Informatique - MPI

Exercice A1

Dans cet exercice, on autorise les doublons dans un arbre binaire de recherche et pour le cas d'égalité on choisira le sous-arbre gauche. On ne cherchera pas à équilibrer les arbres.

Question 1. Rappeler la définition d'un arbre binaire de recherche.

Question 2. Insérer successivement et une à une dans un arbre binaire de recherche initialement vide toutes les lettres du mot $bacddabdbae$ en utilisant l'ordre alphabétique sur les lettres. Quelle est la hauteur de l'arbre ainsi obtenu ?

Question 3. Montrer que le parcours en profondeur infixe d'un arbre binaire de recherche de lettres est un mot dont les lettres sont rangées dans l'ordre croissant. On pourra procéder par induction structurelle.

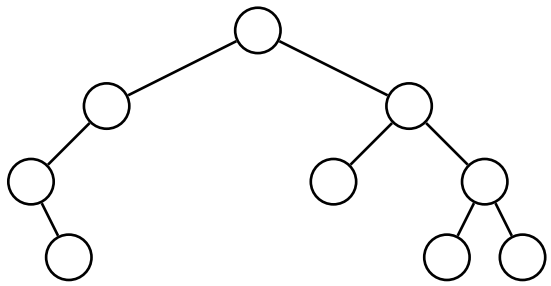
Question 4. Proposer un algorithme qui permet de compter le nombre d'occurrences d'une lettre dans un arbre binaire de recherche de lettres. Quelle est sa complexité ?

Question 5. On souhaite supprimer *une* occurrence d'une lettre donnée dans un arbre binaire de recherche de lettres. Expliquer le principe de l'algorithme permettant de résoudre ce problème et le mettre en oeuvre sur l'arbre obtenu à la question 2 en supprimant successivement une occurrence des lettres e, b, b, c et d. Quelle est sa complexité ?

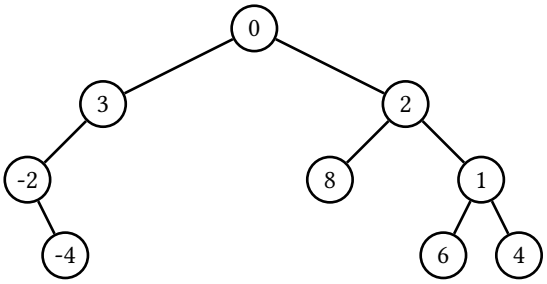
Exercice A2

Dans cet exercice, on considère des arbres binaires étiquetés par des entiers relatifs deux à deux distincts.

Question 1. Un nœud est un *minimum local d'un arbre* si son étiquette est plus petite que celle de son éventuel père et celles de ses éventuels sous-arbres. Les schémas suivants présentent un étiquetage (b) de l'arbre binaire non étiqueté (a).



(a)



(b)

□ **1.1.** Déterminer le ou les minima locaux de l'arbre (b).
□ **1.2.** Donner une définition inductive des arbres binaires ainsi que la définition de la hauteur d'un arbre. Quelle est la hauteur de l'arbre (b) ?
□ **1.3.** Montrer que tout arbre non vide possède un minimum local.
□ **1.4.** Proposer un algorithme qui détermine un minimum local d'un arbre non vide. Quelle est sa complexité ?

Question 2. On souhaite étiqueter un arbre binaire non étiqueté par des entiers relatifs distincts deux à deux de manière à *maximiser le nombre de minima locaux* de cet arbre.
□ **2.1.** Proposer un étiquetage de l'arbre (a) qui maximise le nombre de minima locaux.
□ **2.2.** Proposer un algorithme qui, étant donné un arbre binaire non étiqueté en entrée, calcule le nombre maximal de minima locaux qu'il est possible d'obtenir pour cet arbre. Quelle est sa complexité ?
□ **2.3.** Montrer que, pour un arbre de taille $n \in \mathbb{N}$, le nombre maximal de minima locaux est majoré par $\lfloor (2n+1)/3 \rfloor$.

Exercice B1

Cet exercice est à traiter dans le langage OCaml.

On s'intéresse au *problème SAT* pour une formule en *forme normale conjonctive* (CNF). On se fixe un ensemble $\mathcal{V} = \{v_0, v_1, \dots v_{n-1}\}$ de *variables propositionnelles*. Un *littéral* l_i est une variable propositionnelle v_i ou sa négation $\neg v_i$, représenté en OCaml par un type énuméré : `V i` pour le littéral v_i, `NV i` pour le littéral $\neg v_i$. Une *clause* $c = l_0 \vee l_1 \vee \dots \vee l_{|c|-1}$ est une disjonction de littéraux, représenté en OCaml par un tableau de littéraux. Dans cet exercice, on ne considère que des formules sous CNF, sous forme de conjonctions de clauses $c_0 \wedge c_1 \wedge \dots \wedge c_{m-1}$, représentées en OCaml par une liste de clauses. Une clause peut être vide ou contenir plusieurs fois un même littéral. Une formule sans clause est notée $\top$ et est considérée comme une *tautologie*. Une *valuation* $v : \mathcal{V} \to \{V, F\}$ est représentée en OCaml par un tableau de booléens. Un programme OCaml à compléter est reproduit à la fin de l'exercice.

Question 1. La fonction `initialise : int -> valuation` permet d'initialiser une valuation aléatoire. Implémenter la fonction `evalue : clause -> valuation -> bool` qui vérifie si une clause est satisfaite par une valuation.

Question 2. Étant donné une formule f constituée de m clauses $c_0 \wedge c_1 \wedge \dots \wedge c_{m-1}$ définies sur un ensemble de n variables, la fonction `random_sat` a pour objectif de trouver une valuation qui satisfait la formule, s'il en existe une et qu'elle y arrive. Cette fonction doit effectuer au plus k tentatives et renvoyer un résultat de type `valuation option` avec une valuation qui satisfait la formule passée en paramètre si elle en trouve une et la valeur `None` sinon.
L'idée de ce programme est d'affecter aléatoirement des variables puis de vérifier si chaque clause est satisfaite. Si une clause n'est pas satisfaite, on modifie aléatoirement la valeur associée à un littéral de cette clause, qui devient ainsi satisfaite, puis on recommence.

□ **2.1.** Si cette fonction renvoie `None`, peut-on conclure que f n'est pas satisfiable ? Préciser le type d'algorithme probabiliste.

□ **2.2.** Proposer un jeu de 5 tests élémentaires permettant de tester la correction de ce programme.

□ **2.3.** Ce programme est-il correct par rapport à sa spécification ? Si cela s'avère nécessaire, corriger ce programme pour qu'il remplisse ses objectifs.

Question 3. On s'intéresse maintenant au problème MAX-SAT qui consiste, toujours pour une formule en CNF, à trouver le plus grand nombre de clauses de cette formule simultanément satisfiables. Un algorithme probabiliste naïf fournit une solution approchée en générant aléatoirement k valuations et en retenant celle qui maximise le nombre de clauses satisfaites.

□ **3.1.** Implémenter cet algorithme et le tester. Quelle est sa complexité dans le meilleur et le pire cas ?

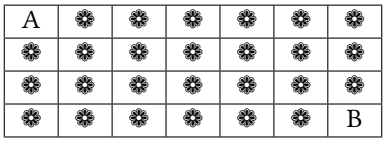
□ **3.2.** Si P ≠ NP, peut-il exister un algorithme de complexité polynomiale pour résoudre MAX-SAT ? Justifier.

```
(* V i : variable propositionnelle d'indice i
   NV i : négation de la variable propositionnelle d'indice i *)
type litteral = V of int | NV of int
type clause = litteral array
type formule = clause list
type valuation = bool array

(* indice : litteral -> int *)
let indice = function V i | NV i -> i

let initialise (n : int) : valuation =
  Array.init n (fun _ -> Random.int 2 = 0)

let evalue (c : clause) (v : valuation) : bool =
  failwith "à compléter"

(* Objectif : prendre en entrée une formule f sur n variables, un nombre maximum
    k strictement positif d'essais et s'évaluer en une option sur une valuation
   qui satisfait f si on en trouve une ou None si on échoue après k tentatives
   *)
let random_sat (f : formule) (n : int) (k : int) : valuation option =
  assert (k >= 1);
  let v = initialise n in
  let rec test (g : formule)  (k : int) : valuation option =
    match g with
    | [] -> Some v
    | c :: cs ->
       if evalue c v then test cs k
       else if k < 1 then None
            else begin
              let i = Random.int (Array.length c) in
              v.(indice c.(i)) <- not v.(indice c.(i));
              test g (k - 1)
            end
  in
  test f k
```

Exercice B2

Cet énoncé est accompagné d'un code compagnon en C dont le contenu est reproduit à la fin de l'énoncé.

Un individu se trouve initialement sur la case A d'une grille rectangulaire de taille $m \times n$. Il doit se rendre sur la case B de cette grille (figure ci-dessous).



Chaque case de la grille, *y compris les cases A et B*, contient un certain nombre de fleurs. Depuis sa case de départ, l'individu connaît le nombre de fleurs présentes dans chaque case. Il se déplace vers B de case en case, les seuls mouvements autorisés étant vers le bas ou vers la droite. À chaque déplacement, il récolte les fleurs de la case atteinte. L'objectif est de récolter le plus de fleurs possible lors du déplacement.

Question 1. On considère la grille suivante.

0 (A)	1	2	3
1	2	3	4
2	3	4	0
3	4	0	1 (B)

Donner le nombre maximal de fleurs cueillies par l'individu.

Question 2. On note $n(i,j)$ le nombre maximum de fleurs que l'individu peut récolter en se déplaçant de A à la case (i,j). Exprimer $n(i,j)$ en fonction de $n(i-1,j)$ et $n(i,j-1)$. En déduire une fonction récursive de prototype `int recolte(int champ[m][n], int i, int j)` qui, étant données les coordonnées `i`, `j` d'une case, calcule le nombre maximum de fleurs cueillies par l'individu de A à la case (`i`, `j`).

Question 3. On suppose $m = n = 4$ et on effectue un appel à `recolte(champ,3,3)` pour résoudre le problème posé. Donner le nombre de fois où votre fonction calcule le nombre de fleurs maximum cueillies dans la case $(1,1)$ (deuxième case de la diagonale).

D'une manière générale, le nombre d'appels à la fonction récursive est important. On a donc intérêt à transformer l'algorithme récursif en algorithme dynamique. On propose de déclarer dans le programme principal un tableau `fleurs` dont la case (i,j) est destinée à contenir la récolte maximale que l'individu peut obtenir en cheminant de A vers la case (i,j).

Question 4. Dans quel ordre remplir le tableau `fleurs` de sorte à éviter de recalculer une valeur ?

Question 5. Écrire une fonction de prototype `int recolte_iterative(int champ[m][n], int i, int j, int fleurs[m][n])` qui calcule, stocke dans `fleurs[i][j]` et renvoie la cueillette maximale obtenue en parcourant le champ de A à la case (`i`, `j`).

Question 6. La fonction `recolte_iterative` permet de déterminer la cueillette maximale en (`i`, `j`) mais ne précise pas le chemin parcouru pour l'obtenir.

□ **6.1.** Écrire la fonction de prototype `void deplacements(int fleurs[m][n], int i, int j)` qui affiche la suite des déplacements effectués par l'individu sur un chemin permettant de récolter le nombre maximum de fleurs entre (0,0) et (`i`, `j`).

□ **6.2.** Insérer un appel de `deplacements` dans la fonction `recolte_iterative` pour afficher le chemin parcouru.

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

/*
La directive #define est utilisée pour définir des valeurs pour
des constantes qui serviront à déclarer des tableaux de taille fixe.
*/
#define m 5
#define n 3

/* Macro de calcul du maximum entre i et j */
int max(int i,int j){
    if (i<j)
        return j;
    else
        return i;
}

int recolte(int champ[m][n], int i, int j){
    /* Code question 2 */
}

void deplacements(int fleurs[m][n], int i, int j){
    /*Code question 6 */
}

int recolte_iterative(int champ[m][n], int i, int j,int fleurs[m][n]){
    /*Code question 5 */
}

int main(){
     int champ[m][n],fleurs[m][n];
     int i,j;
    /* Exemple du champ de fleurs : le nombre de fleurs par case est un entier
    aléatoire entre 0 et 10. On utilise la fonction int rand() de stdlib. Le gén
   érateur
    de nombre pseudo-aléatoires est tout d'abord initialisé.*/
     srand(time(NULL));
     for (i=0;i<m;i++) for(j=0;j<n;j++)
          champ[i][j] = rand() % 11;
     printf("\n Nombre de fleurs maximum cueillies : %d\n",recolte_iterative(
   champ,3,3,fleurs));
     return 0;
}
```