Université de Strasbourg
UFR de mathématique et d'informatique

L2/CMI-2ème année
automne 2016

Structures de données et algorithmes

TP noté sujet 1

Durée 2h. Vous ne pouvez consulter que le code C que vous avez programmé lors des TP précédents .

1 Fonctions de [0..N-1] dans [0..N-1]

On va dans cette section implanter les fonctions mathématiques dont les ensembles de départ et d'arrivée sont finis et égaux à $[\![0, N-1]\!]$ où les doubles crochets désignent l'intervalle des entiers entre 0 et $N-1$ inclus et qu'on note `[0..N-1]` dans le code.

En mathématiques, de telles fonctions peuvent être définies par une formule, comme, par exemple,dans $f : x \mapsto (2.x + 3)\%N$ où $\%$ désigne le reste de la division entière, ou en extension en donnant explicitement toutes les valeurs prises par la fonction comme, par exemple, les fonctions booléennes ($N = 2$) ou encore avec $N = 3$, la fonction f définie par

$$\begin{array}{rcl} f : [\![0,2]\!] & \to & [\![0,2]\!] \\ 0 & \mapsto & 1 \\ 1 & \mapsto & 1 \\ 2 & \mapsto & 0 \end{array}$$

Dans ce TP on suit plutôt le deuxième point de vue et on spécifie la notion de fonction mathématique de la manière suivante :

```
Spéc FONC0 étend BASE
   Sorte Fun
   Opérations
       f-id    : Nat -> Fun          ** f_id(N) :  fonction identité de [0..N-1]
       f-taille : Fun -> Nat         ** N = cardinal de l'ensemble de définition
       f-val   : Fun Nat -> Nat      ** valeur f(i) d'une fonction f pour un entier i
       f-modif : Fun Nat Nat -> Fun  ** modification d'une fonction en un entier
       f-comp  : Fun Fun -> Fun      ** composition de fonctions
   Vars
      F G : Fun
      i j k N: Nat
   Préconditions
     préc f-val(F, i) = 0 <= i et i < f-taille(F)        ** à compléter
   Axiomes
   f-taille(f-id(N)) = N
   f-taille(f-modif(F,i,j)) = f-taille(F).
   f-taille(f-comp(F,G)) = f-taille(F)
   f-val(f-id(N), i) = i
   f-val(f-modif(F,i,j),k) = si i==k alors j sinon f-val(F,k) finsi.
   f-val(f-comp(F,G),i) = f-val(F, f-val(G,i))
FinSpéc
```

On implante classiquement cette spécification en utilisant un tableau alloué dynamiquement pour stocker les $f(i)$ avec la structure de données décrite dans le fichier `*.h` suivant (`val[i]` correspond à $f(i)$) :

```
// fichier fonc0.h
// ... à compléter

typedef unsigned int Nat;
typedef struct sf { Nat *val;      // val[i] correspond à l'image de i par la fonction
                    Nat   N;       // espace mémoire réservé
                  } Fun;           // type mutable
Fun f_id(Nat N);                   // fonction identité de [0..N-1] : f(i) = i
Nat f_taille(Fun f);               // retourne la taille de la mémoire réservée
```

```
Nat f_val(Fun f, int i);           // retourne f(i). Prec : .... à compléter
Fun f_modif(Fun f, int i, int j);  // modifiie la valeur de f(i). Prec ...
Fun f_comp(Fun f, Fun g);          //  retourne f o g : composifition de f et g
void f_print(Fun f);                    // affichage
```

1. Complétez en commentaire les préconditions et indiquez pourquoi ce type concret est qualifié de mutable.
2. programmez dans un fichier de nom `fonc0.c` les fonctions `C` indiquées dans `fonc0.h`
3. faites un programme de test simple dans un fichier de nom `test0.c`

2 Permutations

Une permutation de $[\![0, N-1]\!]$ est une fonction de $[\![0, N-1]\!]$ dans $[\![0, N-1]\!]$ qui est bijective c'est-à-dire telle que tout élément de $[\![0, N-1]\!]$ possède un et un seul antécédent. Avec des ensembles finis, il est suffisant de montrer que tout élément de $[\![0, N-1]\!]$ à un antécédent ou, en d'autres termes, pour tout j de $[\![0, N-1]\!]$ il existe i dans $[\![0, N-1]\!]$ tel que $f(i) = j$.

Par exemple, la fonction identité est une permutation de $[\![0, N-1]\!]$. On note habituellement une permutation par la liste des images, par exemple :

- $(0, 1, 2, \ldots N-1)$ correspond à l'identité et
- $(1, 2, \ldots N-1, 0)$ correspond à la permutation $x \mapsto (x+1)\%N$.

La liste des images doit donc contenir tous les entiers de 0 à $N-1$ et la composée de deux permutations de même taille est évidemment une permutation.
On se propose de programmer différentes fonctions sur les permutations en utilisant la structure de données pour les fonctions. On aura donc le fichier `permut.h`

```
// fichier permut.h
// ...
#include "fonc0.h"
typedef Fun Perm;

Bool p_ispermut(Fun F);                    // teste si une fonction est une permutation
Nat *p_orbite(Perm p, Nat i);              // orbite d'un lément suivant p
Perm p_transp(Nat N, Nat i, Nat j);        // transposition de taille N échangeant i et j
Perm p_cycle(Nat N, Nat elems[], Nat r);   // cycle de taille N et de longueur r
                                           // défini par le tableau elems[], Préc ....
Perm *p_decomp(Perm p);      // décompose p en cycles, retourne un tableau de permutations
```

la première fonction `Csert` à tester si une fonction est une permutation, les autres fonctions `C` correspondent à des notions décrites plus loin.

1. Programmez dans le fichier `permut.c` la fonction `p_ispermut()`.

2. On appelle orbite de l'élément i suivant la permutation p l'ensemble des images de i par p, p^2, p^3, *etc.* c'est-à-dire $\{i, p(i), p(p(i)), p^3(i), \ldots p^r(i), \ldots\}$ qui est nécessairement fini.
Programmez la fonction `p_orbite()` qui retourne l'orbite d'un élément sous forme d'un tableau.

Cycles. Un cycle est une permutation circulaire d'éléments donnés de $[\![0, N-1]\!]$, les autres étant inchangés. Plus formellement, un cycle de longueur r est une permutation c définie par la donnée de r entiers $\{x_0, ..., x_{r-1}\} \subset [\![0, N-1]\!]$ et $c(x_k) = x_{k+1}$ si $k \neq r-1$ et $c(x_{r-1}) = x_0$ sinon; et enfin $c(i) = i$ si $i \notin \{x_0, ..., x_{r-1}\}$. Par exemple, voici un cycle de taille 6 et de longueur 3 :$(0, 3, 2, 5, 4, 1)$ où l'orbite de 1 est $\{1, 3, 5\}$. En utilisant les orbites, on voit facilement qu'on peut décomposer toute permutation en cycles (avec éventuellement des cycles de longueur 1).

On appelle transposition tout cycle de longueur 2 : une transposition échange donc deux valeurs i et j.

3. Programmez les dernières fonctions dont les prototypes sont donnés plus haut.

3 Ordre

On ordonne habituellement les permutations de même taille en utilisant l'ordre lexicographique de la liste des images. Précisément, on compare 2 N-uplets en comparant les premiers termes, dès qu'on en trouve un différent les deux N-uplets, on peut les classer. Avec `t1` et `t2` deux n-uplets (sorte `Vect` vue en TP), cela conduit à la spécification :

Opérations

```
    _<_ : Vect Vect -> Bool         ** comparaison de 2 N-uplets
    inf   :  Vect Vect Nat -> Bool  ** comparaison à partir du rang i
```

Axiomes

```
   t1 < t2 = inf(t1,t2,0)
   inft(t1,t2,i) = si i == N-1
                   alors t1[N-1] < t2[N-2]
                   sinon t1[i] < t2[i] ou (t1[i] == t2[i] et inf(t1,t2,i+1))
                   fin si
```

On se sert de cet ordre pour comparer les permutations en comparant les N-uplets des images. Par exemple, la permutation $(3, 2, 0, 1)$ est plus petite que la permutation $(3, 2, 1, 0)$.

1. Implantez cette opération et testez votre implantation.

Pour les plus rapides ...

2. Étant donnée une permutation p, quelle est la permutation immédiatement supérieure ? On appellera `p_next()` cette fonction. Programmez-la.
(On pourra utiliser la fonction de tri dont le code est donné à la suite :

```
void bsort(Nat tab[], Nat i, Nat N)    // tri à bulle
// trie le tableau tab par ordre croissant  entre les indices i et N (N exclu)
// tab est modifié par effet de bord
{
  int tmp;
   for(int j = i; j < N-1; j++)
         for(int k=i; k < N+i-1-j; k++)
               if(tab[k]>tab[k+1])
      {tmp = tab[k]; tab[k]=tab[k+1]; tab[k+1]=tmp; }
}
```

3. Utilisez la fonction précédente pour programmer une fonction `C` qui affiche toutes les permutations d'une taille donnée (attention, pour la taille N il y a $N!$ permutations et, par exemple, $6! = 720$).