Sorbonne Université **M1 Mathématiques**
2022–2023 **4M056 : Programmation en C++**

TP noté 1 : éboulements de tas de sable

À l'issue du TP, chaque étudiant envoie 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 numéro d'étudiant de chacun des membres du binôme.**

1 Présentation du modèle de tas de sable

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 présent dans la grille [1] (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 :

```
using upair=std::pair<unsigned, unsigned>;//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);
};
```

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) ;

1. Les voisins de (i, j) sont les sommets parmis $(i-1, j), (i+1, j), (i, j-1), (i, j+1)$ dont les coordonnées vérifient les contraintes pour appartenir à la grille.

— 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). `std::pair` est défini dans l'entête `utility`.

2 Fonctionalités de base : construction et affichage

Le but de cette section est de faire fonctionner le code suivant du fichier `test_sandpile.cpp` :

```
int main() {
  Sandpile s(3,2);
  std::cout << "La hauteur de sable en (1,1) est " << s(1,1) << std::endl;
  std::cout << "La configuration complète est :" << std::endl << s;
  return 0;
}
```

1. Compléter le fichier d'en-tête `sandpile.hpp` pour inclure ce qu'il faut pour notre implémentation de la classe. (L'ajout abusif sera sanctionné : ne mettez que le strict nécessaire.)

2. 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` ≥ 4, tous les sommets sont candidats à l'éboulement, et si `k` < 4, la configuration est stable.

3. 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`.

4. 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(3,2)` devrait donner

```
444
444
```

À ce stade, l'intégralité du code de la fonction `main` ci-dessus doit fonctionner. Ne passez à la suite que si le résultat affiché est celui attendu.

5. Dans le fichier `test_sandpile.cpp`, modifier la fonction main `main` pour que la configuration de sable `s` soit de taille 8×8 et de hauteur initiale 5.

3 Éboulements et stabilisation

6. 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`. 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.

7. Écrire une méthode `std::list<upair> Sandpile::voisins(upair p)` qui renvoie la liste des paires de coordonnées correspondant aux voisins du sommet de coordonnées `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.

8. Écrivez dans la fonction `main` les instructions permettant de vérifier que le nombre de voisins de la case $(0, 0)$ (resp. $(1, 2)$) de `s` est 2 (resp. 4).

9. Écrire une méthode privée[2] `incr_and_test` qui ne renvoie rien, qui prend en argument une paire de coordonnées `p`. Elle augmente de 1 la hauteur au sommet `p`, et l'ajoute à la liste `next_ebouls` si la hauteur est maintenant supérieure ou égale à 4.

10. Écrire la méthode `eboul` de la classe `Sandpile`, qui ne renvoie rien et prend en argument une paire de coordonnées `p`. Elle 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 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.

11. Déclarer et écrire le code de la méthode `int 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). L'entier renvoyé est le nombre d'éboulements qui ont eu lieu pendant la stabilisation.

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

13. 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.

14. Déclarer ensuite une configuration `t` de taille 100×100 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).

2. 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.

15. Ajouter à la classe `Sandpile` une méthode `add_random_grain` qui prend en argument un générateur de nombre pseudo-aléatoires (de la façon correcte) et un entier k, et qui k fois, choisit une position aléatoirement dans la grille, ajoute un grain de sable à cette position. Elle stabilise ensuite la configuration courante, et renvoie le nombre d'éboulements pendant la stabilisation. On modifiera la déclaration de la méthode pour que par défaut, si le paramètre k n'est pas précisé, un seul grain est ajouté.

16. Ajouter à la fonction `main` les lignes nécessaires pour ajouter 3 grains à la configuration `t` .

17. On peut définir une opération interne sur l'ensemble des configurations de sable : si `s` et `t` sont deux configurations, alors leur somme `s+t` est définie comme la stabilisation de la configuration dont la hauteur en chaque point est la somme des hauteurs des deux configurations. On supposera pour simplifier que `s` et `t` sont définies sur la même grille, et on initialisera toutes les cases du vecteur `nb_eboul` de la somme à 0. Définir l'opération d'addition comme une fonction globale.

4 Approfondissement

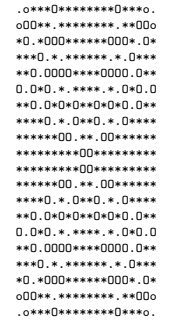
Ne commencer cette section que si le code de la question précédente donne le résultat escompté.

18. 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 `".oOO"`. 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).

19. Ajouter une méthode `volume_total` qui calcule le nombre de grains de sables contenus dans la configuration entière.

20. L'opération `+` permet de définir une structure de groupe sur le sous-ensemble des configurations de sable dites *récurrentes*. Une configuration obtenue comme la stabilisation d'une configuration initialement à une hauteur supérieur à 4 est récurrente. Sachant que le cardinal du groupe pour un terrain de taille 3×2 est 2415, quelle est la configuration jouant le rôle de l'identité ? Même question pour un terrain de taille 3×3 (cette fois le cardinal[3] est 100352).

21. Ajouter un constructeur qui prend les tailles `m0` et `n0` du terrain, ainsi qu'une `std::map<upair, unsigned>` qui contiendrait les hauteurs non-nulles d'une configuration. Donner un exemple d'appel.



3. de manière générale, le cardinal du groupe est le déterminant du laplacien discret sur la grille, avec conditions aux bords de Dirichlet. Il croît exponentiellement avec le produit mn.