I Autour de la dichotomie (E3A 2019)

Voici une tentative de code d'une fonction C pour tester par dichotomie si un entier `e` se trouve dans un tableau d'entiers `T` :

```
bool dicho(int e, int* T, int n) {
        // Précondition : n >= 1 et T est un tableau de n entiers trié
        // dans l'ordre croissant
        int g = 0, d = n - 1;
        while (d - g > 0) {
            int m = (g + d) / 2;
            if (T[m] >= e) d = m;
            else g = m;
        }
        return T[g] == e;
    }
```

1. Pour quelle raison ne remplace-t-on pas la précondition par un appel à une fonction qui trierait `T` dans l'ordre croissant ?
2. Montrer que la fonction `dicho` ne termine pas forcément.
3. Indiquer sans justification la ou les corrections à apporter pour que la fonction `dicho` termine, tout en restant correcte.
4. Justifier que la fonction corrigée termine et est correcte.

Une première extension consiste à réduire le problème non pas en 2 mais en 3 : c'est le principe de trichotomie.

5. Écrire une fonction `bool tricho(int e, int* T, int n)` qui renvoie `true` si l'élément `e` se trouve dans `T` et `false` sinon.
6. Estimer la complexité de `tricho(e, T, n)` en fonction de `n`. Comparer avec la méthode par dichotomie.

Une seconde extension consiste à adapter le principe de dichotomie au cas d'une matrice d'entiers dont les éléments sont triés en colonne de haut en bas et de gauche à droite. L'illustration ci-dessous indique l'ordre des éléments d'une matrice à 4 lignes et 6 colonnes :

$$\begin{pmatrix} 0 & 4 & 8 & 12 & 16 & 20 \\ 1 & 5 & 9 & 13 & 17 & 21 \\ 2 & 6 & 10 & 14 & 18 & 22 \\ 3 & 7 & 11 & 15 & 19 & 23 \end{pmatrix}$$

7. Expliquer comment déterminer si un entier `e` se trouve dans une telle matrice de taille $n \times p$, en complexité logarithmique en $\max(n, p)$.

II Conversion d'arbre binaire de recherche en tas

On définit un arbre binaire par : `type 'a arb = V | N of 'a * 'a arb * 'a arb;;`

1. Rappeler la définition d'une file de priorité min, d'un tas min et d'une implémentation par un tableau. Le tableau est-il forcement trié ?
2. Écrire une fonction `infixe : 'a arb -> 'a list`, si possible en complexité linéaire, renvoyant la liste des sommets d'un arbre parcourus dans l'ordre infixe.
3. Rappeler la définition d'un arbre binaire de recherche (ABR). Montrer qu'un arbre binaire est un ABR si et seulement si son parcours infixe est croissant.
4. Écrire une fonction `tas_of_abr : 'a arb -> 'a array` telle que, si `a` est un arbre binaire de recherche à n éléments, `tas_of_abr a` renvoie, en $O(n)$, un tableau représentant un tas min avec les mêmes éléments que `a`.
5. Comment peut-on implémenter les fonctions `take` (pour renvoyer et supprimer la racine d'un tas) et `add` (pour ajouter un élément à un tas) sur un tas min représenté par un tableau ? Avec quelle complexité ?
6. En déduire une fonction `abr_of_tas : 'a array -> 'a arb` en $O(n \log n)$ telle que, si `t` représente un tas min à n éléments, `abr_of_tas t` renvoie un arbre binaire de recherche presque complet et avec les mêmes éléments que `t`.
7. Peut-on faire mieux que $O(n \log n)$ pour la question précédente ?

III Plus courts chemins et Dijkstra

On considère un graphe orienté dont les sommets sont les entiers de 0 à 5. Chaque arc est muni d'un poids entier strictement positif. Le graphe est représenté par listes d'adjacence : une paire `(v, p)` représente un arc vers `v` de poids `p`.

```
let g = [|
    [(1, 4); (2, 2)];           (* sommet 0 *)
    [(3, 5)];                   (* sommet 1 *)
    [(1, 1); (3, 4); (4, 5)];   (* sommet 2 *)
    [(4, 1); (5, 3)];           (* sommet 3 *)
    [(5, 1)];                   (* sommet 4 *)
    []                           (* sommet 5 *)
|]
```

On note $\delta(s, v)$ la longueur d'un plus court chemin du sommet s au sommet v, avec $+\infty$ lorsqu'il n'existe aucun tel chemin.

1. Exécuter à la main l'algorithme de Dijkstra à partir du sommet 0. Donner l'ordre dans lequel les sommets deviennent vus, les distances obtenues et un plus court chemin de 0 à 5.
2. Énoncer un invariant de l'algorithme de Dijkstra et démontrer sa correction.
3. Écrire une fonction `dijkstra : (int * int) list array -> int -> int array` qui renvoie les distances depuis une source. On utilisera `max_int` pour représenter $+\infty$ et on cherchera naïvement, à chaque étape, le sommet non vu d'estimation minimale.
4. Déterminer la complexité de cette implémentation pour un graphe à n sommets et m arcs. Comment l'améliorer pour un graphe peu dense ?
5. On souhaite également compter le nombre de plus courts chemins de s à chaque sommet. Expliquer comment modifier les mises à jour, puis donner ces nombres pour le graphe de l'énoncé avec $s = 0$.
6. Pourquoi l'hypothèse de positivité des poids est-elle importante ? Donner un exemple où Dijkstra échoue en présence d'un poids négatif et indiquer un algorithme qui fonctionne dans ce cas.