Sorbonne Université
Master 1 de mathématiques et applications
Unité d'enseignement MU4MA056

Programmation en C++ :

sujets des travaux pratiques

(version du 19 janvier 2024)

Année universitaire 2023–2024

Table des matières

1 Premiers programmes, entrées/sorties, statistiques **2**

2 Un exemple de bibliothèque d'algèbre linéaire : Eigen **7**

3 Classes et permutations **14**

Information importante : Vous trouverez sur la page web du cours un fichier `fichiersTP.zip` qui contient tous les fichiers de données et les codes à trous nécessaires pour les TP de cette brochure :

Paquets nécessaires sur un système Linux Debian/Ubuntu/Mint :

- compilateur `g++` et la bibliothèque `libstdc++5` (ou la version 6), cette dernière étant en général automatiquement installée
- la bibliothèque `libeigen3-dev` pour l'algèbre linéaire
- l'éditeur `geany`
- visualisation des données : Python avec `matplotlib` ou bien le logiciel `gnuplot`
- (optionnel) l'utilitaire de compilation `make`
- (optionnel) le compilateur `clang` (si vous souhaitez tester un autre compilateur)

T.P. 1

Premiers programmes, entrées/sorties, statistiques

1.1 Création et compilation d'un premier programme

1.1.1 Un premier programme

Considérons un programme très simple :

```
#include <iostream>
#include <cmath>

double aire_du_cercle(double x) {return M_PI*x*x;}

int main () {
        double r;
        std::cout << "Entrez le rayon du cercle:" << std::endl;
        std::cin >> r;
        std::cout << "Son aire est " << aire_du_cercle(r) << std::endl;
        return 0;
}
```

1. Créez un répertoire `TP1` sur votre bureau de travail.
2. Ouvrez un terminal et tapez `cd Desktop/TP1` ou `cd Bureau/TP1` (selon la langue du système) pour vous déplacer dans le répertoire créé. Laissez le terminal ouvert.
3. Ouvrez le logiciel Geany, créez un nouveau fichier, copiez le code ci-dessus et sauvegardez le fichier sous le nom `premierprog.cpp` dans le répertoire `TP1`. *Nous vous conseillons de le recopier à la main en essayant de comprendre chaque ligne.*
4. Revenez dans le terminal puis tapez

   ```
   g++ premierprog.cpp -o premierexec
   ```

   (selon la machine, il faut éventuellement ajouter `-lm` à la fin). Si un message apparaît, c'est qu'une erreur a été commise en recopiant le programme : corrigez-la.

5. Tapez à présent `./premierexec` dans le terminal. Il ne vous reste plus qu'à interagir avec votre programme !
6. Ajouter une fonction au programme précédent pour qu'il puisse calculer également l'aire d'un carré.
7. Modifier la fonction `main()` pour demander la longueur du côté d'un carré et afficher son aire.

1.1.2 Un deuxième programme

Considérons le code suivant :

```
#include __________
#include _________
#include ________

___ _____() {
        ___ n;
        _________ << "Entrez un nombre entier <100:" << std::endl;
        std::cin __ n;
        std::vector<int> tableau(n);
        for(__________________) {
                tableau[i]=i*i;
        }
        _____ofstream fichier("donnees.dat");
        fichier << "Voici les carrés des entiers:" << std::endl;
        for(__________________) {
                fichier << i <<": " <<tableau[i] << std::endl;
        }
        fichier._______;
        return 0;
}
```

1. Écrire ce programme dans un fichier `programme2.cpp` dans le répertoire `TP1`. Remplacer tous les `_____` par ce qu'il faut. Le compiler et l'exécuter. Que voyez-vous apparaître dans le répertoire `TP1` ?
2. Modifier ce programme pour avoir, dans le fichier `donnees.dat` sur les mêmes lignes, également les cubes des entiers.
3. Modifier le programme pour que les données soient écrites dans l'ordre décroissant dans le fichier.

1.2 Un brève présentation des entrées/sorties vers les fichiers

— L'écriture dans le terminal se fait avec `std::cout` et l'opérateur `<<`. Cet objet est défini dans la bibliothèque `iostream`.
— La lecture dans le terminal se fait avec `std::cin` et l'opérateur `>>`. Cet objet est défini dans la bibliothèque `iostream`.

— Un saut de ligne se fait avec l'opérateur `std::endl` .
— La déclaration d'un fichier en écriture se fait par

```
std::ofstream F ("Nom du fichier");
```

Cette commande est définie dans la bibliothèque `<fstream>` . On alimente le fichier en contenu avec l'opérateur d'injection `<<` selon la commande `F << CONTENU` où `CONTENU` est une valeur ou une variable.
— La déclaration d'un fichier en lecture se fait par

```
std::ifstream F ("Nom du fichier");
```

Cette commande est définie dans la bibliothèque `<fstream>` . On lit les données entre deux espaces avec `>>` selon `F >> x` où `x` est la variable de stockage des données lues.
— Avant la fin du programme, tout fichier doit être fermé avec `F.close();` .
— Il est possible de se débarrasser de tous les préfixes `std::` en écrivant :

```
using namespace std;
```

juste après les `include` .
— Toute la documentation des classes de la STL (vecteurs, listes, etc) est disponible sur http://www.cplusplus.com/reference/stl/ et sur http://www.cplusplus.com/reference/std/ pour celle sur les algorithmes, l'aléatoire, les complexes, etc.

1.3 Quelques calculs statistiques simples

1.3.1 Sans la bibliothèque `<algorithm>`

L'archive de données `fichiersTP.zip` contient un fichier `smalldata.txt` . Récupérez-le et placez-le dans votre répertoire de travail pour ce TP.

Ce fichier contient 2500 personnes, décrites par leur prénom, leur ville, leur âge et leur temps de course à l'épreuve du 100 mètres. Pour décrire une personne, nous introduisons la structure suivante :

```
struct Fiche {
        std::string prenom;
        std::string ville;
        int age;
        double temps;
};
```

Le type `std::string` permet de stocker des chaînes de caractères et est défini dans la bibliothèque `<string>` .

1. Créez dans `TP1` un programme `analyse.cpp` qui contient les bibliothèques nécessaires, la définition de la structure ci-dessus, une fonction `main()` qui ouvre le fichier `smalldata.txt` en lecture.
2. Déclarez dans ce programme un tableau `vdata` de taille 2500 et contenant des objets de type `Fiche`. Remplir ce tableau avec les données du fichier.
3. En utilisant uniquement des boucles `for`, des tests logiques `if` et en déclarant des variables, écrivez un programme (ou des programmes si vous préférez faire le travail en plusieurs fois) qui répond aux questions suivantes :
   (a) Combien de personnes habitent Lyon ? Quelle est le pourcentage de Lyonnais ?
   (b) Combien de personnes habitent Lyon et ont strictement moins de 30 ans ?
   (c) Existe-t-il un Toulousain dont le prénom commence par la lettre A ?
   (d) Quel est l'âge minimal ? L'âge maximal ? Comment s'appelle le plus jeune ? Le plus âgé ?
   (e) Quel est l'âge moyen des personnes du fichier ? Quel est l'écart-type de leur âge ?
   (f) Les Parisiens sont-ils en moyenne plus rapides au 100 mètres que les Marseillais ?
   (g) Produire un fichier `toulousains.txt` qui contient toutes les informations sur les personnes qui habitent Toulouse. On remplacera dans ce fichier leur âge par leur date de naissance (on supposera que les âges ont été déclarés en 2018).
   (h) Quelle est la covariance empirique entre âge et temps à l'épreuve du 100 mètres sur cet échantillon de Toulousains ?
   (i) Afficher dans le terminal la liste des villes représentées. *Attention, ce n'est pas si facile ! Vous pouvez utiliser si vous le souhaitez le conteneur* `std::set` *pour avoir une solution rapide ou sinon tout refaire à la main.*
4. (bonus) Supposons à présent que nous n'ayons pas donné initialement le nombre de personnes du fichier : cela empêcherait la déclaration du tableau statique `individu` à la bonne taille. Réécrire le début du programme en utilisant à présent un tableau `individu` de type `std::vector<Fiche>` de la classe `<vector>`. *Indication : le remplir avec* `push_back()` *en parcourant le fichier.*

1.3.2 Avec la bibliothèque `<algorithm>`

1. Refaire intégralement toutes les questions précédentes 3.(a) jusqu'à 3.(i) **sans écrire une seule boucle** `for` et en utilisant intensivement la bibliothèque standard `<algorithm>` dont une documentation est disponible sur le site suivant :

   http://www.cplusplus.com/reference/algorithm/

   Vous vous inspirerez des exemples décrits sur cette page pour chaque fonction. Pour certaines questions, vous pourrez également utiliser la fonction `std::accumulate` de la bibliothèque `<numeric>`.

   La plupart des fonctions de `<algorithm>` prennent en argument une fonction de test ou de comparaison : vous pourrez, au choix, soit déclarer ces fonctions dans le préambule de votre programme, soit dans le corps de la fonction `main()` en utilisant des lambda-fonctions du standard C++11.

   Exemple : pour la question (3)-a, il suffit d'écrire :

```
auto is_from_Lyon=[](Fiche f) {
                return (f.ville=="Lyon");}
int nb_Lyon=std::count_if(vdata.begin(),vdata.end(),is_from_Lyon);
std::cout << "Il y a " << nb_Lyon << " Lyonnais.\n";
```

2. Dans `<algorithm>` , il existe une fonction de tri `std::sort` qui fonctionne de la manière suivante. Si `v` est un vecteur d'objets de type `T` et `compare` une fonction de prototype :

```
bool compare(T x, T y)
```

qui renvoie `true` si `y` est plus grand que `x` et `false` sinon, alors l'instruction

```
std::sort(v.begin(),v.end(),compare)
        // pour v de type std::vector ou std::list
```

trie le tableau par ordre croissant. Produire un fichier `data_tri.txt` qui contient les 100 personnes les plus rapides au 100 mètres triées par vitesse décroissante.

1.3.3 Quelques questions additionnelles plus difficiles pour plus de réflexion

Vous pourrez traiter ces questions à la fois en écrivant vous même les boucles nécessaires et en définissant les variables nécessaires au calcul et à la fois en vous appuyant sur les outils de la bibliothèque standard.

1. Quel est le plus petit écart entre les temps de courses au 100 mètres de deux personnes (indice en note de bas de page [1]) ?
2. Créer deux vecteurs `jeunes` et `moinsjeunes` contenant respectivement les fiches des personnes de moins de 40 ans et de strictement plus de 40 ans (indice en bas de page [2]).
3. Écrire dans un fichier `ordre.dat` la liste des 2500 personnes classées selon l'ordre suivant :
   — par ordre alphabétique des prénoms
   — en cas d'égalité, par ordre alphabétique des villes
   — en cas d'égalité à nouveau, de la plus âgée à la plus jeune
   — en cas d'égalité à nouveau, de la plus lente à la plus rapide.
   et, bien sûr, vérifier que le fichier produit est correct.
4. Nous souhaitons établir l'histogramme des âges qui permette de connaître la répartition des âges. En utilisant le conteneur `std::map<int,int>` , calculer l'histogramme et l'afficher ligne par ligne dans le terminal. Quelle est la classe d'âge la plus nombreuse ?

1. Vous pouvez utiliser `std::sort` , `std::adjacent_difference` et rechercher un extremum. Nous vous conseillons tout d'abord d'extraire les temps de courses dans un `std::vector<double>` avant d'appliquer `std::adjacent_difference` à des `double` et non des `Fiche` .

2. Vous pourrez utiliser `std::partition_copy` et les itérateurs `std::back_inserter` de `<iterator>` si vous souhaitez utiliser `<algorithm>` pleinement et ne pas avoir à calculer au préalable le nombre de personnes de chaque catégorie.

T.P. 2

Un exemple de bibliothèque d'algèbre linéaire : Eigen

L'objectif principal de ce TP est de se familiariser avec l'utilisation de bibliothèques externes, en l'occurrence ici la bibliothèque `Eigen` qui est une boîte à outils très complète pour l'algèbre linéaire, et d'en voir deux applications possibles en mathématiques. Toute la documentation nécessaire figure sur le site http://eigen.tuxfamily.org/dox/. **Aller lire la documentation pour comprendre les prototypes et les modes d'emploi des différentes fonctions fait pleinement partie de l'exercice !**

Nous rappelons les bases suivantes :

— Il faut installer la bibliothèque `Eigen` sur son ordinateur (facile sous Ubuntu ou Linux Mint : il suffit d'installer le paquet `libeigen3-dev` par la commande `apt install libeigen3-dev` dans un terminal).

— Il faut compiler avec `g++` et l'option `-I /usr/include/eigen3` sur un système Linux. Pour tirer pleinement parti de la bibliothèque, l'option d'optimisation `-O2` est également conseillée (essayez avec et sans !).

— Il faut déclarer la bibliothèque `<Eigen/Dense>` ou `<Eigen/Sparse>` dans les en-têtes de fichiers selon le type de matrices souhaité.

— Une matrice à coefficients réels et de taille $N \times M$ fixée à l'écriture du programme se déclare par

```
Eigen::Matrix<double, N, M> A;
```

— Si la taille n'est pas fixée à l'écriture du programme mais à son exécution,il faut utiliser

```
Eigen::Matrix<double, Eigen::Dynamic, Eigen::Dynamic> A(N,M);
```

Pour alléger le code, on pourra enregistrer ce type sous la forme

```
typedef Eigen::Matrix<double, Eigen::Dynamic, Eigen::Dynamic>
        MatrixDouble;
```

— On accède au coefficient (i, j) par `A(i,j)` et la numérotation commence à 0 (et non à 1).

— On écrit une matrice avec `o << A` où `o` est un flux d'écriture de type `std::ostream`.
— on additionne (resp. multiplie) des matrices avec l'opérateur `+` (resp. `*`) et on les affecte avec `=`.

2.1 Premiers pas : la fonction puissance

Attention : les sections suivantes vous demandent de coder plusieurs fonctions de puissance : n'utilisez pas le même nom à chaque fois ! ! !

2.1.1 Un prototype de fonction récursive

On se propose d'écrire une fonction récursive [1] permettant de calculer la puissance d'une matrice. Elle se déclare comme suit :

```
MatrixDouble puissance_lente(const MatrixDouble & M, int n){
        if (n==0){  ...  }
        else if (n==1){  ...  }
        else {
                MatrixDouble N(...,...);
                N=puissance_lente(M,n-1);
                return ...
        }
}
```

et elle repose sur le raisonnement suivant :

si $n = 0$ alors $M^n = Id$, sinon $M^n = M(M^{n-1})$.

Question 2.1. Créer un fichier "matrice.cpp" et compléter le code ci-dessus. La matrice identité peut s'obtenir directement par

```
MatrixDouble::Identity(N,N) // pour une matrice carrée de taille N par N
```

qui renvoie la matrice identité. *Bonus : réécrire avec un* `switch` *au lieu d'une suite de test* `if`.

Question 2.2. Dans votre code, combien de multiplications sont effectuées ? Qu'en pensez-vous ? Par ailleurs, pourquoi ajoute-t-on le test `n==1` alors que, mathématiquement, il n'est pas nécessaire ?

Question 2.3. Dans le code `main()` de ce même fichier, déclarer la matrice

$$A = \begin{pmatrix} 0.4 & 0.6 & 0 \\ 0.75 & 0.25 & 0 \\ 0 & 0 & 1 \end{pmatrix},$$

1. Une fonction récursive est une fonction qui fait appel à elle-même en modifiant ses arguments jusqu'à atteindre une condition de sortie. Typiquement, une telle fonction f peut être décrite par $f(x, 0) = f_0(x)$, et $f(x, n) = g(x, f(n-1)(x))$ pour tout $n \geq 1$, avec g une fonction donnée.

puis calculer A^{100} et afficher le résultat dans le terminal. Pour déclarer une matrice ligne par ligne et sans devoir écrire `A(i,j) =...` , il est possible d'utiliser la *comma-initialization* décrite sur la page suivante de la documentation : https://eigen.tuxfamily.org/dox/group__TutorialMatrixClass.html.

Il est **fortement déconseillé** de passer aux questions suivantes avant d'avoir le bon résultat pour le calcul de A^{100}, à savoir

$$A^{100} = \begin{pmatrix} 0.555556 & 0.444444 & 0 \\ 0.555556 & 0.444444 & 0 \\ 0 & 0 & 1 \end{pmatrix}.$$

Question 2.4. Pourquoi y a-t-il une esperluette `&` dans le prototype de la fonction `puissance` ? Combien de copies sont réalisées lors du calcul ? Combien de copies seraient réalisées si la référence `&` était absente ? Faire le test sans `&` pour des puissances 100, 1000 et 10000 [2] et comparer les temps de calcul approximatifs (pour l'instant approximativement, voir plus bas pour des mesures précises).

2.1.2 Perspectives d'optimisation

La matrice A définie dans la section précédente est de petite taille, aussi est-il rapide de calculer ses puissances. Mais pour calculer la puissance n-ème d'une matrice de taille $N \times N$, il faudra n appels à la multiplication de matrices $N \times N$ qui quant à elle correspond à N^2 opérations, ce qui fait nN^2 calculs (on dit alors que l'algorithme qui sous-tend la fonction est de *complexité* $O(nN^2)$). On peut améliorer l'algorithme de la question 2 de la façon suivante :

> Si $n = 0$ alors $M^n = Id$. Sinon, si n est pair, alors $M^n = M^{n/2}M^{n/2}$, et si n est impair, alors $M^n = M(M^{(n-1)/2})(M^{(n-1)/2})$.

Question 2.5. Écrire une fonction `puissance_rapide` qui prend les mêmes types d'arguments d'entrée et de sortie que `puissance_rapide` mais fonctionne sur la récurrence ci-dessus.

Question 2.6. Combien de multiplications utilise votre code (en fonction de n) ? Qu'en pensez-vous ?

Question 2.7. Test : comparer les temps de calcul pour la puissance 1000-ème [3] de la matrice B de taille 30×30 donnée dans le fichier `matrice.dat` [4] (disponible sur le site du cours) selon que l'on utilise `puissance` ou `puissance2` . Tester également l'effet de l'utilisation ou non de l'option de compilation `-O2` de `g++` ainsi que de l'option `-DNDEBUG` (associée à Eigen).

2. Le choix de l'exposant est arbitraire et dépend surtout de votre machine : sur une machine plutôt ancienne et peu puissante, des différences apparaissent dès les petits exposants ; sur une machine récente, les différences de temps ne deviennent sensibles que pour des exposants grands. Nous vous laissons augmenter les exposants jusqu'à observer des différences notables.

3. Là encore, l'exposant est arbitraire et, selon la puissance de votre machine, nous vous laissons l'augmenter jusqu'à voir des différences significatives.

4. On rappelle que l'on peut lire les éléments successifs d'un fichier, séparés par une tabulation, en utilisant l'opérateur `>>` .

Pour cela, on pourra utiliser la bibliothèque `<chrono>` de `C++11` qui est plus précise que `time()`. Pour déterminer le temps effectué par un calcul donné, il suffit de procéder comme suit :

```
auto t1 = std::chrono::system_clock::now();
... // Calcul dont on veut mesurer la durée
auto t2 = std::chrono::system_clock::now();
std::chrono::duration<double> diff = t2-t1;
std::cout << "Il s'est ecoule " << diff.count() << "s." << std::endl;
```

Par ailleurs, la matrice B n'a pas été choisie complètement au hasard : il s'agit d'une *matrice creuse*, ou *sparse matrix* en anglais, c'est-à-dire une matrice qui possède un nombre limité de coefficients non nuls. La bibliothèque `Eigen` possède une façon d'encoder ce type de matrice qui permet de réduire drastiquement les temps de calcul. Pour cela, il faut déclarer `<Eigen/Sparse>` en tête de fichier, et utiliser le type `Eigen::SparseMatrix<double>` au lieu du type `Eigen::Matrix<double, N,M>`. La déclaration d'une matrice creuse se fait de manière similaire à celle d'une matrice dont la taille n'est pas connue à l'avance :

```
Eigen::SparseMatrix<double> Mat(N,M);
```

où N et M sont de type `int`. Les opérations $+$, $-$ et $\times$ s'utilisent de la même façon pour les matrices creuses que pour les matrices classiques, et il est également possible d'effectuer des opérations entre des matrices creuses et classiques :

```
Eigen::Matrix<double, Eigen::Dynamic, Eigen::Dynamic> A(N,N);
Eigen::SparseMatrix<double> B(N,N);
Eigen::Matrix<double,Eigen::Dynamic, Eigen::Dynamic> C1(N,N);
Eigen::Matrix<double,Eigen::Dynamic, Eigen::Dynamic> C2(N,N);
C1 = A*B;
C2 = A+B;
```

En revanche, contrairement au cas des matrices denses, l'accès et la modification du coefficient (i, j) d'une matrice creuse se fait avec `A.coeffRef(i,j)`.

On se référera à la page pour une documentation rapide sur les matrices creuses.

Une matrice creuse `M` prédéfinie peut être mise égale à la matrice identité avec la commande `M.setIdentity();`.

Question 2.8. Écrire une fonction `puissance_rapide_sparse` qui calcule la puissance n-ème d'une matrice creuse. Calculer[5] B^{1000} en écrivant B comme une matrice creuse et en lui appliquant cette fonction, puis comparer le temps de calcul à ceux des questions précédentes.

Question 2.9. (À traiter plus tard dans le semestre) Fusionner les fonctions `puissance_rapide` et `puissance_rapide_sparse` en un unique template de fonction `puissance_rapide<MatrixType>` compatible avec leurs deux types respectifs.

5. Là encore, choisir l'exposant suffisamment grand selon votre machine.

2.2 Matrices aléatoires et leur spectre

En théorie des matrices aléatoires, l'*ensemble gaussien orthogonal* (ou GOE, pour *gaussian orthogonal ensemble*) est l'ensemble des matrices symétriques $A \in \mathcal{M}_N(\mathbb{R})$ dont les coefficients de la diagonale et au-dessus de la diagonale sont indépendants et tels que $a_{ii} \sim \mathcal{N}(0,1)$ et $a_{ij} \sim \mathcal{N}(0,2)$ pour tout $1 \leq i < j \leq N$. En tant que matrices symétriques réelles, elles sont diagonalisables avec des valeurs propres réelles. On peut montrer que presque-sûrement (par rapport à la mesure de Lebesgue sur les matrices) ces valeurs propres $(\lambda_1, ..., \lambda_N)$ sont toutes distinctes. On peut alors définir la mesure empirique spectrale des valeurs propres [6]

$$\mu_N = \frac{1}{N} \sum_{i=1}^{N} \delta_{\frac{\lambda_i}{2\sqrt{N}}}.$$

Un résultat classique est que cette mesure converge étroitement, lorsque N tend vers l'infini, vers la mesure du demi-cercle

$$\mathrm{d}\sigma(x) = \frac{1}{\pi}\sqrt{4-x^2}\mathbf{1}_{[-2,2]}(x)\mathrm{d}x \tag{2.1}$$

dont la densité est un demi-cercle centré en 0 et de rayon 2. En particulier, la mesure limite est à support compact.

Diagonaliser avec `Eigen`. La bibliothèque `<Eigen/Eigenvalues>` permet de calculer les valeurs propres d'une matrice : étant donné une matrice `MatrixDouble M(N,N)`, on déclare l'algorithme de calcul de ses valeurs propres par

```
Eigen::EigenSolver<MatrixDouble> Solver(M);
```

et `auto spectrum=Solver.eigenvalues()` renvoie un vecteur `spectrum` de taille N à coefficients complexes contenant les différentes valeurs propres. Un nombre complexe $x+iy$ est modélisé comme un couple (x,y) et n'est donc pas de type `double`, mais comme on sait que dans le cas du GOE celles-ci sont réelles, elles sont égales à leur partie réelle. Aussi, on obtient la i-ème valeur propre réelle par `spectrum[i].real()`.

Générer des nombres aléatoires en `C++11`. On (rappelle qu'on) peut simuler une loi uniforme sur $[a,b[$ de la façon suivante en C++11 :

1. On inclut les bibliothèques `<random>` [7] et `<ctime>` ;
2. On déclare un générateur aléatoire :

```
std::mt19937_64 G(time(NULL));
```

3. On déclare la loi uniforme sur $[a,b[$:

6. modulo une renormalisation en $1/2\sqrt{N}$ pour des raisons de convergence.
7. La bibliothèque `<random>` nécessite de compiler avec l'option c++ 11 !

```
uniform_real_distribution<double> Loi(a,b);
```

ou encore la loi normale $\mathcal{N}(m,s)$ par :

```
std::normal_distribution<double> Loi(m,s);
```

4. On simule une variable aléatoire X qui suit cette loi cette loi via

```
double X;
X = Loi(G);
```

Tous les appels successifs de `Loi(G)` produisent des variables aléatoires indépendantes.

Afin de tester la convergence de μ_N vers la loi du demi-cercle décrite en (2.1), nous souhaitons réaliser un histogramme des valeurs propres. Cela signifie diviser le segment $[-3,3]$ (par exemple) qui contient le support de σ en K segments consécutifs de même taille et compter le nombre de valeurs propres qui tombent dans chaque segment.

Question 2.10. Nous souhaitons à présent réaliser un histogramme à $K = 20$ boîtes sur le segment $[-3,3]$ des valeurs propres normalisées $\lambda/(2\sqrt{N})$. Pour cela, on crée un vecteur `std::vector<double> hist(K,0)` dont chaque case `hist[k]` va contenir le nombre de valeurs propres normalisées qui tombent dans le segment $[-3+k(6/K), -3+(k+1)6/K[$.

On simule ensuite un nombre[8] $n = 20$ de matrices indépendantes $(GOE_k)_{1\leq k\leq n}$ de taille $N = 150$. Pour chacune de ces matrices, calculer ses valeurs propres (λ_i) et incrémenter `hist[k]` de $\frac{1}{nN}$ si la valeur propre normalisée $\lambda/(2\sqrt{N})$ tombe dans le segment $[-3+k(6/M), -3+(k+1)6/M[$. Si la valeur propre normalisée ne tombe pas dans $[-3,3[$, alors aucune case de `hist` n'est incrémentée.

Dans un fichier `"eigenvalues.dat"`, stocker dans deux colonnes séparées par une tabulation `"\t "` les centres de chaque segment $[-3+k(6/M), -3+(k+1)6/M[$ et la valeur de `hist` correspondant à ce segment.

Remarque : si vous n'y parvenez pas, passez cette question et celle qui suit.

Question 2.11. En utilisant gnuplot, afficher le résultat de la question précédente avec la commande `plot "eigenvalues.dat" with boxes`. On pourra au besoin adapter l'échelle des ordonnées à l'aide de la commande `set yrange[a:b]` avec `a` et `b` respectivement les valeurs minimale et maximale des ordonnées que l'on veut afficher.

Question 2.12. Créer une fonction

```
auto generate_random_spectrum(std::mt19937 & G, N)
```

qui génère le spectre (cf. ci-dessus) d'une matrice du GOE de taille `N` en utilisant le générateur `G`. Tester.

8. Si le temps le permet, ne pas hésiter à simuler un nombre plus grand, par exemple 50 !

Question 2.13. Pour l'histogramme, le plus simple est de créer une classe assez simpliste qui va permettre d'organiser les choses. Nous vous proposons la déclaration suivante :

```
class Histogramme {
   private:
      double a;
      double b;
      double delta;
      std::vector<int> bars;
      int nb_out;
   public:
      Histogramme(double a, double b, int N):
         a(a), b(b), delta((b-a)/N), bars(N,0), nb_out(0) {}
      bool operator+=(double x) ; //ajoute un point dans la bonne barre
      double lower_bound() const; //accède à a
      double upper_bound() const; //accède à b
      double nb_boxes() const; //accède au nombre de barres de l'histo.
      int out_of_domain() const; //accède à nb_out
      void print(std::ostream & out) const;// affiche sur le flux out
      void reset(); //réinitialise à 0 en conservant les paramètres
};
```

où `a` et `b` sont les extrêmités du segment $[a, b]$ sur lequel on réalise un histogramme à N boîtes, chacune de largeur $\delta = (b - a)/N$. Chaque case `bars[k]` indique le nombre de points qui tombent dans $[a + k\delta, a + (k + 1)\delta]$ avec $0 \leq k < N$. Si un point x tombe en dehors du segment $[a, b]$, c'est le compteur `nb_out` qu'on incrémente.

1. Que signifient les différentes parties des lignes 9 et 10 ?
2. Comprendre les différentes méthodes (et les `const` et références associés).
3. Pourquoi ajouter le champ privé `delta` qui est a priori redondant avec `a` et `b`
4. Écrire le code des accesseurs.
5. Écrire le code de la méthode `print` . Pour cela, on écrira sur chaque ligne deux nombres séparés par une espace : le centre de la k-ème boîte et la hauteur de la barre associées dans l'histogramme.

Question 2.14. Reprendre la question 2.10 à l'aide de cette classe. Constater la grande amélioration de la conception et de la lisibilité du programme.

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.

La deuxième partie des TP viendra plus tard...