OPTION INFORMATIQUE
Concours blanc

Toutes les fonctions sont à écrire en OCaml. On attachera le plus grand soin à leur lisibilité. On pourra toujours librement utiliser une fonction demandée à une question précédente, même si cette question n'a pas été traitée.

1 Pile avec oubli

On s'intéresse ici à une structure de données de pile avec oubli, similaire à une pile avec capacité, mais dans laquelle un empilement est toujours possible. Dans le cas où on empile une nouvelle valeur alors que la capacité est déjà atteinte, la valeur en fond de pile est retirée en compensation. Cette structure de données est par exemple utilisée par les éditeurs de documents pour stocker les versions d'un fichier et permettre d'annuler les dernières modifications.
Par souci de simplicité, on considère des piles d'entiers. On suppose que la capacité `c` est une variable globale. Les opérations disponibles sont :

- `creer_pile ()` : renvoie une pile avec oubli de capacité c,
- `est_vide p` : détermine si la pile `p` est vide,
- `empiler p v` : modifie la pile `p` en ajoutant la valeur `v` au sommet. Si la capacité est dépassée, la valeur en fond de pile est retirée.
- `depiler p` : modifie la pile `p` en retirant le sommet, et renvoie cette valeur. Si la pile est vide, une erreur est provoquée à la place.

Implémenter ces opérations en Caml à l'aide de tableaux. Chaque opération devra avoir une complexité en $O(1)$. On expliquera sur un exemple la représentation utilisée.

2 Conversion d'une liste en matrice

Écrire une fonction `conversion : int list -> int -> int array array` prenant en argument une liste d'entiers l et un entier n, et renvoyant une matrice $n \times n$, sous forme de tableau de tableaux, définie de la manière suivante :

- On remplit la matrice ligne par ligne avec les éléments de la liste,
- si la liste est trop courte, on complète le reste de la matrice avec des 0,
- si la liste est trop longue, on n'utilise pas la fin.

On pourra écrire `Array.make_matrix p q x` pour créer une matrice de taille $p \times q$ dont toutes les cases sont initialisées à x.

3 Arbres et arités

On considère le type d'arbres suivant :

```
type 'a arbre =
  V
| N of 'a * 'a arbre list;;
```

1. Écrire une fonction `somme : int arbre -> int` prenant en argument un arbre contenant des entiers, et renvoyant la somme de ses noeuds.
2. On définit l'arité d'un noeud de l'arbre comme son nombre de fils **non vides**. Écrire une fonction `arite_max : 'a arbre -> int` prenant en argument un arbre, et renvoyant l'arité maximale de ses noeuds. On conviendra que l'arbre vide a pour arité maximale 0.
3. On dit qu'un arbre est entier si tous ses noeuds qui ne sont pas d'arité 0 ont la même arité. Écrire une fonction `est_entier : 'a arbre -> bool` testant si un arbre est entier.

4 Problème de l'arrêt

Si f est une fonction et x un objet quelconque, on dit que f termine sur x si l'évaluation de l'appel $f\ x$ renvoie un résultat au bout d'un temps fini, ou est interrompue à la suite d'une erreur, par exemple si f et x n'ont pas des types compatibles.

1. (a) Déterminer les entiers pour lesquels la fonction suivante termine :

```
let rec f1 n =
    match n with
         0 -> 0
        |_ -> n + f1(n-2)
```

   (b) Démontrer la terminaison de cette fonction sur cet ensemble d'entiers.

2. Mêmes questions pour la fonction :

```
let f2 n =
    let i = ref n in
    while !i != 10 do
        print_int(!i);
        incr(i);
    done;;
```

3. Écrire une fonction qui ne termine sur aucun entier.

4. L'objectif de cette question est d'examiner la possibilité de décider automatiquement si une fonction f termine sur une entrée x, en analysant le code de f.

   On suppose l'existence d'une fonction `arret` prenant en arguments une chaîne de caractères `code_f` correspondant au code d'une fonction f, ainsi qu'un objet de type quelconque x, et renvoyant `true` si f termine sur x et `false` sinon. On notera en particulier que la fonction `arret` termine dans les deux cas.

   (a) Donner le type de la fonction `arret`.

   (b) Écrire une fonction `bizarre` prenant en argument une chaîne de caractères `code_f` correspondant au code d'une fonction f, ainsi qu'un objet de type quelconque x, telle que `bizarre` termine sur `code_f` et x si et seulement si f ne termine pas sur x. Le résultat renvoyé par `bizarre` quand elle termine n'a aucune importance.

   (c) Écrire une fonction `paradoxe` prenant en argument une chaîne de caractères `code_f` correspondant au code d'une fonction f, et terminant sur `code_f` si et seulement si f ne termine pas sur `code_f`.

   (d) En utilisant `code_paradoxe` le code de la fonction `paradoxe`, exhiber une contradiction et conclure.

5 Recherche de valeur absolument majoritaire

Soit $k \in \mathbb{N}^*$ et T un tableau de longueur N dont toutes les valeurs sont dans $[\![0; k-1]\!]$. On dit que $n \in [\![0; k-1]\!]$ est *élu* par T si le nombre d'occurrences de n dans T est strictement supérieur à $N/2$. Clairement, il y a au plus une valeur élue par T, mais il peut ne pas y en avoir.

Par exemple, 3 est élu par `[|1;3;2;3;3|]`, et aucune valeur n'est élue par `[|1;3;2;3;2;3|]`.

Le fil directeur de cet exercice est de chercher la manière la plus efficace de calculer la valeur élue par un tableau donné, si elle existe.

5.1 Méthode naïve

1. (a) Écrire une fonction `occurrences` prenant en argument un **tableau** *tab* (de type `int array`) et un entier n, et renvoyant le nombre d'occurrences de n dans *tab*.
   (b) Déterminer la complexité de cette fonction.
2. (a) En déduire une fonction `elu1` prenant en argument un entier k et un tableau *tab* d'entiers de $[\![0; k-1]\!]$, et renvoyant la valeur élue par *tab*, ou -1 si elle n'existe pas.
   (b) Déterminer la complexité de cette fonction.

5.2 Par le tableau des occurrences

1. Écrire une fonction `tableau_occ` prenant en argument un entier k et un tableau *tab* d'entiers de $[\![0; k-1]\!]$, et renvoyant un tableau *occ* de longueur k tel que $occ.(n)$ ait pour valeur le nombre d'occurrences de n dans *tab*.

   Cette fonction ne devra faire qu'un seul parcours du tableau *tab*.

   On rappelle que `Array.make k 0` renvoie un tableau de k zéros, avec une complexité en $O(k)$.
2. En déduire une fonction `elu2` prenant en argument un entier k et un tableau *tab* d'entiers de $[\![0; k-1]\!]$, et renvoyant la valeur élue par *tab*, ou -1 si elle n'existe pas.
3. Déterminer la complexité de la fonction précédente.

5.3 *Diviser pour régner* simple

1. Soit T un tableau de longueur N, soit $m = \lfloor N/2 \rfloor$. Démontrer que si n est élue par T, alors n est élue par $T[0:m]$ ou $T[m:N]$ (où $T[a:b]$ est la notation à la Python pour la portion de T de l'indice a inclus à l'indice b exclu).
2. Écrire une fonction `occurrences2` prenant en argument un tableau *tab*, un entier n et deux indices i et j et renvoyant le nombre d'occurrences de n dans $tab[i:j]$.
3. (a) Écrire une fonction `dpr_simple` prenant en argument un tableau *tab* et deux bornes i et j, et renvoyant $(-1, 0)$ si $tab[i:j]$ n'a pas de valeur élue, et (n, q) si n est élue par $tab[i:j]$ et y apparaît exactement q fois. Cette fonction devra appliquer le principe diviser pour régner.
   (b) Démontrer la terminaison de la fonction précédente.
4. En déduire une fonction `elu3` prenant en argument un tableau *tab* d'entiers et renvoyant la valeur élue par *tab*, ou -1 si elle n'existe pas.
5. Déterminer la complexité de la fonction précédente.

5.4 *Diviser pour régner* avancé

Soit T un tableau de longueur N. On dit que $a \in \mathbb{N}$ est un *postulant* de T pour la valeur $n \in \mathbb{N}$ si :
- $n > N/2$;
- a apparaît au plus n fois dans T ;
- tout entier b différent de a apparaît au plus $N - n$ fois dans T.

On dit que a est un postulant de T s'il existe n tel que a soit postulant de T pour n.

1. Montrer que 3 est un postulant de $T =$ `[|1;2;3;4;3;2;3;3|]`.
2. Démontrer que si a est élu par T, alors a est un postulant de T.
3. Démontrer que si a est un postulant de T, alors aucune autre valeur ne peut être élue par T.
4. Donner un exemple de tableau ayant un postulant mais pas d'élu et un exemple de tableau n'ayant aucun postulant.
5. Soit T un tableau de longueur N **paire**. Soit $m = N/2$.

(a) On suppose que $T[m : N]$ n'a pas d'élu, et que a est postulant de $T[0 : m]$ pour une valeur l. Démontrer que a est postulant de T pour une valeur n que l'on exprimera en fonction de m et l.

(b) On suppose que a est un postulant de $T[0 : m]$ pour l, et que b un postulant de $T[m : N]$ pour q.

i. On suppose $a = b$. Démontrer que a est un postulant de T pour une valeur n que l'on exprimera en fonction de l et q.

ii. On suppose que $a \neq b$ et $q > l$. Démontrer que b est un postulant de T pour $n = m + q - l$.

iii. On suppose que $a \neq b$ et $q = l$. Démontrer que T n'admet pas d'élu.

6. Écrire une fonction `postulant` prenant en argument un tableau tab et deux indices i et j, et renvoyant un couple (a, n) tel que :
   - si $(a, n) = (-1, 0)$, alors $tab[i : j]$ n'a pas d'élu ;
   - sinon, a est un postulant de $tab[i : j]$ pour n.

   On supposera pour simplifier que la longueur de $tab[i : j]$ est une puissance de 2. Cette fonction devra appliquer le principe diviser pour régner.
7. En déduire une fonction `elu4` prenant en argument un tableau tab de longueur une puissance de 2, et renvoyant la valeur élue par tab, ou -1 si elle n'existe pas.
8. Déterminer la complexité de la fonction précédente.