PARIS DIDEROT université

EA4 – Éléments d'algorithmique

Examen de 2e session – 25 juin 2014

Durée : 3 heures

Aucun document autorisé excepté une feuille A4 manuscrite
Appareils électroniques éteints et rangés

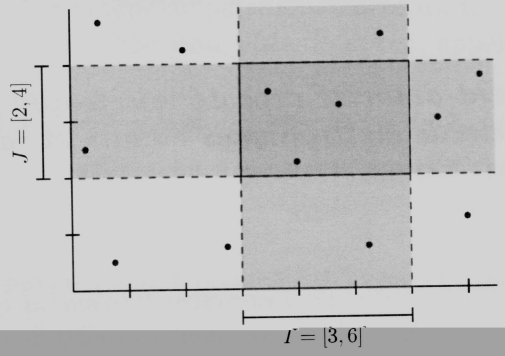
Préliminaires. *Ce sujet est constitué de 4 exercices totalement indépendants qui peuvent être traités dans l'ordre de votre choix. Il est donc vivement conseillé de lire l'intégralité du sujet avant de commencer. Le sujet est long, le barème en tiendra compte. Il est bien entendu préférable de ne faire qu'une partie du sujet correctement plutôt que de tout bâcler.*

Pour tous les exercices demandant d'écrire des algorithmes, vous êtes libres du langage utilisé, du moment que la description est suffisamment précise : python, pseudo-code, français, java... Veillez tout de même à préserver la lisibilité. Pour cela, il pourra être judicieux de faire appel à des fonctions auxiliaires au nom évocateur pour réaliser certaines opérations simples sans entrer dans les détails d'implémentation.

Quel que soit le langage choisi, vous pouvez utiliser comme structures de base, sans redéfinir leur fonctionnement, les tableaux et les listes chaînées, ainsi que des arbres binaires pour lesquels les champs ou fonctions `fils_gauche`, `fils_droit`, `pere`, `valeur`, `est_vide` seront supposés

Exercice 2 : recherche par plage

On s'intéresse au problème suivant : soit $\mathcal{P}$ un ensemble (fini) de points du plan, et deux intervalles I et J ; déterminer le nombre de points de $\mathcal{P}$ situés dans le rectangle $I \times J$. Par exemple, le rectangle $[3,6] \times [2,4]$ de la figure ci-dessous contient 3 points.



Dans un premier temps, on considère le problème analogue en dimension 1 : $\mathcal{P}$ est un ensemble de nombres, I un intervalle, et on cherche à compter les points de $\mathcal{P}$ appartenant à I. On suppose pour simplifier que toutes les valeurs sont distinctes.

1. Décrire un algorithme simple `compteIntervalle(P, a, b)` permettant de résoudre ce problème si $\mathcal{P}$ est représenté par un tableau `P` (non trié) et I par un couple `(a, b)`.
2. De quelle(s) opération(s) élémentaire(s) est-il raisonnable de tenir compte pour évaluer la

7. Quelle est la hauteur minimale d'un $2d$-arbre contenant n points ? Pour obtenir un $2d$-arbre de hauteur minimale à partir de `P`, quel point p_0 faut-il placer à la racine ? Quels points doivent être à la racine de son sous-arbre gauche et de son sous-arbre droit ?
8. Décrire un algorithme du cours permettant de déterminer p_0. Rappeler sa complexité (au pire et en moyenne).
9. Décrire un algorithme `construire2Darbre(P)` permettant de construire un $2d$-arbre de hauteur minimale à partir d'un tableau de points `P` et déterminer sa complexité.
10. Décrire un algorithme `comptePoints(A, a, b, c, d)` qui compte les points d'un $2d$-arbre `A` situés dans le rectangle $[a, b] \times [c, d]$.
11. Quelle est la complexité de cet algorithme dans le pire des cas ?

On peut en fait montrer que, si l'intersection de P et du rectangle contient k points, la complexité de l'algorithme qui vous est demandé est en $O((k+1)\log n)$.

Exercice 3 : redimensionnement de table de hachage

On considère une table de hachage `H` de longueur m à adressage fermé et résolution des collisions par chaînage. Pour simplifier, on suppose que les éléments stockés dans `H` sont des entiers. On note $h : \mathbb{N} \to \mathbb{N}$ la fonction de hachage utilisée, supposée assurer un hachage uniforme.

1. Rappeler l'algorithme de recherche d'un élément `x` dans `H`. Quelle est sa complexité au pire ?
2. On suppose que `H` contient n éléments. Rappeler, sans preuve, la complexité en moyenne de l'algorithme précédent si m est grand par rapport à n.
3. Que se passe-t-il quand n/m augmente ?

On suppose que n est jugé trop grand par rapport à m, et on souhaite redimensionner la table.

4. Écrire un algorithme permettant de remplir une nouvelle table `H2` de longueur $2m$ avec les éléments présents dans `H` *(on utilisera toujours la fonction h)*.
5. Quelle est la complexité de cet algorithme ?

Exercice 4 : tri flou d'intervalles, *bis*

On considère un tableau `T` de données à trier dont la valeur n'est pas connue de manière exacte, mais seulement par un encadrement : `T` est donc un tableau de couples représentant des intervalles `T[i]` $= [$`T[i][0]`$,$ `T[i][1]`$]$ tels que le i^{e} élément x_i de l'ensemble appartient à `T[i]` :

$$\texttt{T[i][0]} \leqslant x_i \leqslant \texttt{T[i][1]}.$$

Le *tri flou* d'un tel tableau de longueur n consiste à réordonner les éléments de `T` en un tableau `S` tel qu'il existe $(\sigma_0, \sigma_1, \ldots, \sigma_{n-1})$ vérifiant :

3. Que peut-on dire lorsque certains intervalles à trier ont une intersection non vide ?

On souhaite tirer parti de cette propriété pour obtenir un algorithme de tri flou, basé sur le tri rapide (QUICKSORT), mais de complexité moindre lorsque les instances du problème sont plus simples. Pour cela, on va modifier l'étape de partitionnement du tri rapide. Une fois l'intervalle `pivot` choisi, on définit un sous-intervalle ***non vide*** de `pivot`, appelé `chevauchement`, ayant la propriété suivante :

`chevauchement` est égal à l'intersection des intervalles de `T` qui l'intersectent. (*)

En particulier, si un intervalle `I` appartenant à `T` ne contient pas `chevauchement`, alors ils sont disjoints.

4. Quels sont les intervalles `chevauchement` possibles si `T` contient les intervalles de l'exemple de la question 1, et si `pivot`$= [6, 13]$?

L'ensemble `T` d'intervalles est alors partitionné en trois sous-ensembles :
- les `petits` intervalles, dont tous les éléments sont plus petits que ceux de `chevauchement`,
- les intervalles `moyens`, qui intersectent (et donc chevauchent) `chevauchement`,
- les `grands` intervalles, dont tous les éléments sont plus grands que ceux de `chevauchement`.

Dans un premier temps, nous allons décrire un algorithme de tri flou en admettant l'existence de `chevauchement`. Plus précisément, on supposera que `trouve_chevauchement(T, pivot)` calcule un tel intervalle en temps linéaire en la longueur de `T`.

Pour simplifier, on construira le tableau trié dans un nouveau tableau.

5. Écrire une fonction `compare(I, J)` qui renvoie -1, 0 ou 1 selon la position de l'intervalle `I` par rapport à l'intervalle `J` :
   - -1 si tous les éléments de `I` sont strictement plus petits que ceux de `J`,
   - 0 si `I` et `J` s'intersectent,
   - 1 si tous les éléments de `I` sont strictement plus grands que ceux de `J`.
6. Écrire la fonction `partition(T, chevauchement)` qui partitionne le tableau `T` par rapport à un intervalle `chevauchement` ayant la propriété (*).
7. Décrire un algorithme de type QUICKSORT réalisant le tri flou d'un tableau d'intervalles.
8. Quelle est sa complexité dans le pire des cas ?
9. Quelle est sa complexité si les intervalles à trier ont une intersection non vide ?
10. En vous basant sur la complexité en moyenne de QUICKSORT, borner la complexité en moyenne de votre algorithme.

Nous allons maintenant écrire la fonction `trouve_chevauchement(T, pivot)`