**Introduction à l'algorithmique** **Hiver 2024**

---

IFT2125-6001 Évaluation (7.5%)

**Devoir 3 : Un devoir qui en vaut le détour!**

---

**Consignes générales :** À faire en équipe de 1 à 2 personnes. Remettre un seul pdf et les fichiers de code python et C++. Indiquez vos noms et matricules sur tous les fichiers (pdf et code).

Votre code doit obligatoirement passer les tests fournis avec le devoir, sinon une note de 0 sera attribuée aux exercices dont les tests de base ne sont pas réussis. D'autres tests seront rajoutés lors de la correction. Seulement les librairies standards de Python et C++ sont permises, sauf si mentionné autrement.

Il va de soit que le code doit être clair, bien indenté et bien commenté.

**La date de remise est le jeudi 28 mars à 22h30** La structure de la remise dans studium doit être la suivante :

```
StudiUM
├── devoir3_NOM1_NOM2.pdf
├── ClimbingDifficultyCalculator.cpp
├── ClimbingDifficultyCalculator.h
├── vitre.py
└── labyrinth_generator_creator.py
```

# 1 Trouvez l'erreur (10 points)

Chacune des preuves ou des énoncés suivants comporte au moins une erreur. Identifiez clairement où se trouvent toutes ces erreurs, et expliquez de façons appropriée pourquoi vous dites que ce sont des erreurs (par exemple, dites pourquoi une telle étape n'est pas correcte, donnez ce qui manque ou qui devrait être remplacé dans une définition, donnez des contre-exemples invalidant un énoncé, etc)

**Question 1:** (2 points) Preuve que $2^n \in \Omega(4^n)$ :

Par définition de $\Omega$, on cherche $c$ et $n_0$ tels que $2^n \geq c4^n, \forall n \geq n_0$.

$$2^n \geq c4^n \tag{1}$$
$$\frac{2^n}{4^n} \geq c \tag{2}$$
$$\left(\frac{1}{2}\right)^n \geq c \tag{3}$$
$$\log_{\frac{1}{2}} \left(\frac{1}{2}\right)^n \geq \log_{\frac{1}{2}} c \tag{4}$$
$$n \geq \log_{\frac{1}{2}} c \tag{5}$$

Comme $\log_{\frac{1}{2}} c$ est une constante, en prenant $c = \frac{1}{2}$, on aura $n \geq \log_{\frac{1}{2}} \frac{1}{2} = 1$, ce qui est vrai en prenant $n_0 = 1$.

Ainsi, nous avons trouvé $c = \frac{1}{2}$ et $n_0 = 1$ tels que $2^n \geq c4^n, \forall n \geq n_0$.

Nous avons donc $2^n \in \Omega(4^n)$. ■

**Question 2:** (2 points) Preuve que $2^n \in \Omega(4^n)$ :

En utilisant le critère de la limite,

$$\lim_{n\to\infty} \frac{2^n}{4^n} = \lim_{n\to\infty} \frac{2^n}{2^{2n}} \tag{1}$$
$$= \lim_{n\to\infty} \frac{\lg(2^n)}{\lg(2^{2n})} \tag{2}$$
$$= \lim_{n\to\infty} \frac{n}{2n} \tag{3}$$
$$= \lim_{n\to\infty} \frac{1}{2} \tag{4}$$
$$= \frac{1}{2} \tag{5}$$
$$\tag{6}$$

Comme $\frac{1}{2} \in \mathbb{R}^{>0}$, le critère de la limite nous permet de conclure que $2^n \in \Theta(4^n)$.

Ainsi, par définition de $\Theta(4^n)$, nous avons $2^n \in \Omega(4^n)$. ■

**Question 3:** (2 points) Voici la définition formelle de $O(f(n))$ :

$$\Omega(f(n)) = \left\{t(n) : \mathbb{N} \to \mathbb{R}^{\geq 0} \middle| (\exists c \in \mathbb{R}^{\geq 0})(\exists n_0 \in \mathbb{N})(\forall n \geq n_0)\ f(n) \geq cf(n)\right\}$$

**Question 4:** (2 points) Preuve que $(\exists x \in \mathbb{N})(\forall y \in \mathbb{N})\ x \geq y$ est vrai :

$$(\exists x \in \mathbb{N})(\forall y \in \mathbb{N})\ x \geq y \iff \quad (1)$$
$$(\forall y \in \mathbb{N})(\exists x \in \mathbb{N})\ x \geq y \iff \quad (2)$$

La dernière ligne est vraie car pour un certain $y$, on prend $x = y + 1 \geq y$ donc $x \geq y$.■

**Question 5:** (2 points)

Soit $f$ et $g$ deux fonctions $\mathbb{N} \to \mathbb{R}^{\geq 0}$ telles que $f(n) < g(n)\ \forall n \geq n_0$.

Alors $lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$

## 2 Escalade - Code C++ (20 points)

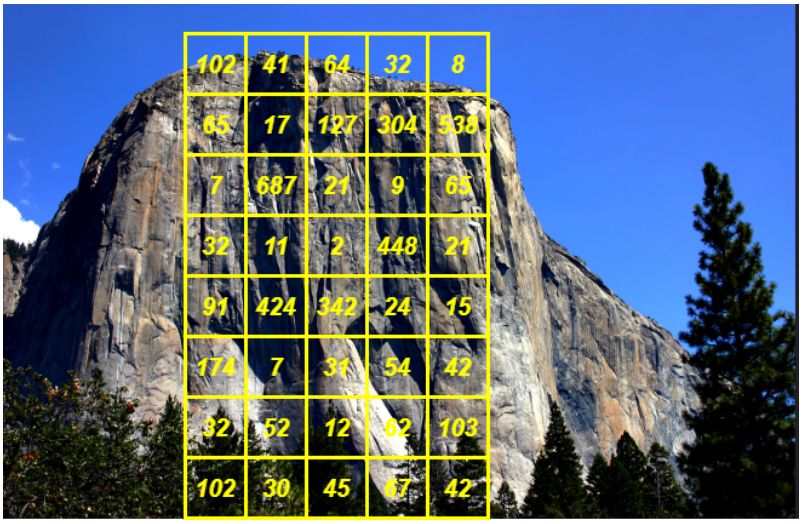
Dans ce problème, on vous demande de trouver la façon optimale pour une personne d'escalader un mur en utilisant le langage C++.

**Votre algorithme doit obligatoirement faire usage de la programmation dynamique.** Votre algorithme devrait être efficace pour avoir tous les points.

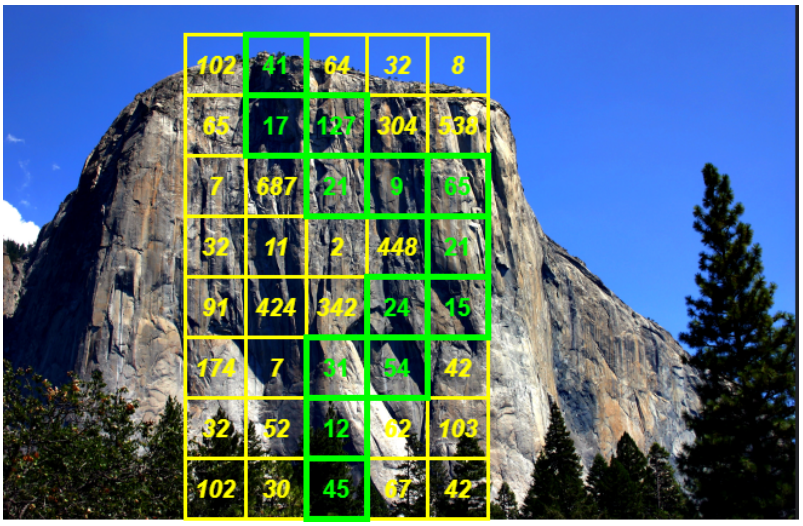
La parois du mur d'escalade sera représenté comme un rectangle découpé en $m \times n$ régions. Chaque région aura un degré de difficulté lui correspondant, représenté par un nombre entier. Votre algorithme reçoit donc en entrée une matrice d'entiers $m \times n$.

La personne qui grimpe commence au bas du mur, c'est à dire sur la $m^{\text{ième}}$ ligne de la matrice. Elle est libre de choisir dans laquelle des $n$ régions du bas du mur elle ira se positionner initialement pour débuter son ascension. Son objectif est d'atteindre n'importe laquelle des régions du haut du mur, soit de la première ligne de la matrice. Lors de son ascension, la personne qui grimpe ne peut pas descendre dans le mur. Les seules options de mouvement qu'elle possède est d'aller soit dans la zone immédiatement à droite, à gauche, ou en haut de celle où elle se trouve. La personne qui grimpe cherche le chemin le plus facile lui permettant d'escalader le mur, c'est à dire le chemin dont la somme des degrés de difficultés des régions visitées est minimal.

Par exemple, sur le mur suivant :




L'ascension optimale serait :




**Données** :

L'entrée est fournis en fichier .txt qui contient les éléments de la matrice. En sortie, on vous demande seulement la somme totale correspondant au chemin optimal.

```
Exemple :
102,41,64,32,8
65,17,127,304,538
7,687,21,9,65
32,11,2,448,21
91,424,342,24,15
174,7,31,54,42
32,52,12,62,103
102,30,45,67,42
--> doit donner la longueur du chemin 482
```

**Code**
Exemple d'appel :

```
Compilation :
g++ -o climbing_difficulty.exe climbing_difficulty.cpp
ClimbingDifficultyCalculator.cpp

Execution :
.\climbing_difficulty.exe wall1.txt
```

**Remise**
Compléter les fichiers *ClimbingDifficultyCalculator.cpp* et *ClimbingDifficultyCalculator.h* fournis, et ne remettre **uniquement** que ces fichiers. Ne **pas** remettre le fichier *climbing_difficulty.cpp*.

# 3 Testeurs de fenêtre - Code Python (20 points)

Dans ce problème, on vous demande de trouver le nombre minimal de tests qu'il faut faire en pire cas pour déterminer la solidité d'une fenêtre, en utilisant le langage Python.

**Votre algorithme doit obligatoirement faire usage de la programmation dynamique.** Votre algorithme devrait être efficace pour avoir tous les points.

Le ministère de l'éducation, des loisirs et du sport du Québec (MELS) souhaite installer de belles grandes fenêtres dans les écoles du Québec. Comme des enfants joueront à des jeux de ballons à proximité de ces fenêtres, le ministère souhaite d'abord évaluer la résistance d'une fenêtre à des lancers de ballons qui la frappe, en déterminant le seuil $s \in \mathbb{N}$ en Newtons à partir duquel une fenêtre se casse lorsqu'elle est frappée par un ballons lancé avec une force de $s$ Newtons.

Pour faire ces tests, vous recevez seulement $k$ fenêtres identiques du manufacturier. Vous disposez aussi d'une machine permettant de lancer un ballons avec une force de $x$ Newtons, pour $x$ un entier que vous pouvez modifier entre chaque lancer et pouvant aller de 1 à $N$ ($N$ est un paramètre fournis avec la machine, vous pouvez supposer $1 \leq s \leq N$).

Si un ballon est lancé sur une fenêtre avec une force supérieure ou égale à $s$, la fenêtre se cassera et ne pourra plus être utilisable pour les tests. Par contre, si le ballon est lancé avec une force strictement inférieure à $s$, la fenêtre ne se cassera pas et pourra être réutilisée.

Le but de votre algorithme est de calculer, étant donné $k$ et $N$, le nombre minimal de tests qu'il faudrait faire en pire cas pour trouver $s$. Autrement dit, calculer la valeur de

$$\min_{S \in \text{Stratégies}_{N,k}} \left\{ \max_{s} \{\text{nombre de lancés que fait la stratégie } S \text{ quand le seuil est } s\} \right\}$$

où $\text{Stratégies}_{N,k}$ est l'ensemble des stratégies valides (qui permettent de trouver $s$ sans manquer de fenêtres, peut importe la valeur de $s$) avec une force jusqu'à $N$ et $k$ fenêtres.

Par exemple, si on a qu'une seule fenêtre ($k = 1$) et $N = 4$, la seule solution est d'essayer avec $x = 1$, puis $x = 2$ si la fenêtre ne s'est pas cassée avant, puis $x = 3$ si la fenêtre ne s'est pas cassée avant (Pas besoin d'essayer pour $x = 4$, car si $s = 4$, le lancé $x = 3$ le détectera puisque la fenêtre ne se serait pas cassé) Dans le pire cas, il faudra donc faire 3 lancés, et donc l'algorithme devrait retourner 3.

Si on a deux fenêtres ($k = 2$) et $N = 4$, la solution optimale serait d'essayer d'abord avec $x = 2$, puis d'essayer avec $x = 1$ (si la fenêtre ne s'est pas cassé) ou $x = 3$ (si la fenêtre s'est cassée). Dans tous les cas, il faudra 2 lancés, donc l'algorithme retournerais 2.

Indices : Pour vous aider, voici quelques idées sur lesquelles vous pouvez réfléchir : Votre tableau de programmation dynamique devrait-il être en 2D ou 3D? Que devraient être les axes du tableau? nombre de vitres restantes, nombre de vitres cassées, nombre de tests effectués, plus petite force testée qui a fait casser la vitre, plus grande force testée qui n'a pas fait casser la vitre, nombres de niveaux de forces à tester, etc.

Exemples :

```
Exemple 1 :
4 1
--> doit donner 3

Exemple 2 :
4 2
--> doit donner 2
```

**Code**
Votre code prendra directement en paramètre dans le main le $N$ et le $k$. Une fonction nommée *vitre(N,k)* sera ensuite appelée et retournera la valeur recherchée sous forme de int. Il n'y a pas de lecture/écriture de fichier pour cette question.

Exemple d'appel :

```
python3 vitre.py 4 2
```

Un fichier *test_vitre.py* et des fichiers d'entrées vous sont fournis pour vous aider à tester votre code.

Exemple d'appel :

```
python3 test_vitre.py
```

**Remise**
Compléter le fichier *vitre.py* fournis, et ne remettre **uniquement** que ce fichier. Ne **pas** remettre le fichier *test_vitre.py*, ou n'importe quel autre fichier de test.

# 4 Labyrinthe - Code Python (50 points)

Dans ce problème, on vous demande d'implémenter 2 algorithmes de génération de labyrinthe, en utilisant le langage Python.

Vous êtes libre de choisir les algorithmes. Vous pouvez les inventer vous-même, ou implémenter des algorithmes qui existent déjà (dans un tel cas, citez-vous sources!). La page Wikipédia Maze generation algorithm pourrait être un bon point de départ.

Ensuite, vous devrez transformer votre labyrinthe en modèle 3D sous format .scad qui doit s'ouvrir avec OpenSCAD.

Voici deux exemples de résultat final :

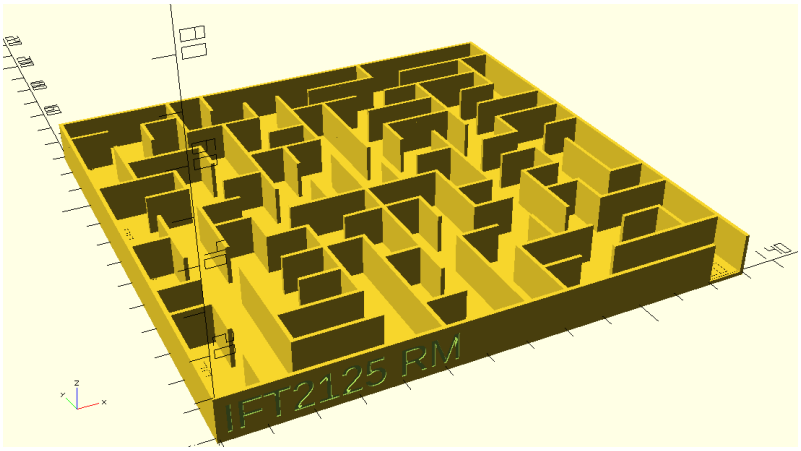



Figure 1: Un labyrinthe

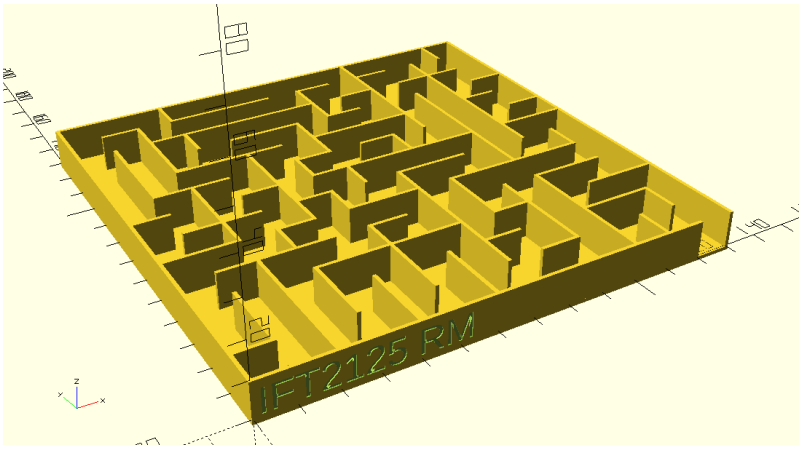



Figure 2: Un autre labyrinthe

Pour avoir tous les points, vos labyrinthes doivent avoir l'air "intéressant". Ils ne doivent pas être trop petit, il faut une bonne densité de murs et de couloirs, pas de zones inaccessibles, pas trop trivial à résoudre, etc.

**Code**

Votre code doit prendre en argument un nombre qui sera soit 1 ou 2, et qui permet de sélectionner lequel de vos deux algorithmes on utilise. Il doit ensuite écrire le labyrinthe en 3D dans un fichier .scad ouvrable avec OpenSCAD, et le nom du fichier doit contenir le numéro de l'algorithme utilisé (exemple : labyrinthe_algo1.scad et labyrinthe_algo2.scad).

Exemples d'appel :

```
python3 labyrinth_generator_creator.py 1
```

```
python3 labyrinth_generator_creator.py 2
```

Pour cette question, il n'y a pas de fichiers de tests qui vous sont fournis. On vous donnes deux exemples de fichiers .scad en guise de référence.

**Remise**
Compléter le fichier *labyrinth_generator_creator.py* fournis, et ne remettre **uniquement** que ce fichier. Ne **pas** remettre d'autres fichiers, inclants vos propres fichiers .scad (nous les générerons nous-mêmes).

**Bonus 10%**
Nous allons accorder un bonus maximal de 10% (pour le devoir) aux équipes qui font imprimer en 3D un exemplaire de labyrinthe créé avec leur propre code (impression gratuite à la bilio math-info, voir Indiana Delsart. Un atelier d'introduction sera donné spécifiquement pour les personnes du cours à un moment à décider). Le modèle doit être d'environ 8-12cm par 8-12cm avec une hauteur de 0.8-1.2cm avec 10x10 cases ou +. Il doit aussi porter le sigle du cours (IFT2125) et vos initiales, comme c'est le cas sur les figures plus hautes.