Sorbonne Université
Master 2 Ingénierie mathématique
Programmation en C++

**Examen du 26 octobre 2018**
**Durée : 1 heure 30**

— **Tous** *les fichiers rendus devront contenir vos* **nom et prénoms**.
— *Le fait d'envoyer des fichiers qui ne se compilent pas correctement sera sanctionné.*
— *Les deux parties sont indépendantes entre elles.*

# 1 Marche aléatoire à boucles supprimées

Dans cette partie, on considère la *marche aléatoire à boucles supprimées*[1] $(\mathcal{X}_n)_{n\in\mathbb{N}}$, qui est une marche aléatoire autoévitante dans $\mathbb{Z}^2$. Plus précisément, on définit une suite $(X_n)_{n\in\mathbb{N}}$ de variables aléatoires à valeurs dans $\mathbb{Z}^2$ par

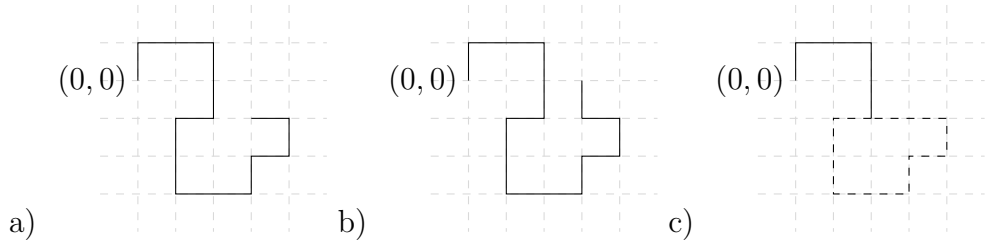
$$Z_0 = (0,0),\ \ Z_{n+1} = Z_n + Y_n,$$

où $Y_n$ suit la loi uniforme sur $\{(0,1),(0,-1),(1,0),(-1,0)\}$.

Ensuite, à chaque étape, on "supprime les boucles" de $Z_n$. Plus précisément, si $\mathcal{X}_n = (X_0, X_1, \dots, X_d)$ représente la marche obtenue après les $n$ premiers pas de $(Z_n)$, on définit

$$\mathcal{X}_{n+1} = \begin{cases} (X_0, X_1, \dots, X_i) & \text{si il existe } 0 \le i \le n \text{ avec } Z_{n+1} = X_i, \\ (X_0, X_1 \dots, X_d, Z_{n+1}) & \text{sinon.} \end{cases}$$

On se retrouve donc avec une marche aléatoire qui ne passe jamais deux fois au même point. Par exemple, sur la figure ci-dessous, la trajectoire a) peut mener à la trajectoire b) (si la marche fait un pas vers le haut), mais aussi à la trajectoire c) (si la marche fait un pas vers la gauche).




1. Écrire une classe `lerw`, contenant en membre privé une variable `marche` de type `std::vector<std::pair<int,int> >` et une variable `nb_pas` de type `int`[2]. La variable `nb_pas` correspondra au nombre de pas effectués par la marche *une fois les boucles effacées*. La trajectoire sera donc constituée des `nb_pas+1` premiers éléments du vecteur `marche`. En particulier, au moment de supprimer une boucle, il ne sera pas nécessaire de diminuer la taille du vecteur : il suffira de modifier `nb_pas`.

1. En anglais, *loop-erased random walk*, ou LERW.
2. Le type `std::pair<int,int>` correspond aux couples de deux variables de type `int`. Cette classe possède un constructeur prenant en argument deux `int` correspondant aux deux éléments du couple.

2. Écrire un constructeur pour la classe `lerw` créant une marche positionnée en $(0,0)$, avec `nb_pas==0`.
3. Écrire une méthode `reset` qui fait repartir la marche de son point de départ et "oublie" l'historique de la trajectoire.
4. Écrire une méthode `pas` qui renvoie la valeur de `nb_pas`.
5. Écrire une méthode `ajout` prenant en argument une référence sur un générateur aléatoire qui effectue un pas de la marche. Pour cela, suivant les cas, on devra ou bien modifier le contenu du vecteur `marche`, ou bien modifier la valeur de `nb_pas`.
6. Surcharger l'opérateur `<<` pour afficher les points (non effacés) successifs de la marche.
7. Écrire un programme qui simule 1000 réalisation de la marche jusqu'au premier instant où cette marche ait effectué 100 pas et qui affiche la distance moyenne à l'origine de ces marches.

# 2 Permutations

Dans cette partie on va implémenter une classe représentant les permutations, c'est-à-dire les bijections de $\{0, 1, \ldots, n-1\}$ dans lui-même. Une permutation $\sigma$ est caractérisée par les valeurs $(\sigma(i))_{i\in\{0,1,\ldots,n-1\}}$.

1. Écrire une classe `permutation` contenant en membres privés un vecteur `sigma` qui représentera la liste des $(\sigma(i))_{i\in\{0,1,\ldots,n\}}$, et un entier `n` correspondant à la taille $n$ de l'ensemble que l'on permute.
2. Écrire un constructeur pour la classe `permutation` prenant en argument un entier $n$ et créant la permutation identité sur $\{0, \ldots, n-1\}$ (donnée par $\sigma(i) = i$ pour tout $i$).
3. Surcharger l'opérateur `<<` pour afficher une permutation sous la forme `( 2 1 0 3 )` (pour la permutation de $\{0, 1, 2, 3\}$ qui envoie $0, 1, 2, 3$ respectivement sur $2, 1, 0, 3$).
4. Écrire une méthode `inv` qui renvoie l'inverse de la permutation courante (l'inverse de $\sigma$ est la permutation qui à $\sigma(i)$ associe $i$).
5. Écrire une méthode `t` qui prend en argument deux entiers `i` et `j` et échange les valeurs renvoyée par la permutation courante en `i` et en `j`.
6. Écrire une méthode `melange` qui prend en argument une référence sur un générateur aléatoire et qui permute les éléments de la permutation courante de sorte à ce que la permutation obtenue soit aléatoire de loi uniforme sur l'ensemble des permutations. Pour ce faire, on peut appliquer l'algorithme suivant : pour chaque entier naturel $i$ entre 1 et $n-1$, on tire au hasard un entier $m_i$ entre 0 et $i$ inclus, puis on échange les valeurs $\sigma(i)$ et $\sigma(m_i)$.
7. Surcharger l'opérateur `*` pour pouvoir composer deux permutations. On rappelle que la permutation composée $\sigma \circ \rho$ est la permutation qui à $i$ associe la valeur $\sigma(\rho(i))$.
   On ne vérifiera pas dans cette fonction que les deux permutations composées sont bien de même taille.
8. Surcharger l'opérateur `==` pour tester si deux permutations sont égales.
9. Écrire une méthode `ordre` qui renvoie l'ordre de la permutation, c'est-à-dire le plus petit entier $k$ tel que $\sigma^k$ est la permutation identité.
10. Simuler 10000 réalisations d'une permutation choisie uniformément parmi les permutations sur 10 éléments. Quel est l'ordre moyen de ces permutations ?