TD D'INFORMATIQUE N°4 (PYTHON)

Mise en œuvre des algorithmes de tri

Les exercices marqués d'une † sont des reprises, parfois légèrement reformulées, du *problème du sandwich au jambon* (X – MP/PC, 2012).

Partie I — Tris par sélection

Exercice 1 — En utilisant la fonction `randint` la librairie `random`, écrire une fonction `TableauAléatoire(n)` qui renvoie un tableau constitué d'entiers aléatoires entre -1000 et 1000. On utilisera dans la suite cette fonction de deux manières : avec des petits tableaux (disons une dizaine de cases) pour tester empiriquement la correction des algorithmes écrits, et avec des grands tableaux (une dizaine de milliers de cases) pour tester leurs performances.

†Exercice 2 — Extremums

a) Écrire et tester en Python les fonctions `Maximum(t)`, `Minimum(t)`, `IndiceMaximum(t)`, `IndiceMinimum(t)`, `IndiceMaximumEntreBornes(t, a, b)` et `IndiceMinimumEntreBornes(t, a, b)`. Toutes ces fonctions s'appliquent à un tableau t, les deux dernières prennent deux arguments supplémentaires qui sont des indices.

b) Encore une variante : écrire les fonctions `EstMinimum(t, x)` et `EstMinorant(t, x)` qui renvoient un booléen, indiquant si x est un minimum ou un minorant, respectivement, du tableau t.

Exercice 3 — Écrire une fonction `SontSéparés(t1, t2)` qui renvoie `True` s'il existe un majorant de t_1 qui minore t_2, ou bien s'il existe un minorant de t_1 qui majore t_2, et `False` dans tous les autres cas.

Exercice 4 — Écrire le tri par sélection en deux versions : `TriSélectionMin(t)` qui cherche les minimums et les ordonne progressivement sur la gauche du tableau, et `TriSélectionMax(t)` qui cherche les maximums et les ordonne sur la droite.

Exercice 5 — Chronométrage et complexité temporelle empirique

En utilisant la fonction `ComplexitéEmpirique(tri)` fournie, qui prend en argument le nom d'une fonction effectuant le tri d'un *tableau*, tester les deux versions du tri par sélection de l'exercice précédent.

Exercice 6 — Recherche simultanée du maximum et du minimum

a) Écrire une fonction `IndicesExtremumsEntreBornes(t, a, b)` qui renvoie un couple (i, j) tel que t_i soit la *première* occurrence du minimum et t_j soit la *dernière* occurrence du maximum dans le sous-tableau $[t_a\,;\,t_{a+1}\,;\,\ldots\,;\,t_{b-1}\,;\,t_b]$.

b) En déduire la variante `TriSélectionMinMax(t)` qui place simultanément les minimums à gauche et les maximums à droite pour ranger le tableau t dans l'ordre croissant.

c) Que gagne-t-on, en nombre de comparaisons effectuées, entre cette version et les versions classiques du tri par sélection ? Que donne la mesure empirique de la complexité ?

Exercice 7 — Recherche du k-ième élément (première version)

Adapter l'algorithme du tri par sélection pour obtenir le k-ème plus petit élément d'un tableau en temps $\Theta(n \times \max\{k, n-k\})$. Cet algorithme sera réservé aux situations où n est petit, ou bien où l'on a $k = o(\ln n)$ ou $n - k = o(\ln n)$. On appellera la procédure `RechercheKième1(t, k)`.

†Exercice 8 — Recherche du k-ième élément (deuxième version)

a) Écrire une fonction `NombrePlusPetits(t, v)` qui étant donné un tableau t et une valeur v renvoie un couple $(n_1\,;\,n_2)$ où n_1 est le nombre d'éléments de t strictement inférieurs à v et n_2 est le nombre d'éléments égaux à v.

b) Écrire une fonction `PlusGrandInférieur(t, v)` qui étant donné un tableau t et une valeur v renvoie le plus grand élément de t strictement inférieur à v.

c) On souhaite trouver le k-ième plus petit élément du tableau t de la manière suivante. Appelons a_0 et b_0 respectivement son minimum et son maximum ; on construit récursivement deux suites $(a_n)_{n\geq 0}$ et $(b_n)_{n\geq 0}$ de la manière suivante. Supposons avoir construit a_n et b_n pour un certain entier n, on pose alors $v_n = (a_n+b_n)/2$ et on note k_n le nombre d'éléments de t strictement inférieurs à v_n et k'_n le nombre d'éléments égaux à v_n. Si $k_n = k$, l'élément recherché est le plus grand élément de t qui est inférieur ou égal à v_n et si $k_n < k \leq k_n + k'_n$, l'élément recherché est v_n lui-même. Sinon on procède par dichotomie : si $k_n > k$ alors on pose $a_{n+1} = a_n$ et $b_{n+1} = v_n$, tandis que si $k_n < k$ on pose $a_{n+1} = v_n$ et $b_{n+1} = b_n$. Programmer cette fonction sous le nom `RechercheKième2(t, k)`.

d) En remarquant que la quantité $b_n - a_n$ de la question précédente reste positive ou nulle et tend vers 0, justifier que l'algorithme décrit termine. On distinguera le cas où le k-ième plus petit élément est différent de tous les autres éléments de t, et le cas où sa valeur apparaît plusieurs fois (et où il est donc égal soit au $(k-1)$-ième plus petit, soit au $(k+1)$-ème plus petit).

e) Évaluer la complexité de la fonction précédente. Bien que moins performante, en moyenne, que les fonctions de recherche du k-ième plus petit élément en $\Theta(n)$, quel est son énorme avantage ?

Partie II — Tris par insertion

Exercice 9 — Écrire la procédure `Insérer(t, i)` qui étant donné un tableau t dont les éléments d'indices strictement inférieurs à i sont rangés dans l'ordre croissant effectue les permutations nécessaires sur le sous-tableau $[t_0\,;\,\ldots\,;\,t_i]$ pour qu'il soit trié.

Exercice 10 — Recherche dichotomique de la position d'insertion

a) Écrire une procédure `PositionInsertion(t, i)` qui étant donné un tableau t dont les éléments d'indices strictement inférieurs à i sont rangés dans l'ordre croissant, détermine la position j qu'occupera la valeur t_i à l'issue de l'exécution de la procédure `Insérer(t, i)`.

b) Montrer que le calcul précédent peut se faire en temps $\Theta(\ln i)$.

c) En déduire la variante `InsérerRechercheDichotomique(t, i)` de la procédure `Insérer(t, i)` de l'exercice précédent.

d) Calculer les complexités en temps de ces deux procédures. On évaluera le nombre $\mathscr{C}(n)$ de comparaisons effectuées et le nombre $\mathscr{D}(n)$ de déplacement d'éléments dans t, où n désigne la longueur de t.

Exercice 11 — Programmer le tri par insertion, sans la recherche dichotomique. On nommera la procédure `TriInsertion(t)` dans la suite du TD.

Exercice 12 — Cas d'un tableau presque trié

a) Déterminer la complexité en temps de `TriInsertion(t)` pour un tableau t déjà trié (on rappelle qu'on a retenu la version *sans* la recherche dichotomique de la position d'insertion).

b) Proposer une procédure `MélangerUnPeu(t)` qui déplace environ $1/10$ de ses éléments.

c) Chronométrer le temps mis par `TriInsertion(t)` pour trier un tableau de longueur 10 000, d'abord avec un tableau quelconque, puis avec un tableau presque trié.

Exercice 13 — Algorithme de Shell (1959)

a) Écrire une procédure `TriInsertionPartiel(t, x0, k)` qui applique le tri par insertion sur le sous-tableau $[t_{x_0}\,;\,t_{x_0+k}\,;\,t_{x_0+2k}\,;\,\ldots\,;\,t_{x_0+\mathrm{K}}]$ où K est le plus grand nombre tel que $x_0 + \mathrm{K}$ n'excède pas le plus

grand indice du tableau. Attention, les éléments ne sont pas dans des cases consécutives, on aura donc besoin d'une nouvelle procédure `InsertionPartielle(t, x0, k, i)`.

b) L'algorithme imaginé par Donald Shell consiste à effectué le tri par insertion sur les sous-tableaux de la forme $[t_{x_0} ; t_{x_0+k} ; t_{x_0+2k} ; \dots ; t_{x_0+\mathrm{K}}]$, pour x_0 entre 0 et $k-1$, en faisant varier le pas k de manière décroissante, jusqu'à $k = 1$ (où l'on retrouve le tri insertion standard). Comme valeurs de k, on choisit les termes de la suite définie par $k_0 = 1$ et la relation de récurrence $k_{n+1} = 3k_n + 1$; on part du plus grand terme k_N dont la valeur ne dépasse pas la taille du tableau, et on les utilise dans l'ordre décroissant jusqu'à $k_0 = 1$. Programmer cet algorithme sous le nom `TriShell(t)`.

c) Tester. Est-il plus rapide que le tri insertion ?

Partie III — Tri rapide (Tony Hoare, 1961)

†Exercice 14 — Pivotages

a) Écrire une fonction `Pivotage(t, a, b, IndicePivot)` qui étant donnés un tableau t et trois indices a, b et i_P tels que $a \leq i_\mathrm{P} \leq b$ permute les éléments du sous-tableau $[t_a ; t_{a+1} ; \dots ; t_{b-1} ; t_b]$ de sorte à n'avoir à gauche de t_{i_P} (le *pivot*) que des éléments qui ne sont pas plus grands que lui et à sa droite que des éléments qui lui sont strictement supérieurs. La fonction renverra l'indice de la nouvelle case où se trouve le pivot à l'issue de la manœuvre.

b) Écrire une fonction `PivotageDeLuxe(t, a, b, p)` qui prend en argument un tableau t, une valeur p (la *pivotte*), et qui permute les éléments du sous-tableau $[t_a ; t_{a+1} ; \dots ; t_{b-1} ; t_b]$ de sorte que les éléments inférieurs à p soient le plus à gauche, les éléments strictement supérieurs le plus à droite. La fonction renverra un couple (i, j) tel que t_i est l'élément d'indice le plus élevé (dans le sous-tableau) parmi ceux qui sont strictement inférieurs à p, et t_j est l'élément d'indice le plus faible (toujours dans le sous-tableau) parmi ceux qui sont strictement plus grands. On remarquera qu'à aucun moment on a supposé $p \in t$.

Exercice 15 — En déduire la procédure `TriRapide(t)`. On écrira pour commencer une procédure récursive `TriRapideAuxilliaire(t, a, b)` qui réalise le pivotage autour de t_a, avant de s'appeler sur les deux-sous tableaux qu'elle créé. On n'effectuera un appel récursif que si le sous-tableau concerné est non vide (pourquoi ?).

†Exercice 16 — Situations défavorables

a) Évaluer le nombre d'opérations effectuées par le tri rapide sur un tableau trié.

b) Chronométrer le tri rapide sur des tableaux de longueur 10 000 : un tableau trié, un tableau presque trié, et un tableau aléatoire.

Exercice 17 — Randomisation

Cela consiste, avant chaque appel à `TriRapideAuxilliaire(t, a, b)` à choisir un indice au hasard entre a et b et à l'échanger avec l'élément d'indice a. De cette façon, on a peu de chances, à chaque étape, de choisir comme pivot un élément proche des valeurs extrêmes. Programmer ainsi la procédure `TriRapideRandomisé(t)`, et la tester sur un tableau trié et sur un tableau non trié.

†Exercice 18 — Application de la méthode au calcul du k-ième élément (troisième version)

a) Modifier l'algorithme du tri rapide en une procédure `RechercheKième3(t, k)`.

b) Évaluer la complexité, dans le pire des cas, de cet algorithme.

c) Adapter le principe de randomisation à l'algorithme `RechercheKième3(t, k)` pour avoir une nouvelle procédure `RechercheKième4(t, k)`. Quelle complexité temporelle peut-on espérer obtenir ainsi ?

d) Chronométrer les deux versions sur des tableaux aléatoires d'une part, sur des tableaux triés d'autre part.

Partie IV — Tri par fusion (John von Neumann, 1945)

Exercice 19 — Fusion de deux tableaux triés

Écrire une fonction `Fusionner(t1, t2)` qui étant donnés deux tableaux triés t_1 et t_2 renvoie en temps $\Theta(n_1 + n_2)$ un tableau trié t contenant, avec répétitions, tous les éléments apparaissant dans t_1 et t_2. Les quantités n_1 et n_2 désignent les longueurs respectives des tableaux t_1 et t_2.

Exercice 20 — Déduire de l'exercice précédent la procédure récursive `TriFusion(t)`, et évaluer sa complexité en temps.

Exercice 21 — Utilisation des listes chaînées

Définissons les listes chaînés (d'objets quelconques) en Python de la manière suivante :

```
LV = []

def liste(t, q) :
    return (t, q)

def tête(L) :
    (t, q) = L
    return t

def queue(L) :
    (t, q) = L
    return q
```

Adapter les fonctions des deux exercices précédents pour qu'elles s'appliquent à des listes chaînées.