Devoir en temps limité du Vendredi 13 Juin

Dans l'ensemble du devoir, toutes les fonctions seront :

- *écrites en langage OCaml,*
- *précédées d'une explication des variables utilisées,*
- *précédées d'une explication de l'algorithme.*

Exercice 1 (Du cours)

1. Une implémentation en OCaml de la structure de pile d'entier est le type enregistrement suivant :

```
type pile = {mutable contenu : int list} ;;
```

   (a) Définir dans cette situation les primitives d'accès `creer_pile`, `etre_pile_vide`, `empiler` et `depiler`.

   (b) Écrire en OCaml une fonction `insere_pile x p` ayant pour signature

```
insere_pile : int -> pile -> pile
```

   qui place un entier `x` à la bonne place dans une pile `p` triée.

   (c) Écrire en OCaml une fonction `tri_insertion_pile p` ayant pour signature

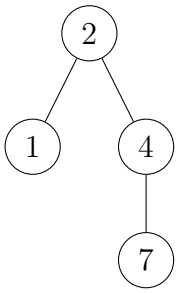
```
tri_insertion_pile : pile -> pile
```

   qui trie une pile `p` en suivant le principe du tri par insertion.

2. Une implémentation en OCaml d'un type d'arbre binaire homogène est le type personnalisé suivant :

```
type 'a arbre =
  | Vide
  | N of 'a * 'a arbre * 'a arbre
;;
```

   (a) Donner les instructions pour définir l'arbre suivant sur OCaml :



   (b) Définir en OCaml une fonction récursive `noeuds` ayant pour signature

```
noeuds : 'a arbre -> int
```

   qui calcule le nombre de noeuds d'un arbre binaire homogène.

   (c) Définir en OCaml une fonction récursive `hauteur` ayant pour signature

```
hauteur : 'a arbre -> int
```

   qui calcule la hauteur d'un arbre binaire homogène.

(d) En testant les deux fonctions précédentes sur différents exemples, on remarque que, pour tout arbre binaire non vide à n noeuds et de hauteur h, on a :

$$h \leqslant n \leqslant 2^h - 1.$$

Démontrer cette propriété par induction structurelle.

(e) Rappeler la définition d'un parcours en profondeur en mode préfixe d'un arbre. On pourra donner un exemple pour illustrer cette définition.

(f) Définir en OCaml une fonction récursive `prefixe` ayant pour signature

```
prefixe : 'a arbre -> 'a list
```

qui renvoie la liste du traitement des étiquettes des sommets dans le cas d'un parcours en profondeur en mode préfixe d'un arbre binaire homogène.

Exercice 2 (Logique et calcul des propositions)

Dans cet exercice, les variables propositionnelles seront notées $x_1, x_2, \ldots$ Les connecteurs propositionnels $\wedge$ (conjonction), $\vee$ (disjonction), $\Rightarrow$ (implication) et $\Leftrightarrow$ (équivalence) seront classiquement utilisés.

De même, la négation d'une variable propositionnelle x_i (respectivement d'une formule $\mathcal{F}$) sera notée $\neg x_i$ (resp. $\neg\mathcal{F}$).

Partie 1 : Définitions

Définition (Minterme, maxterme). Soit $(x_1, \ldots, x_n)$ un ensemble de n variables propositionnelles.

— On appelle minterme toute formule de la forme $y_1 \wedge y_2 \wedge \ldots \wedge y_n$ où pour tout $i \in [\![1, n]\!]$, y_i est un élément de $\{x_i, \neg x_i\}$.

— On appelle maxterme toute formule de la forme $y_1 \vee y_2 \vee \ldots \vee y_n$ où pour tout $i \in [\![1, n]\!]$, y_i est un élément de $\{x_i, \neg x_i\}$.

Les mintermes (respectivement maxtermes) $y_1 \wedge y_2 \wedge \ldots \wedge y_n$ et $y'_1 \wedge y'_2 \wedge \ldots \wedge y'_n$ (resp. $y_1 \vee y_2 \vee \ldots \vee y_n$ et $y'_1 \vee y'_2 \vee \ldots \vee y'_n$) sont considérés identiques si les ensembles $\{y_i \mid i \in [\![1, n]\!]\}$ et $\{y'_i \mid i \in [\![1, n]\!]\}$ le sont.

1. Donner l'ensemble des mintermes et des maxtermes sur l'ensemble (x_1, x_2).

Définition (Formes normales conjonctives et disjonctives). Soit $\mathcal{F}$ une formule propositionnelle qui s'écrit à l'aide de n variables propositionnelles $(x_1, \ldots, x_n)$.

— On appelle forme normale conjonctive de $\mathcal{F}$ toute conjonction de maxtermes logiquement équivalente à $\mathcal{F}$.

— On appelle forme normale disjonctive de $\mathcal{F}$ toute disjonction de mintermes logiquement équivalente à $\mathcal{F}$.

Définition (Système complet). Un ensemble de connecteurs logiques $\mathcal{C}$ est un système complet si toute formule propositionnelle est équivalente à une formule n'utilisant que les connecteurs de $\mathcal{C}$.

Par définition, $\mathcal{C} = \{\neg, \vee, \wedge, \Rightarrow, \Leftrightarrow\}$ est un système complet.

Partie 2 : Le connecteur de Sheffer

On définit le connecteur de Sheffer, ou d'incompatibilité, par $x_1 \diamond x_2 = \neg x_1 \vee \neg x_2$.

2. Construire la table de vérité du connecteur de Sheffer.
3. (a) Exprimer le connecteur de Sheffer en fonction de $\neg$ et $\wedge$.
   (b) Vérifier que $\neg x_1 = x_1 \diamond x_1$.
   (c) En déduire une expression des connecteurs $\wedge$, $\vee$ et $\Rightarrow$ en fonction du connecteur de Sheffer. Justifier en utilisant des équivalences avec les formules propositionnelles classiques.
4. (a) Donner une forme normale conjonctive de la formule $x_1 \diamond x_2$.
   (b) Donner de même une forme normale disjonctive de la formule $x_1 \diamond x_2$.
5. (a) Démontrer par induction sur les formules propositionnelles que l'ensemble de connecteurs $\mathcal{C} = \{\diamond\}$ est un système complet.
   (b) Soit $\mathcal{F}$ la formule propositionnelle $x_1 \vee (\neg x_2 \wedge x_3)$.
   Donner une forme logiquement équivalente de $\mathcal{F}$ utilisant uniquement le connecteur de Sheffer.

Exercice 3 (Sur les permutations)

La notion mathématique de permutation formalise la notion intuitive de réarrangement d'objets discernables. La permutation est une des notions fondamentales de la combinatoire, l'étude des dénombrements et des probabilités discrètes. Elle sert par exemple à étudier sudoku, rubik's cube, etc. Plus généralement, on retrouve la notion de permutation au coeur de certaines théories des mathématiques, comme celles des groupes, des déterminants, de la symétrie, etc.

n désigne un entier naturel supérieur ou égal à 2, E_n l'ensemble des entiers naturels compris entre 1 et n inclus.

Une *permutation* est une bijection de E_n dans lui-même. Une permutation σ de E_n s'écrit mathématiquement sous la forme :

$$\sigma = \begin{pmatrix} 1 & 2 & 3 & 4 & \dots & \dots & \dots & n-2 & n-1 & n \\ \sigma(1) & \sigma(2) & \sigma(3) & \sigma(4) & \dots & \dots & \dots & \sigma(n-2) & \sigma(n-1) & \sigma(n) \end{pmatrix},$$

la seconde ligne contenant donc tous les entiers compris entre 1 et n plus ou moins mélangés.

Par exemple, $\sigma = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 3 & 1 & 4 & 2 \end{pmatrix}$ est une permutation de E_4 et nous pouvons lire que : $\sigma(2) = 1$.

Nous allons représenter alors simplement une permutation σ de E_n par une variable p de type `int array` de longueur $n+1$ de la manière suivante : k appartenant à E_n, $p.(k)$ contient $\sigma(k)$, la case $p.(0)$ étant inutilisée dans la suite.

Par exemple, la permutation $\sigma = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 3 & 1 & 4 & 2 \end{pmatrix}$ est représentée par le tableau

$p =$

0	1	2	3	4
*	**3**	**1**	**4**	**2**

.

Plus généralement, toute fonction f de E_n dans lui-même sera représentée par un tableau de $n+1$ cases défini de la même manière que dans le cas d'une permutation.

Test de permutation

1. f est application de E_n dans lui-même, i un entier naturel, p le tableau de taille $n+1$ représentant la fonction f.

   En remarquant que f est une permutation de E_n si et seulement si, pour tout entier i compris entre 1 et n, i est une valeur prise par f, écrire en OCaml, une fonction itérative `test_permutation_1` qui prend une permutations p de E_n en argument et retourne un booléen égal à VRAI dans le cas où f est une permutation de E_n, un booléen égal à FAUX sinon.

2. La méthode précédente présente l'inconvénient de lire le tableau de nombreuses fois.

   f est application de E_n dans lui-même, i un entier naturel, p le tableau de taille $n+1$ représentant la fonction f.

   Afin de déterminer si f est une permutation ou non de E_n, nous allons utiliser la caractérisation suivante : toute valeur prise par f n'est atteinte qu'au plus une fois.

   Ainsi, après avoir construit un tableau v de booléens de taille $n+1$, toutes les cases de v étant initialisées à FAUX, nous commençons la lecture du tableau p. Pour toute valeur k rencontrée, nous affectons la valeur VRAI à $v[k]$. Et lorsque nous aurons à valider une valeur du tableau v contenant déjà VRAI, f ne sera pas une permutation de E_n .

   Écrire en OCaml, une fonction itérative `test_permutation_2` qui prend une permutation p de E_n en argument et suivant ce schéma, retourne un booléen égal à VRAI dans le cas où f est une permutation de E_n, un booléen égal à FAUX sinon.

Dans la suite du problème, nous manipulons systématiquement des permutations de E_n, sans avoir à le vérifier. Dans cet esprit, nous confondons désormais la permutation et le tableau la représentant.

Composition de permutations

3. Écrire en OCaml, une fonction itérative `ident` qui prend un entier naturel n supérieur ou égal à 2 en argument et renvoie l'identité de E_n .
4. Écrire en OCaml, une fonction itérative `composition` qui prend deux permutations p et q de E_n en arguments et renvoie la composée $p \circ q$, qui est aussi une permutation de E_n.
5. Écrire en OCaml, une fonction `power` qui étant donnés une permutation p de E_n et un entier naturel m, renvoie la permutation $p^m = \underbrace{p \circ p \circ \ldots \circ p}_{m \text{ fois}}$.

   Deux solutions sont attendues, l'une récursive, l'autre itérative.

   La fonction précédente `composition` peut subir quelques adaptations à préciser et être utilisée.

Signature d'une permutation

σ étant une permutation de E_n, on appelle *inversion de* σ, tout couple (i, j) tel que :

$$1 \leqslant i < j \leqslant n \quad \text{et} \quad \sigma(j) < \sigma(i).$$

Rappelons que la signature de σ est égale à "(-1) puissance le nombre d'inversions de σ".

6. Écrire en OCaml, une fonction `signature` qui prend en argument une permutation p de E_n et détermine la signature de la permutation p à l'aide du nombre d'inversions de p.
7. σ étant une permutation de E_n, nous définissons *le code de Lehmer de* σ comme le n-uplet $c = (c_1, c_2, \ldots, c_n)$ où, pour tout i appartenant à $[\![1, n]\!]$, c_i est le nombre d'inversions $(i, j), j \in [\![i+1, n]\!]$, de σ.
   Comme pour les permutations de E_n, le code de Lehmer d'une permutation p de E_n sera représenté par un tableau c de $(n+1)$ cases tel que : $\forall i \in [\![1, n]\!]$, $c.(i) = c_i$.
   (a) Quel est le code de Lehmer de la permutation $\sigma = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 4 & 6 & 1 & 5 & 2 & 3 \end{pmatrix}$?
   (b) Écrire en OCaml, une fonction `code` qui prend en argument une permutation p de E_n et renvoie le code de Lehmer c de la permutation p.
8. L'idée du "décodage" est assez simple... σ étant une permutation de E_n, c son code de Lehmer,
   — c_1 désigne le nombre d'inversions $(1, j), j \in [\![2, n]\!]$, de σ et $\sigma(1)$ est donc le (c_1+1)-ième élément dans la séquence ordonnée $1, 2, \ldots, n$,
   — c_2 désigne le nombre d'inversions $(2, j), j \in [\![3, n]\!]$, de σ et $\sigma(2)$ est donc le (c_2+1)-ième élément dans la séquence ordonnée $1, 2, \ldots, n$ privée de $\sigma(1)$,
   — c_3 désigne le nombre d'inversions $(3, j), j \in [\![4, n]\!]$, de σ et $\sigma(3)$ est donc le (c_3+1)-ième élément dans la séquence ordonnée $1, 2, \ldots, n$ privée de $\sigma(1), \sigma(2)$,
   — etc.
   (a) Quelle est la permutation dont le code de Lehmer est $(2, 0, 3, 1, 0, 0)$?
   (b) Écrire en OCaml, une fonction `decode` qui prend en argument le code de Lehmer `c` d'une permutation `p` de E_n et renvoie la permutation `p`.

Génération des permutations

L'objet de cette question est de construire toutes les permutations de E_n.
On rappelle que $\text{card}(E_n) = n!$ et que l'ensemble des permutations de E_n est totalement ordonnée par l'ordre lexicographique noté $<_{\text{lex}}$.
Pour cet ordre, le minimum de E_n est donc l'identité de E_n et le maximum est la permutation $\sigma_{\max} = \begin{pmatrix} 1 & 2 & \ldots & n-2 & n-1 & n \\ n & n-1 & \ldots & 3 & 2 & 1 \end{pmatrix}$.
A toute permutation σ de $E_n - \{\sigma_{\max}\}$, nous appliquons l'algorithme suivant :
— nous recherchons le plus grand entier i compris entre 1 et $n-1$ tel que :

$$\sigma(i) < \sigma(i+1),$$

— nous recherchons ensuite k compris entre $i+1$ et n tel que :

$$\sigma(k) = \min\{\sigma(j)/j \in [\![i+1, n]\!], \sigma(i) < \sigma(j)\},$$

— nous échangeons alors $\sigma(k)$ et $\sigma(i)$,
— nous trions par ordre croissant (méthode au choix) la séquence des valeurs :

$$\sigma(i+1), \ldots, \sigma(n).$$

Nous avons ainsi construit une nouvelle permutation σ' de E_n.

9. Dans le cas particulier où $n = 3$, appliquer cet algorithme à l'identité sur E_3, puis répéter cet algorithme sur les permutations successivement obtenues tant que possible.

   Vérifier que nous avons ainsi construit toutes les permutations de E_3 et dans l'ordre lexicographique.

10. Soit σ une permutation de E_n distincte de $\sigma_{\max}$. Les notations sont celles utilisées dans la description de l'algorithme ci-dessus.

    Justifier l'existence des entiers i et k .

11. Écrire en OCaml, une fonction `next` qui étant donné une permutation p de E_n distincte de $\sigma_{\max}$, applique l'algorithme décrit ci-dessus à la permutation p, la permutation p est modifiée.

    La fonction `next` comporte donc quatre étapes bien distinctes. Aussi, si l'une des étapes pose problème, vous pouvez la limiter à une phrase comme "trier la séquence des valeurs $p(i+1), \ldots, p(n)$", les autres étapes seront évaluées.

12. Soit σ une permutation de E_n distincte de $\sigma_{\max}$. Les notations sont celles utilisées dans la description de l'algorithme ci-dessus.

    Montrer que : $\sigma <_{\text{lex}} \sigma'$.

    Nous admettrons que σ' est le successeur immédiat de σ dans E_n ordonné par l'ordre lexicographique.

13. Justifier que l'application répétée de notre algorithme à l'identité de E_n génère en un temps fini, l'ensemble des permutations de E_n .

14. Écrire en OCaml, une fonction `list` qui étant donné un entier naturel n, affiche l'ensemble des permutations de E_n .

    Nous disposons d'une fonction `affiche` de signature `'a array -> unit` qui affiche le contenu d'une permutation à l'écran.
