide, c'est que l'accumulateur contenant le résultat peut être retourné ;
   - sinon, il faut parcourir la liste initiale et faire remonter, par échanges successifs la plus grande valeur à la fin de la liste ;
   - ensuite il faut couper la liste initiale pour la priver de ce dernier élément, et l'ajouter en début de la liste de résultat avant de rappeler récursivement la fonction.

3. En déduire, une fonction qui prend en argument une liste chaînée de valeurs initiales et retourne cette même liste dont les éléments sont triées par ordre croissant.

Il y a **plusieurs façons** d'arriver au résultat escompté, soit en utilisant un parcours de liste itératif pour la remontée qui permet de couper la fin de liste en une opération, soit en utilisant des fonctions récursives dédiées pour ces parcours. Comme pour les autres algorithmes, vous choisirez l'implémentation qui vous semble la plus adaptée et vous justifierez ce choix.

3 Tri par éclatement-fusion

Le tri par éclatement-fusion est un algorithme rapide et simple de la famille "divide and conquer" (diviser pour régner) qui part du principe que trier une liste revient à fusionner deux sous-listes triées issues de la liste initiale. L'efficacité de la méthode repose sur le fait que la fusion de 2 listes triées peut s'exécuter en temps linéaire avec le nombre de données ($O(n)$).

1. Écrire une fonction qui prend 2 listes chaînées contenant des valeurs entières et triées par ordre croissant et retourne la liste fusionnée, également en ordre croissant. Cette question a déjà été abordée lors des séances de TD/TP, elle ne doit donc pas vous prendre beaucoup de temps.

2. Pour réaliser le tri, nous avons besoin d'une fonction qui permet d'éclater une liste en deux sous-listes de taille équilibrée de façon à accélérer au mieux l'algorithme. Pour ce faire, on se propose ici de découper la liste en 2 sous-listes en fonctions des indices pairs et impairs des valeurs. Écrire une telle fonction qui prend en argument une liste chaînée d'entiers et retourne un tuple de 2 listes chaînées d'entiers contenant respectivement les éléments en positions paire et impaire dans la liste initiale.

   Indications : la liste impaire des éléments d'une liste `L` est égale à la liste paire des éléments de `L.suivant`. Une autre optimisation possible consiste à produire la liste paire et impaire en une seule passe sur la liste. Pour cela, si la liste est vide, retourner `None, None` comme listes paires et impaires. Si L ne possède qu'un élément, retourner une liste possédant l'élément de `L` et `None` et sinon, lorsque L a au moins 2 éléments, calculer le résultat sur L.suivant.suivant et ajouter `L.valeur` et `L.suivant.valeur` aux listes obtenues.

3. Écrire maintenant la fonction de tri par éclatement fusion qui prend en argument une liste d'entiers et la retourne triée par ordre croissant. Cette fonction réalise les traitements suivants :

   - si la liste initiale est vide, alors la fonction retourne la liste vide ;
   - sinon, on éclate la liste initiale, on trie chacune des sous listes obtenues et on retourne la fusion des deux sous-listes triées ainsi obtenues.

4 Tri rapide

Dans cet exercice nous intéressons au tri rapide (“quicksort”) inventé par C.A.R. Hoare en 1961. Le principe de l'algorithme tri-rapide est très similaire à celui du tri par éclatement-fusion précédent. Nous décrivons le tri rapide ci-dessous. Pour trier une liste de nombres réels `L` :

- si la liste `L` est non vide :
  - on choisit le premier élément de `L`, que l'on nomme pivot,
  - on partitionne ensuite le reste de la liste `L` en deux sous-listes `Linf` et `Lsup` telles que :
    - tous les éléments de `Linf` sont inférieurs ou égaux au pivot,
    - tous les éléments de `Lsup` sont strictement supérieurs au pivot
  - on applique l'algorithme de tri rapide aux listes `Linf` et `Lsup`, on obtient ainsi deux listes `L1` et `L2`,
  - pour obtenir le résultat attendu, il faut concaténer la liste `L1` avec la liste réduite au pivot et la liste `L2`,
- enfin, si la liste `L` est vide alors elle est déjà triée.

Les questions suivantes vont vous guider dans les principales étapes de l'algorithme de tri rapide.

1. Écrire une fonction qui réalise la partition d'une liste chaînée d'entiers en fonction d'un pivot. La fonction devra retourner un tuple de listes chaînées contenant la liste des valeurs inférieures ou égales au pivot en premier élément et la liste des éléments supérieurs au pivot en second élément.
2. Écrire maintenant une fonction qui prend en argument un liste chaînée d'entiers et retourne une liste triées par ordre croissant des valeurs de la liste initiale.

5 Expérimentations

Nous nous intéressons ici à la comparaison expérimentale des algorithmes de tri étudiés précédemment. L'objectif est de vous sensibiliser à la notion de complexité d'un algorithme et à son coût en terme de temps de calcul ou de mémoire consommée. Nous nous intéresserons ici au temps de calcul qui sont plus facilement mesurables, même s'il est possible de faire des erreurs de mesure malgré tout car nous ne maîtrisons pas les processus systèmes sur les machines et leur consommation de temps processeur.

Chaque algorithme de tri va être testé sur différentes listes de valeurs possédant différentes difficultés :

- des listes uniformément aléatoires,

- des listes croissantes déjà triées,
- des liste décroissantes triées en ordre inverse de ce qui est souhaité (a priori, le pire des cas).

Pour chaque cas de figure, vous allez programmer des tests avec des tailles de listes allant de 10, 100, 1000 à 10000 éléments. De façon à avoir des mesures fiables pour comparer, dans chaque cas de figure, vous réaliserez 10 tests dans les mêmes conditions expérimentales. À chaque fois, vous mesurerez le temps de calcul en milli-secondes à l'aide du module Python `time.time()` qui permet de récupérer le temps système en milli-secondes. En mesurant le temps juste avant et juste après un algorithme de tri, vous aurez une idée assez "précise" de son temps d'exécution. À partir des 10 valeurs, vous calculerez ensuite une durée moyenne ainsi qu'un écart-type autour de la moyenne qui permettra de caractériser la distribution des valeurs autour de la moyenne (en faisant l'hypothèse d'une distribution normale). Vous pouvez aussi mettre en place des tests statistiques en cas de résultats proches entre vos méthodes pour vérifier si les différences observées sont significatives (t-test par exemple).

- Les questions suivantes sont là pour vous aider à structure votre travail d'expérimentation.

1. Écrire une fonction qui prend en argument une liste de nombres et retourne la moyenne des valeurs.
2. Écrire une fonction qui prend en argument une liste de nombres ainsi que la moyenne des éléments de ce tableau et retourne l'écart-type des valeurs. Pour rappel, l'écart-type non biaisé de valeurs x_i en fonction de leur moyenne $\bar{x}$ est défini comme suit :
$$\delta = \frac{1}{n-1} \times \sqrt{\sum_i (x_i - \bar{x})^2}$$
3. Écrire une fonction qui prend en argument un entier positif `n` et retourne une liste chaînée contenant `n` entiers triés par ordre croissant.
4. Même question que précédemment, mais pour écrire une fonction génère une liste chaînée dont les éléments sont triés par ordre décroissant.
5. Pour chaque algorithme de tri, produire des mesures de temps d'exécution moyen plus écart-type pour des listes aléatoire / croissantes / décroissantes de taille comprise entre 10 et 100000 éléments.
6. En déduire quel algorithme utiliser dans quel cas.