T.P. 3

Classes et permutations

L'objectif de ce TP est d'écrire une classe représentant une famille d'objets mathématiques, les permutations, et de l'utiliser pour calculer numériquement certaines propriétés de ces objets.

Après quelques rappels théoriques pour fixer les définitions, on propose quelques questions auxquelles on essaiera de répondre numériquement pour deviner la réponse (en attendant une démonstration mathématique que vous pourrez chercher sur votre temps libre)

3.1 Permutations

3.1.1 Introduction mathématique

Une permutation σ de taille n est une bijection de $\{0, \dots, n-1\}$ dans lui-même. On note $\mathfrak{S}_n$ l'ensemble des permutations de taille n. Cet ensemble est de cardinal $n!$.

L'ensemble $\mathfrak{S}_n$ muni de la composition des applications a une structure de groupe, dont l'identité $id : i \mapsto i$ est l'élément neutre. Le produit $\sigma \cdot \tau$ de deux permutations est donc la permutation qui envoie tout élément i vers $\sigma(\tau(i))$. Attention, ce n'est pas commutatif.

L'*ordre* d'une permutation σ est le plus petit entier strictement positif k tel que $\sigma^k = id$. Les éléments $\{id, \sigma, \cdots, \sigma^{k-1} = \sigma^{-1}\}$ forment un sous-groupe de $\mathfrak{S}_n$ (le groupe engendré par σ) de cardinal k. Son action sur $\{0, \cdots, n-1\}$ définit naturellement une relation d'équivalence : $i \sim_\sigma j \Leftrightarrow \exists l \in \mathbb{Z} : i = \sigma^l j$. Les classes d'équivalence s'appellent les *orbites* de σ. Lorsqu'une orbite est un singleton, de la forme $\{i\}$, on dit que l'orbite est triviale et que i est un point fixe.

Un *cycle* non trivial est une permutation dont exactement une orbite est de longueur supérieure ou égale à deux. Cette orbite est appelée *support du cycle.*[1] La longueur d'un cycle est la longueur de sa plus grande orbite. L'ordre d'un cycle est la longueur du cycle. Une *transposition* est un cycle de longueur 2, qui échange donc deux éléments.

Un théorème dit que toute permutation se décompose en produit de cycles de supports disjoints. Cette décomposition est unique à l'ordre près des facteurs, qui commutent. L'ordre d'une permutation est alors le ppcm des ordres de tous les cycles qui la composent.

Une permutation σ peut être représentée de plusieurs façons : comme un tableau dont la première ligne liste les éléments de 0 à $n-1$, et la deuxième liste les images de

1. Par extension, pour inclure l'identité comme cycle trivial, on peut dire qu'un cycle a au plus une orbite non triviale.

l'élément au dessus :

$$\sigma = \begin{pmatrix} 0 & 1 & \cdots & n-1 \\ \sigma(0) & \sigma(1) & \cdots & \sigma(n-1) \end{pmatrix}$$

On donne parfois uniquement la seconde ligne du tableau, écrite comme un « mot dans les lettres $0, \ldots, n-1$ ».

Un cycle est parfois donné par la suite des images d'un élément de l'orbite non triviale (en oubliant les points fixes). On peut alors représenter une permutation en juxtaposant les cycles qui la composent.

Par exemple, la permutation σ de $\mathfrak{S}_6$ qui envoit respectivement 0,1,2,3,4,5 sur 2,4,5,3,1,0 peut être représenté par

$$\sigma = \begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 \\ 2 & 4 & 5 & 3 & 1 & 0 \end{pmatrix}$$

ou simplement $\sigma =$ "245310". Par itérations successives de σ, on a $0 \mapsto 2 \mapsto 5 \mapsto 0$, et $1 \mapsto 4 \mapsto 1$ alors que 3 est un point fixe. σ est donc le produit d'un cycle de longueur 2 (transposition) (14) et d'un cycle de longueur 3 (025). On écrit alors

$$\sigma = (025)(14).$$

en prenant la convention d'écrire chaque cycle avec le plus petit élément en premier, et de ranger les cycles dans l'ordre décroissant des premiers éléments (cette écriture est alors unique). Cette permutation est donc d'ordre $3 \times 2 = 6$.

3.1.2 Questions d'intérêt

Une fois introduites toutes ces définitions, on voudrait avoir une idée des nombres qui apparaissent pour toutes les grandeurs qu'on a introduites : combien y a-t-il de permutations sans point fixe[2] ? Avec 1, 2,..., k points fixes ? Quelle est la distribution de l'ordre des permutations parmi les éléments de $\mathfrak{S}_n$? du nombre de cycles ?

On essaiera de répondre à ces questions en cherchant à faire fonctionner le code suivant écrit dans un fichier `test_perm.cpp` . Il s'agit donc de se placer du côté du développeur mathématicien qui va développer une bibliothèque C++ `my_permutation` dans deux fichiers `my_permutation.hpp` et `my_permutation.cpp` où tous les formules mathématiques seront implémentées.

Les fichiers `file_s.dat` et `file_t.dat` seront à télécharger depuis le site du cours.

```
// ------ CONTENU DU FICHIER test_perm.cpp -------
#include "my_permutation.hpp"
#include <iostream>
#include <fstream>
#include <algorithm>
#include <iterator>
#include <random> // pour le générateur aléatoire std::mt19937
int main () {
// *******Premiere partie: bases de la bibliothèque
// Déclaration de deux permutations de deux manières différentes:
    Permutation b(6); //identite
```

2. Les permutations sans point fixe sont appelées des *dérangements*.

```
12      std::vector<int> v{2,4,5,3,1,0};//syntaxe C++11 avec { }
        Permutation a(v);
14  // Calcul des itérées d'une permutation
        for(int i=0; i<=6; ++i) {
16        std::cout << "a^" << i << "\n" << b << "\n";
          b = b*a;
18      }
    // Calcul des points fixes d'une permutation
20      std::list<int> fp = a.fixed_points();
        std::copy( fp.begin(), fp.end(),
22                 std::ostream_iterator<int>(std::cout," ") );
        std::cout << "\n";
24
    // ******* Deuxieme partie: un peu plus d'algorithmique
26  // lecture de deux permutations dans deux fichiers
        std::ifstream fichier_s("./file_s.dat");
28      std::ifstream fichier_t("./file_t.dat");
        Permutation s,t;
30      fichier_s >> s;
        fichier_t >> t;
32      fichier_s.close();
        fichier_t.close();
34   // Calcul de cycles et ordres
       std::list<Cycle> la = a.cycles();
36     cout << "Les cycles de a sont: ";
       for (const auto & c: la) cout << c << ";";
38     std::cout << "\nL'ordre de la permutation a est égal à "
                 << a.order() << "\n";//est-ce cohérent ?
40     Permutation u=s*t.inverse();
       std::list<Cycle> lu = u.cycles();
42     cout << "Les cycles de u ont pour longueurs: ";
       for(const Cycle & c:lu) cout << c.order() << " ";
44     long int order_u = u.order();
       std::cout << "\nL'ordre de la permutation u est égal à "
46               << order_u << "\n";//est-ce cohérent ?

48  // ******* Troisieme partie: génération aléatoire et Monte-Carlo
        std::mt19937 g(time(nullptr));
50      unsigned n=100;
        unsigned nb_echant = 10000;
52      unsigned nb_derang = 0;
        for(unsigned i = 0; i < nb_echant; ++i) {
54          nb_derang += Permutation(n,g).is_derangement();
        }
56      std::cout << "La proportion de dérangements est environ "
            << nb_derang/double(nb_echant) << "\n";
58      return 0;
    }
```

Pour cela, nous allons procéder pas à pas et nous vous recommandons dans le développement de n'importe quelle bibliothèque de procéder de même.

3.2 Implémentation de la classe

3.2.1 Préparation et outils élémentaires

Question 3.1. Créer deux fichiers `my_permutation.hpp` (en-tête) et `my_permutation.cpp` (code). Faire les inclusions de bibliothèques nécessaires.

Question 3.2. Définir dans le fichier d'en-tête une classe `Permutation` avec comme champs privés :

- un entier `n` qui est la taille de la permutation
- un vecteur d'entiers `images` qui contiendra plus tard les images de chaque point (i.e. `images[i]` est l'image de `i` par la permutation objet).

Question 3.3. Faire la liste dans le code précédent de *tous les constructeurs, méthodes et opérateurs nécessaires* pour faire marcher le code du fichier de test `test_perm.cpp` . Écrire leurs prototypes dans la partie publique de la classe en faisant bien attention aux `const` et aux passages des arguments par référence.

Il va falloir maintenant écrire dans le fichier `.cpp` les codes de chacun et les tester au fur et à mesure : il est donc *important* de s'y prendre dans le bon ordre. Nous vous conseillons donc la progression suivante.

3.2.2 Initialisation et entrées-sorties

Question 3.4. Écrire le code du constructeur qui permet de déclarer une variable en l'initialisant à l'identité de taille n (ligne 10 du code ci-dessus). Donner une valeur par défaut 1 à l'argument du constructeur.

Question 3.5. Surcharger l'opérateur d'affichage `<<` (ligne 15 du code ci-dessus). On souhaite qu'une permutation de taille n s'affiche sur une seule ligne avec la taille n sur la première ligne puis une espace puis deux points `:` puis une espace et enfin les images successives séparées par des espaces. La permutation `a` précédente s'affiche alors sous la forme :

```
6 : 2 4 5 3 1 0
```

Question 3.6. Tester les deux dans un court programme de quelques lignes pour vérifier que vous êtes capable d'afficher une permutation égale à l'identité.

Question 3.7. Surcharger l'opérateur de lecture `>>` (lignes 28 et 29). Le tester dans un court programme en lisant le fichier `file_s.dat` dans une permutation et en la réécrivant dans un fichier `test_file_s.dat` (il faut alors vérifier que les deux sont identiques !).

Question 3.8. Ajouter un accesseur `size()` au champ `n` (taille de la permutation).

Question 3.9. Ajouter un accesseur à la i-ème case du vecteur `images` via l'opérateur `[]` de sorte que `a[i]` renvoie l'image de `i` par la permutation `a`. Le tester.

Question 3.10. À quoi ressemblerait le mutateur associé ? Pourquoi ne faut-il surtout pas le mettre ?

Question 3.11. Écrire le constructeur utilisé à la ligne 11 qui crée une permutation telle que les images de chaque point sont les nombres lus dans le vecteur donné en argument. On supposera que le contenu du vecteur de départ contient une et une seule fois chaque entier.

Question 3.12. Faut-il définir un constructeur par copie ? Un opérateur d'affectation `=` ? Justifier.

3.2.3 Méthodes et opérateurs supplémentaires

Nous pouvons à présent commencer à coder le reste des méthodes et opérateurs en utilisant le travail précédent pour vérifier rapidement des résultats.

Question 3.13. Écrire le code de la composition de deux permutations comme surcharge de `*` (utilisé par exemple ligne 16). On pourra le faire soit par une méthode, soit par une fonction. Tester les lignes 14 à 17. On prendra les conventions suivantes :

- si les deux permutations sont de même taille, alors c'est la composition usuelle définie ci-dessus.
- si les deux permutations ont des tailles n et m avec $n < m$, alors on considérera la "petite permutation" sur $\{1, \dots, n\}$ comme une permutation σ sur $\{1, \dots, m\}$ telle que $\sigma(i) = i$ pour $i > n$. Le résultat sera alors de taille m.

Question 3.14. Écrire le code de la méthode `fixed_points`. Tester les lignes 19 à 22. Écrire également la méthode `is_derangement` qui renvoie vraie ou faux selon que la permutation est un dérangement ou non.

Question 3.15. Écrire la méthode `inverse` telle que `a.inverse()` ne modifie pas `a` et renvoie l'inverse de `a` pour la composition.

3.2.4 Décomposition en cycles

Pour décrire un cycle qui est un exemple particulier de permutation, nous introduisons la classe suivante

```
class Cycle {
   private:
      std::vector<int> elem;
      void add_last_point(int);//ajoute un point à la fin du cycle
      //std::vector<int>::iterator find(int) const;
   public:
      long int order() const;
      //int operator[](int ) const; //uses find
```

```
    //Cycle inverse() const;
};
```

Le champ `elem` contient les éléments du cycle de sorte que `elem[k+1]` est l'image par la permutation de `elem[k]` et `elem.front()` est l'image de `elem.back()`. Les éléments qui n'apparaissent pas dans `elem` sont considérés comme des points fixes.

Question 3.16. Est-il nécessaire de définir le constructeur par défaut ?

Question 3.17. Compléter la méthode *privée* `add_last_point` qui ajoute un point dans le cycle sans aucune vérification. Pourquoi mettre cette méthode privée ?

Question 3.18. Compléter l'accesseur `order()` à l'ordre du cycle.

Question 3.19. Surcharger `<<` pour les cycles, de telle sorte que l'affichage soit le suivant

```
[ elem[0] ..... elem[k] ]
```

Question 3.20. Écrire la méthode `cycles()` de la classe `Permutation` qui renvoie une liste d'éléments de type `Cycle`. On propose pour cela l'algorithme suivant : créer une liste de cycles vide `L`. Mettre dans un ensemble `S` tous les éléments de 0 à $n-1$. Tant que l'ensemble `S` n'est pas vide, retirer le plus petit `x` (accessible par l'itérateur `begin()` de la classe `std::set`). Si `x` n'est pas un point fixe, commencer un cycle en ajoutant `x` à un cycle vide. Retirer (utiliser `find` et `erase` dans `std::set`) de `S` les images itérées de `x` et les ajouter au cycle commencé jusqu'à retomber sur `x`. Ajouter alors le cycle ainsi formé à la liste `L`. Une fois que `S` est vide, renvoyer `L`.

Remarque : observer que le cycle ainsi construit commence bien par son plus petit élément car `S` est ordonné.

On observe également que pour cela il faut que la classe `Permutation` soit amie de `Cycle` afin de pouvoir utiliser `add_last_point`. Faites le nécessaire.

Question 3.21. Nous allons avoir besoin de calculer le PGCD (GCD en anglais) et le PPCM (LCM en anglais) d'une liste de grands nombres. Rappel : pour deux entiers a et b, le pgcd d de a et b peut se calculer itérativement par l'algorithme de la division euclidienne. On pose $(u_1, u_2) = (a, b)$. Si $u_2 = 0$, on renvoie u_1, sinon, tant que $u_2 \neq 0$, on remplace (u_1, u_2) par (u_2, r) où `r=u_1 % u_2` est le reste dans la division euclidienne de u_1 par u_2. Le PPCM est alors $m = ab/d$ où d est le PGCD de a et b. Écrire deux fonctions

```
long int pgcd(long int a,long int b);
long int ppcm(long int a,long int b);
```

qui calculent ces PGCD et PPCM.

Question 3.22. Les PGCD et PPCM sont deux opérations associatives et commutatives de telle sorte que le PGCD (resp. PPCM) de n nombres $(a_1, \ldots, a_n)$ peut se faire de la manière suivante : on pose $P_0 = 0$ puis $P_{k+1} = pgcd(P_k, a_k)$ (resp. $ppcm(P_k, a_k)$, pour $1 \leq k \leq n$ et le PGCD (resp. PPCM) est donné par le dernier terme P_n. Écrire le code de la méthode

```
long int Permutation::order() const;
```

en combinant la méthode `Cycle`, la fonction `ppcm` et `std::algorithm`. Le code ne doit pas prendre plus de deux ou trois instructions (sinon, faites à votre manière).

Question 3.23. Tester les lignes 34 à 46 sur le calcul de les ordres de `a` et `u`. Cela vous semble-t-il normal ? Où est le problème ? Avez-vous des idées pour le résoudre ?

Question 3.24. Ajouter un opérateur de comparaison `<` entre deux cycles qui fonctionne de la manière suivante : si les deux cycles ont des longueurs différentes alors on compare leurs longueurs ; en cas d'égalité de longueurs, on regarde l'ordre de leurs premiers éléments puis, en cas d'égalité on regarde leurs deuxièmes éléments, etc (on pourra si on le souhaite se référer à `std::mismatch` dans `<algorithm>`, voire encore mieux à `std::lexicographical_compare`). Cette méthode est nécessaire pour `std::max_element` en ligne 43 du code. Tester les lignes 39 à 44 sur `u`.

Questions supplémentaires sur les cycles non-nécessaires pour le code précédent mais nécessaires pour la section 3.3 ci-dessous.

Question 3.25. Ajouter une méthode `find(i)` qui cherche un élément `i` dans `elem`, renvoie l'itérateur sur son emplacement si `i` est présent et renvoie `elem.end()` sinon.

Question 3.26. Ajouter un accesseur `[i]` à l'image de `i` par un cycle. On pourra utiliser `find` ci-dessus pour distinguer les points fixes pour lesquels il faut renvoyer `i`.

Question 3.27. Ajouter une méthode `inverse()` à la classe `Cycle`.

Question 3.28. Ajouter une méthode `cycles()` à `Cycle` sur le modèle de la même méthode de la classe permutation (attention, c'est trivial).

3.2.5 Génération aléatoire

Nous allons maintenant coder le constructeur utilisé ligne 54. Il prend comme argument un entier qui est la taille de la permutation et un générateur aléatoire de la bibliothèque standard et construit une permutation tirée uniformément sur le groupe symétrique $\mathfrak{S}_n$. On suit l'algorithme suivant de Fisher-Yates-Knuth [3] : on commence par remplir le vecteur `images` par les éléments de 0 à $n-1$. Puis pour tout i entre 0 et $n-2$, on génère un entier j uniforme entre i et $n-1$ inclus (avec `std::uniform_int_distribution<int>`) et échanger les valeurs `images[i]` et `images[j]` (si i=j, on a un point fixe). On pourra

3. Remarque : cet algorithme de mélange de n éléments est également implémenté par `std::shuffle` dans `<algorithm>`.

utiliser `std::swap` pour l'échange sans tampon intermédiaire explicite (c'est géré en interne par `std::swap`). Estimer la complexité d'une construction d'une permutation aléatoire de taille n par cet algorithme.

Question 3.29. Écrire le code du constructeur pour un générateur de type `std::mt19937`. Pourquoi faut-il passer ce dernier par référence ?

Question 3.30. Tester la dernière partie du programme.

Question 3.31. Remplacer le constructeur précédent par un template de constructeur qui marche pour tout générateur de la bibliothèque standard. Tester à nouveau la dernière partie du programme.

3.3 Héritage virtuel

On se rend compte que la classe `Cycle` est un cas particulier de permutation et on souhaite s'assurer que les prototypes des méthodes sont bien les mêmes entre les deux classes. De plus, on souhaite pouvoir écrire des fonctions qui utilisent indifféremment un objet de type `Cycle` ou `Permutation` (tant que, bien sûr, les différences entre les deux objets ne sont pas pertinentes). Pour cela, tant que nous n'avons pas vu les templates, une solution historique consiste à utiliser l'héritage pur.

Pour cela, nous définissons la classe virtuelle pure suivante :

```
class Non_modifiable_Permutation {
        public:
                virtual int size() const = 0;
                virtual int operator[](int) const = 0;
                virtual long int order() const = 0;
                //...
};
```

Question 3.32. Faire hériter `Permutation` et `Cycle` de cette classe.

Question 3.33. Compléter toutes les méthodes possibles dans la classe-mère, communes aux deux classes-filles.

Question 3.34. Que manque-t-il pour `Cycle` ? (compilez et vous saurez) Pour associer une taille au cycle, on pourra par exemple regarder le plus grand élément du cycle et le prendre comme taille.

Question 3.35. Vérifier que tout fonctionne bien.

3.4 Une implémentation alternative pour les permutations avec beaucoup de points fixes

L'implémentation précédente de `Permutation` est nécessaire lorsque la permutation a peu de points fixes : il faut stocker les images de presque tous les points et ajouter quelques points fixes ne change rien (voire permet de simplifier certains algorithmiques). Supposons à présent que nous soyons dans une situation très différente avec n très grand et la plupart des points sont des points fixes. Il devient alors avantageux de stocker seulement l'image des points qui ne sont pas fixes.

Pour cela, on introduit la classe suivante

```
class SparsePermutation {
        private:
                int n;
                std::map<int,int> non_trivial_images;
        public:
                int operator[] (int i) const;
                //...
};
```

(qu'on pourra plus tard si on le souhaite faire hériter de `Non_modifiable_Permutation`). Cela nous donne l'occasion d'introduire le conteneur `std::map<T1,T2>` de la bibliothèque `std::map` (cf. cours). Il permet de décrire une application $f : E_1 \to E_2$ où E_1 et E_2 sont des ensembles *finis* d'objets de type `T1` et `T2` respectivement où `T1` est un ensemble ordonné et `T2` est muni d'une valeur par défaut. D'un point de vue informatique, cette fonction est encodée sous la forme d'une p_1-uplet de valeurs $((x_1, f(x_1)), \ldots, (x_{p_1}, f(x_{p_1})$ où $x_1 < x_2 < \ldots < x_{p_1}$ est une énumération croissante des éléments de E_1.

Si l'on sait que x fait partie de l'ensemble E_1, on peut accéder à l'image $f(x)$ via la syntaxe `f[x]` . Si $x \notin E_1$ alors, l'appel `f[x]` modifie l'objet f en *ajoutant à* E_1 l'élément x et l'élément (x, d) au p_1-uplet de valeur où d est la valeur par défaut du type `T2` . Si on cherche seulement à savoir si $x \in E_1$ et, dans ce cas seulement, à utiliser $f(x)$ alors il faudra faire

```
T1 x= ??? ;
auto where_x = f.find(x);
T2 value_x;
if ( where_x != f.end() ) {
        value_x = where_x->second;
}// si where_x == f.end() alors x n'est pas dans la map.
```

Ici `where_x` est un itérateur vers la case où est écrit la valeur `x` et la valeur `*where_x` est une paire $(x, f(x))$ de type `std::pair<T1,T2>` , de telle sorte que `where_x->first` et `where_x->second` donnent x et $f(x)$.

Nous prendrons alors l'exemple suivant :

```
int SparsePermutation::operator[](int i) const {
   auto where_i = non_trivial_images.find(i);
```

```
    if ( where_i != non_trivial_images.end() )
4       return where_i->second;
    else
6       return i;
}
```

Question 3.36. Ajouter toutes les méthodes nécessaires pour que `SparsePermutation` fonctionne comme `Permutation` ainsi que les opérateurs d'écriture, de lecture et le produit.

Question 3.37. On pourra tester les temps de calcul avec le programme `test_sparse.cpp` sur les fichiers de données `fichier_u.dat` et `fichier_sparse_u.dat` où la même grande permutation est écrite dans deux formats différents.