Algorithmes de tri

Informatique #04

Partie A – Pour commencer...

Exercice 1 — recherche par balayage
Écrire une fonction `balayage(x, L)` qui détecte la présence d'un élément `x` dans une liste `L` par balayage.

Exercice 2 — recherche par dichotomie dans une liste triée
Écrire une fonction `dichotomie(x, L)` qui détecte la présence d'un élément `x` dans une liste triée `L` par dichotomie.

Exercice 3 — liste triée dans l'ordre croissant
Écrire une fonction `croissante(L)` qui détecte la présence d'une liste `L` d'entiers ou de flottants triés dans l'ordre croissant.

Exercice 4 — ordre lexicographique
Écrire une fonction `ordre_lex(mot1, mot2)` qui compare deux chaînes de caractères `mot1` et `mot2` au sens de l'ordre lexicographique. Elle retournera en particulier `True` si et seulement si `mot1` $\leqslant$ `mot2`.
On utilisera seulement les opérateurs < et > sur des caractères.

Exercice 5 — calcul de la médiane
Écrire une fonction `mediane(L)` qui retourne la médiane d'une liste d'entiers `L`.
On pourra pour cela utiliser la méthode `sort` sur la liste `L`.

Partie B – Algorithmes de tris usuels

On supposera dans les exercices suivants que les listes à trier sont constituées uniquement d'entiers naturels ou de flottants.

Exercice 6 — tri stupide (bogosort)
Le tri stupide consiste à regarder si une liste `L` donnée est triée. Dans le cas contraire, on mélangera aléatoirement ses éléments jusqu'à obtenir une liste triée. Écrire une fonction `tri_stupide(L)` mettant en œuvre cet algorithme.
On pourra pour cela utiliser la fonction `shuffle` disponible dans la bibliothèque `random`. Ne pas tester la fonction sur une liste qui contiendrait plus de 8 éléments.

Exercice 7 — tri par sélection (selection sort)
Implémenter le tri par sélection étudié en cours.
Rappeler sa complexité dans le pire et le meilleur des cas.

Exercice 8 — tri par insertion (insertion sort)
Implémenter le tri par insertion étudié en cours.
Rappeler sa complexité dans le pire et le meilleur des cas.

Exercice 9 — tri à bulles (bubble sort)
Le tri à bulles consiste à comparer deux à deux et successivement tous les éléments d'une liste donnée et à les permuter dans le cas où le deuxième serait plus petit que le premier.

1. Combien d'échanges sont-ils nécessaires pour trier une liste donnée dans le meilleur/pire des cas?
2. Combien de comparaisons sont-elles nécessaires pour trier une liste donnée dans le meilleur/pire des cas?
3. Écrire une fonction `tri_bulle(L)` mettant en œuvre le tri à bulles.

Exercice 10 — tri rapide (quick sort)

1. Implémenter le tri rapide étudié en cours (tri non nécessairement en place).
2. Écrire une fonction `partition(L, g, d)` qui parcourt les éléments de la liste L dont les indices sont compris entre g (inclus) et d (exclu) et qui place à gauche les éléments inférieurs ou égaux à L[g] et à droite ceux qui lui sont supérieurs. En déduire une implémentation du tri rapide en place.

Exercice 11 — tri fusion (merge sort)

1. Écrire une fonction `fusion(L1, L2)` qui fusionne deux listes triées `L1` et `L2` en une nouvelle liste triée.
2. En déduire une fonction `tri_fusion(L)` mettant en œuvre le tri fusion tel qu'étudié en cours.

Exercice 12 — tri par dénombrement (counting sort) – ENSAM 2014
Soit N un entier naturel non nul. On cherche à trier une liste L d'entiers naturels strictement inférieurs à N.

1. Écrire une fonction comptage, d'arguments L et N, renvoyant une liste P dont le k-ième élément désigne le nombre d'occurences de l'entier k dans la liste L.
2. Utiliser la liste P pour en déduire une fonction tri, d'arguments L et N, renvoyant la liste L triée dans l'ordre croissant.
3. Tester la fonction tri sur une liste de 20 entiers inférieurs ou égaux à 5, tirés aléatoirement.
4. Quelle est la complexité temporelle de cet algorithme? La comparer à la complexité d'un tri par insertion ou d'un tri fusion.

Exercice 13 — tri par paquets (bucket sort)
On suppose que tous les éléments de la liste `L` à trier sont dans l'intervalle $I = [a, b[$. Le tri par paquets consiste à découper l'intervalle I en `len(L)` sous-intervalles de même longueur puis à répartir les données en paquets correspondant à chacun de ces sous-intervalles. On triera alors chacun de ces paquets à l'aide d'un des algorithmes de tri précédemment implémentés.

1. Quel intérêt présente cet algorithme?
2. Si on note I_k le k^{e} sous-intervalle de I, comment déterminer dans quel intervalle se situe le réel $x \in [a, b[$.
3. Écrire une fonction `tri_paquets(L)` implémentant l'algorithme présenté ci-dessus.

Exercice 14 — tri baquets ou tri par base (radix sort) (♣)
Dans la description de l'algorithme du tri par baquets, on appelle chiffre de rang $r \geqslant 1$ d'un entier positif ou nul p le r-ième chiffre en partant de la droite de l'entier p écrit dans la base 10; par définition, si r est supérieur au nombre total de chiffres de p, le chiffre de rang r vaut alors 0.

Par exemple, pour l'entier $p = 597$, le chiffre de rang 1 est 7, celui de rang 2 est 9, celui de rang 3 est 5, et le chiffre de rang i pour $i \geqslant 4$ est 0.

Les données à trier sont dans une liste `L`. Pour trier la liste `L`, on dispose d'une liste appelée `baquets` constituée de 10 sous-listes. On note N le nombre de chiffres maximum des entiers constituant la liste `L`.
L'algorithme est le suivant :

> Pour r qui varie de 1 à N, faire :
> - vider les dix baquets;
> - considérer successivement les entiers de la liste `L` : en notant p l'entier considéré, déterminer le chiffre de rang r de l'entier p; en notant k ce chiffre, ajouter p à la liste `baquets[k]` ;
> - vider `L` et concaténer dans `L` les dix baquets.

Écrire une fonction `tri_baquets(L)` mettant en œuvre cet algorithme. On écrira au préalable une fonction `chiffre(p, r)` qui renvoie le chiffre de rang r de l'entier p (lu de droite à gauche).