**Sorbonne Université** **M1 Mathématiques**
**2018–2019** **4M056 : Programmation en C++**

**TP noté 1 : éboulements de tas de sable**

À l'issue du TP, chaque binôme envoie par email trois fichiers `sandpile.hpp` , `sandpile.cpp` et `test_sandpile.cpp` . **Il est impératif de mettre en commentaire, dans tous les fichiers, les nom, prénom et n° d'étudiant de chacun des membres du binôme.**

Dans ce TP, on propose un modèle pour représenter l'effondrement de tas de sable (*sandpile* en anglais). Bien que le modèle du tas de sable puisse être défini sur un graphe quelconque, on travaille ici sur une grille carrée $m \times n$. Les sommets ont des coordonnées $(i, j)$, avec $0 \leq i \leq m-1$ et $0 \leq j \leq n-1$.

Une *configuration de tas de sable* est une fonction $h$ à valeurs entières positives sur ces sommets. La valeur de la configuration en $(i, j)$ est sa hauteur en ce sommet (c'est le nombre de grains de sables au-dessus de ce point). La configuration peut *s'ébouler* au sommet $(i, j)$ (on dit aussi que $(i, j)$ est instable) si la hauteur $h(i, j) \geq 4$. Si la configuration s'éboule, on enlève 4 grains en $(i, j)$ et on en donne un à chaque voisin (sur les bords ou dans les coins, comme le nombre de voisins est strictement inférieur à 4, le nombre total de grains de sable contenus dans la configuration décroît de 1 ou 2). Un résultat important de la théorie dit que si un éboulement peur avoir lieu en deux sommets, alors quelque soit l'ordre dans lequel ces éboulements sont réalisés, le résultat est le même. La configuration $h$ est *stable* s'il n'y a plus de sommet instable. La *stabilisation* de la configuration $h$ est la configuration obtenue à partir d'$h$ en faisant tous les éboulements possibles jusqu'à ce que la configuration soit stable.

Il se trouve que la stabilisation d'une configuration constante d'une grande région a des propriétés fractales remarquables. C'est ce que nous allons essayer de visualiser dans cette séance.

L'objet central de ce TP est la classe `Sandpile` décrivant ce modèle, qui est un peu plus riche que la donnée des valeurs de $h$ à proprement parler. Le début de déclaration de la classe `Sandpile` , écrite dans le fichier `sandpile.hpp` ressemble à ceci :

```
#ifndef _SANDPILE_HPP
#define _SANDPILE_HPP

#include <vector>
#include <list>

typedef std::pair<unsigned, unsigned> upair; // raccourci pour les paires d'entiers

class Sandpile {
  private:
    unsigned m;
    unsigned n;
    std::vector<unsigned> terrain; // terrain[i+j*m] donne la hauteur en (i,j)
    std::vector<unsigned> nb_ebouls;
    std::list<upair> next_ebouls;

  public:
    Sandpile(int m, int n, int k);
};
#endif
```

Le code des méthodes et fonctions un peu longues sera écrit dans le fichier d'implémentation `sandpile.cpp` . Décrivons les champs de la classe :

— les champs `m` et `n` précisent la taille du *terrain* (la grille rectangulaire) ;
— le champ `terrain` est la grille proprement dite, encodée sous forme de vecteur unidimensionnel : le sommet de coordonnées $(i, j)$ correspond à la case $i + m * j$ du vecteur ;
— le vecteur `nb_ebouls` enregistrera le nombre de fois où chaque sommet a été éboulé depuis la création de l'objet (il est initialisé à 0) ;
— la liste `next_ebouls` contient les paires correspondant à tous les sommets candidats à l'éboulement.

On rappelle qu'un objet $p$ de type `std::pair<S,T>` (ici raccourci en `upair` pour `S` et `T` égaux à `unsigned`) correspond à un couple, dont la première (resp. seconde) composante est accessible avec `p.first` (resp. `p.second`). Réciproquement, si `k` et `l` sont des `unsigned`, `upair(k,l)` construit le couple correspondant à $(k, l)$.

**1.** Écrire le constructeur `Sandpile::Sandpile(unsigned m0, unsigned n0, unsigned k)` qui fabrique une configuration sur une grille de taille `m0 x n0` dont la hauteur en tout sommet est `k`. Modifier la déclaration du constructeur pour que si la valeur `k` n'est pas indiquée, la hauteur choisie soit 4. Noter que si `k` $\geq 4$, tous les sommets sont candidats à l'éboulement, et si `k` $< 4$, la configuration est stable.

**2.** Écrire un premier accesseur aux éléments de terrain pour que si `s` est un objet de type `Sandpile`, alors `s(i,j)` renvoie la valeur du coefficient `i+j*m` du champ `terrain` de `s`.

**3.** Écrire une fonction globale amie `operator<<` permettant l'affichage d'une configuration sur l'écran ou son écriture dans un fichier. On convient que l'ordonnée $j$ correspond au numéro de ligne (de haut en bas), et $i$ au numéro de colonne (de gauche à droite). On supposera que toutes les hauteurs sont inférieures ou égales à 9, donc les hauteurs seront écrites les unes collées aux autres. L'affichage de `Sandpile(2,2)` devrait donner

```
44
44
```

**4.** Dans le fichier `test_sandpile.cpp`, écrire une fonction `main` dans laquelle une configuration `s` de taille $8 \times 8$ et de hauteur 5 est déclarée, et affichée.

**5.** On souhaite aussi des accesseurs/mutateurs prenant des paires en argument. Écrire deux versions de la méthode `h(upair p)` donnant accès pour une paire $p = (i, j)$ à l'élément `i+m*j` du vecteur `terrain`. On rappelle qu'on peut accéder à la première (resp. seconde) coordonnée d'une paire `p` avec `p.first` (resp. p.second). Les deux versions permettent l'accès soit en lecture seule (accesseur), soit en écriture (mutateur). Elles seront définies *inline*, à l'intérieur de la définition de la classe.

**6.** Écrire une méthode `std::list<upair> Sandpile::voisins(upair p)` qui renvoie la liste des paires de coordonnées correspondant aux voisins du sommet `p` : si `p=(i,j)` alors la liste renvoyée contient parmi les paires `(i-1,j)`, `(i+1,j)`, `(i,j-1)`, `(i,j+1)` celles qui sont encore des sommets de la grille.

**7.** Vérifier dans la fonction `main` que le nombre de voisins de la case $(0, 0)$ (resp. $(1, 2)$) de `s` est 2 (resp. 4).

**8.** Écrire une méthode privée[1] `void incr_and_test(upair p)` qui augmente de 1 la hauteur au sommet `p`, et qui l'ajoute à la liste `next_ebouls` si la hauteur est maintenant supérieure ou égale à 4.

**9.** Écrire la méthode `void Sandpile::eboul(upair p)`, qui essaie d'ébouler le sommet `p` dans la configuration courante. Si la hauteur en `p` est strictement inférieure à 4, on ne fait rien. Sinon, on augmente de 1 la valeur de la case correspondant à `p` dans `nb_ebouls`, on diminue de 4 la valeur de `h(p)`, puis on applique la méthode `incr_and_test` sur l'objet courant avec chacun des voisins de `p` comme argument. On pourra utiliser `std::for_each` (dans l'entête `<algorithm>`), à qui on passe les itérateurs de début et de fin de la liste des voisins, et la méthode encapsulée dans un lambda : `[this](const upair & p) {incr_and_test(p);}`. On rappelle que dans les coins et sur les bords, parmi les 4 grains supprimés, certains vont aux (2 ou 3) voisins, les autres sont perdus.

**10.** Déclarer et écrire le code de la méthode `void Sandpile::stabil()` qui stabilise (en modifiant) la configuration sur laquelle cette méthode est appelée, c'est à dire en éboulant tous les sommets candidats à l'éboulement (incluant ceux qui n'étaient peut-être pas instables au début, mais qui le sont devenus après ajouts de grains depuis leurs voisins).

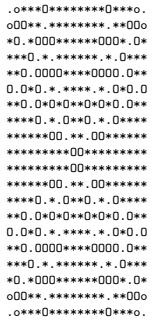
**11.** Écire une méthode `Sandpile::nb_total_ebouls()` qui renvoie le nombre total d'éboulements depuis la création de l'objet.

**12.** Depuis la fonction `main`, faire écrire dans le fichier `stab_8x8_5.txt` le nombre d'éboulement nécessaires à la stabilisation de `s` (on devrait trouver 564) et à la ligne, le résultat de sa stabilisation.

**13.** Déclarer ensuite une configuration `t` de taille $100 \times 100$ de hauteur 4, et faire afficher dans le terminal le résultat de la stabilisation (on veillera à ce que la fenêtre du terminal est assez grande).

**14.** (bonus). On peut embellir l'affichage de la configuration en remarquant qu'une configuration stabilisée a des hauteurs comprises entre 0 et 3. Modifier le code de l'affichage en déclarant une chaine de caractères `chars` égale à `".oO*"`. Si la hauteur de $(i, j)$ est entre 0 et 3, remplacer la valeur de la hauteur à l'affichage par le caractère correspondant dans `chars`. On peut aussi mettre de la couleur dans les environnements UNIX de la façon suivante. La chaîne de caractères `"\x1B[31m"` écrite dans le terminal fait que les caractères suivants sont écrits en rouge. En remplaçant 31 par 32, 33, 34, 35, 36, 37 on obtient les couleurs vert, jaune, bleu, magenta, cyan, blanc. Le code `"\x1B[0m"` rétablit les couleurs par défaut. Afficher chaque caractère avec une couleur différente (en pensant bien à rétablir les couleurs par défaut).

On peut aussi rajouter une méthode `volume_total` qui calcule le nombre de grains de sables contenus dans la configuration entière.



1. Cette méthode est privée car elle sera utilisée de façon interne par la classe, mais on ne veut pas qu'elle soit utilisable à l'extérieur de la classe, car elle correspond à une modification de la configuration qui ne correspond pas à un éboulement ou une stabilisation et pourrait être non sûre si utilisée sans précaution à l'extérieur de la classe.