Projet tutoré : programmation du jeu Gratte-Ciel

L'objectif de ce projet tutoré est de coder en langage C le jeu Gratte-Ciel. Ce projet est à faire en binôme en dehors des heures de cours. Le code de ce programme devra impérativement respecter les spécifications données dans ce document. Cela signifie que la structure de données devra être la même que celle donnée ci-dessous et toutes les fonctions demandées (avec les mêmes en-têtes) devront être implémentées. *Il est cependant possible d'ajouter d'autres structures de données et/ou fonctions.* Ce travail sera évalué lors de 3 séances. À chaque séance, chaque binôme aura un créneau de 15 minutes défini préalablement pendant lequel il sera évalué.

La première partie décrit les règles du jeu Gratte-Ciel. La deuxième partie explique le travail à réaliser et la troisième décrit l'évaluation du projet en indiquant notamment le travail à présenter à chaque séance d'évaluation.

1 Description du jeu Gratte-Ciel

Le Gratte Ciel est un casse-tête logique à un joueur. Un quartier de gratte-ciels (par exemple le quartier de la Défense) a été représenté dans une grille. Chaque case contient un immeuble de 10, 20, 30, 40, 50 ou 60 étages (suivant la taille du quartier). Les immeubles d'une même rangée, ligne ou colonne, sont tous de tailles différentes. Les informations données sur les bords indiquent le nombre d'immeubles visibles sur la rangée correspondante par un observateur situé à cet endroit. Par exemple, si une ligne contient la disposition 20-40-30-10, deux immeubles sont visibles à partir de la gauche (le 20 et le 40) et trois immeubles sont visibles à partir de la droite (le 10, le 30 et le 40). Retrouvez la hauteur de chaque immeuble ! [1]

La figure suivante donne une grille de départ et la solution associée.

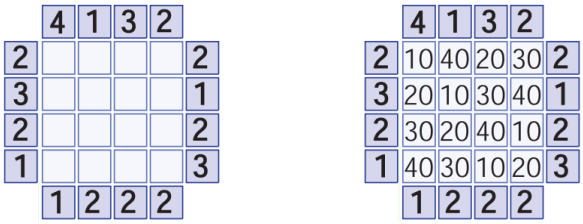

FIGURE 1 – Grille de départ et solution du jeu Gratte-ciel

Par la suite, l'emplacement des immeubles sera appelé *quartier* et le tour contenant les informations sur le nombre d'immeubles visibles *bordure*. Les deux (quartier + bordure) forment la *grille*. On parlera des bordures nord, sud, est et ouest pour désigner les bordures situées en haut, en bas, à droite et a gauche respectivement.

2 Travail à réaliser

Le développement du programme se divise en 3 parties. Chaque partie est décrite précisément dans ce qui suit.

1. L'explication et la figure 1 sont issus du magazine Tangeante Jeux et Stratégie. Il est possible d'essayer sur le Web le jeu à l'adresse `http://www.educmat.fr/categories/jeux/fiches_jeux/gratteciel/gratteciel.swf`.

2.1 Partie 1

Cette partie explique la structure de données à définir modélisant la grille du jeu Gratte-ciel. Elle donne aussi les fonctions de base à implémenter, permettant de modifier cette structure de données.

Question 1 : Définir le type structuré `Gratte_ciel` contenant les champs suivants :
- un pointeur d'entiers `grille`,
- un entier `n`,
- un entier `nb_cases_libres`.

Le type structuré `Gratte_ciel` permettra de représenter une instance du jeu Gratte-ciel. L'entier `n` représentera le nombre de cases par ligne et par colonne du quartier. La grille peut alors être vue comme un carré de `(n+2) * (n+2)` cases :
- `n * n` cases pour le quartier,
- `4 * n` cases pour la bordure,
- 4 cases inutilisées pour avoir un carré (les cases en haut à gauche, en haut à droite, en bas à gauche, en bas à droite).

La première ligne contient une case inutilisée, `n` cases pour la bordure nord puis une autre case inutilisée. Les `n` lignes suivantes contiennent une case pour la bordure ouest, `n` cases pour la ligne puis une case pour la bordure `est`. La dernière ligne contient une case inutilisée, `n` cases pour la bordure sud puis une case inutilisée.

Le carré représentant la grille sera codé par le champ `grille` qui sera un **tableau dynamique à une dimension**. Les lignes seront codées les unes à la suite des autres. Les `n+2` premières cases correspondront à la première ligne, les `n+2` suivantes à la deuxième, etc. La valeur de chaque élément de `grille` indiquera :
- le nombre d'immeubles visibles (0 si l'on n'a pas l'information, c'est-à-dire si la case est vide) si la case est une case de la bordure,
- le nombre de dizaines d'étages de l'immeuble (0 si la case est vide) si la case est une case du quartier.

Remarque : On supposera que l'on a la valeur 0 dans les cases inutilisées.

À titre d'exemple, la grille de départ de la figure 1 est codée avec le tableau `grille` suivant :

0	4	1	3	2	0	2	0	0	0	0	2	3	0	0	0	0	1	2	0	0	0	0	2	1	0	0	0	0	3	0	1	2	2	2	0
0					5					10					15					20					25					30					35	

La grille solution de la figure 1 est codée avec le tableau `grille` suivant :

0	4	1	3	2	0	2	1	4	2	3	2	3	2	1	3	4	1	2	3	2	4	1	2	1	4	3	1	2	3	0	1	2	2	2	0
0					5					10					15					20					25					30					35	

L'entier `nb_cases_libres` correspond au nombre de cases sans immeubles dans le quartier. Cet entier vaut 16 pour la grille de départ et 0 pour la grille solution pour la figure 1.

Question 2 : Définir la fonction `creer_gratte_ciel` prenant en paramètre un entier. La fonction allouera dynamiquement une variable de type `Gratte_ciel`, dont le champ `n` sera égal à la valeur passée en paramètre et le champ `grille` représentera un plateau de taille `(n+2) * (n+2)` ne contenant aucun immeuble ni aucune information sur la bordure. De plus, les cases inutilisées contiendront la valeur 0. (Toutes les cases contiendront donc la valeur 0.) La fonction retournera l'adresse de la variable de type `Gratte_ciel` allouée dynamiquement. La fonction devra vérifier que l'entier passé en paramètre est compris entre 2 inclus et 8 inclus avant de faire l'allocation. Si ce n'est pas le cas, elle devra retourner la valeur NULL sans faire d'allocation.

```
1 /*!
2  *  Fonction allouant dynamiquement un gratte_ciel dont l'adresse est retournée.
3  *
4  *  \param n : nombre d'immeubles par ligne et par colonne dans le quartier
```

```
 */
Gratte_ciel * creer_gratte_ciel(int nb) { ... }
```

Question 3 : Définir la fonction `detruire_gratte_ciel` prenant en paramètre un pointeur sur un `Gratte_ciel` alloué dynamiquement et libérant la mémoire pointée par ce dernier.

```
/*!
 *  Fonction désallouant dynamiquement la mémoire utilisée par le gratte-ciel passé en paramètre
 *
 *  \param p : pointeur sur le gratte-ciel à désallouer
 */
void detruire_gratte_ciel(Gratte_ciel * p) { }
```

Question 4 : Définir la fonction `indice_grille_valide` prenant en paramètre un pointeur sur un `Gratte_ciel` et un entier. La fonction devra retourner 1 si l'entier est un indice valide (de ligne ou de colonne) pour la grille, et 0 sinon.

```
/*!
 * Fonction retournant 1 si i correspond à un indice valide pour la grille du gratte-ciel
 * passé en paramètre, et 0 sinon
 *
 * \param p : pointeur sur un gratte-ciel
 * \param i : indice à tester
 */
int indice_grille_valide(Gratte_ciel * p, int i) { }
```

Question 5 : Définir la fonction `get_case` prenant en paramètre un pointeur sur une variable de type `Gratte_ciel`, un indice de ligne `i` et un indice de colonne `j`, et retournant la valeur de la case dans la grille. Si la case `(i,j)` n'existe pas, la fonction devra retourner -1. (Si `(i,j)` correspond à une case inutilisée, la fonction renvoie quand même la valeur de cette case.)

```
/*!
 * Fonction retournant la valeur de la case (i,j) de la grille du gratte-ciel p.
 * Si la case (i,j) n'existe pas, la fonction renvoie -1.
 *
 *  \param p : pointeur sur un gratte-ciel
 *  \param i : entier correspondant au numéro de ligne
 *  \param j : entier correspondant au numéro de colonne
 */
int get_case(Gratte_ciel *, int i, int j) { }
```

Exemple : Supposons que `pEx` est un pointeur sur une variable de type `Gratte_ciel` dont le champ `grille` correspond à la grille solution de la figure 1. L'exécution du code :

```
printf("Valeur sur la case (0,1) : %d\n", get_case(pEx,0,1));
printf("Valeur sur la case (1,1) : %d\n", get_case(pEx,1,1));
printf("Valeur sur la case (3,1) : %d\n", get_case(pEx,3,1));
```

affichera alors :

```
Valeur sur la case (0,1) : 4
Valeur sur la case (1,1) : 1
Valeur sur la case (3,1) : 3
```

En effet, la case (0,1), correspondant à une case de la bordure nord, contient la valeur 4. Les cases `(1,1)` et `(3,1)`, correspondant à des cases du quartier, contiennent respectivement les valeurs 1 et 3 puisque les immeubles sont de 10 et 30 étages pour ces cases.

Question 6 : Définir la fonction `set_case` permettant de modifier la valeur d'une case de la grille du gratte-ciel passé en paramètre. Si `(i,j)` correspond à une case de la grille et si la valeur `valeur` correspond à une valeur possible (un nombre entre 0 et `n` inclus), alors la fonction modifiera la case `(i,j)` avec la valeur `valeur` et retournera 1. Dans le cas contraire, la fonction retournera 0.

```
/*!
 *
 * Fonction modifiant la valeur de la case (i,j) de la grille du gratte-ciel p
 *
 *  \param p : pointeur sur un gratte-ciel
 *  \param i : entier correspondant au numéro de ligne
 *  \param j : entier correspondant au numéro de colonne
 *  \param valeur : valeur que l'on souhaite mettre dans la case (i,j)
 */
int set_case(Gratte_ciel * p, int i, int j, int val) { }
```

Exemple : upposons que `pEx` est un pointeur sur une variable de type `Gratte_ciel` dont le champ `grille` correspond à la grille solution de la figure 1. L'exécution du code :

```
set_case(pEx,1,1,4);
set_case(pEx,1,3,4);
set_case(pEx,1,4,0);
set_case(pEx,0,1,0);
set_case(pEx,5,1,0);
```

modifiera la première ligne du quartier. Celle-ci contiendra 3 immeubles de 40 étages et un emplacement vide. De plus, il n'y aura plus d'information sur le nombre d'immeubles visibles sur la première colonne.
Important : La fonction mettra automatiquement à jour le nombre de cases libres dans le quartier.

Question 7 : Définir la fonction `get_nb_cases_libres` qui retourne le nombre de cases libres du quartier du jeu Gratte-ciel passé en paramètre. Ce nombre est donné en fonction du champ `nb_cases_libres`.

```
/*!
 * Fontion retournant le nombre de cases libres du gratte-ciel passé en paramètre
 *
 *\param p : pointeur sur le gratte-ciel
 */
int get_nb_cases_libres(Gratte_ciel *p) { }
```

Question 8 : Définir la fonction `affichage_gratte_ciel`. Cette fonction devra afficher à l'écran la grille du gratte-ciel passé en paramètre.

```
/*!
 * Fonction affichant la grille sur le terminal
 *
 * \param p : pointeur sur le gratte-ciel que l'on souhaite afficher
 */
void affichage_gratte_ciel(Gratte_ciel * p) { ... }
```

Il existe plusieurs façons d'afficher le jeu du gratte-ciel. On peut distinguer 3 manières :
- Affichage facile : on affiche les valeurs de la grille sous forme d'un tableau à deux dimensions. Un exemple est donné par la figure 2.
- Affichage moyen : Les zéros ne sont pas affichés et chaque case prend plusieurs lignes et plusieurs colonnes. Un exemple est donné par la figure 3.

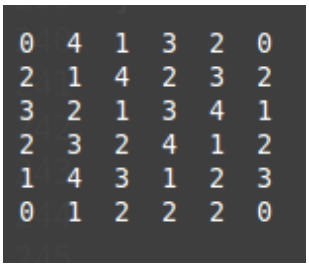

FIGURE 2 – Affichage facile de la grille solution de la figure 1

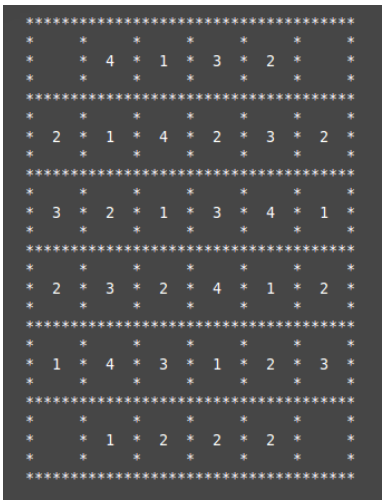

FIGURE 3 – Affichage moyen de la grille solution de la figure 1

— Affichage difficile : on réalise un affichage en couleurs et on affiche le nombre d'étages de chaque immeuble. On affiche également les numéros de ligne et de colonne (les lignes sont numérotées avec des lettres). Un exemple est donné par la figure 4. **Pour utiliser les couleurs du terminal, vous pouvez utiliser les fichiers `couleurs_terminal.c` et `couleurs_terminal.h` qui se trouvent sur la page :** `http://lipn.univ-paris13.fr/~lacroix/enseignement.php`. Les explications se trouvent dans le deuxième fichier en commentaires.

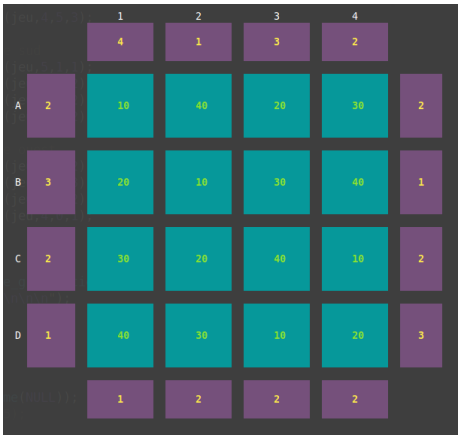

FIGURE 4 – Affichage difficile de la grille solution de la figure 1

Vous pouvez choisir l'une des trois méthodes d'affichage, sachant que plus la méthode est difficile, plus elle rapporte de points. (Cependant, il vaut vraiment mieux faire la plus méthode facile correctement que la méthode difficile fausse !)

2.2 Partie 2

Les fonctions à implémenter dans cette partie permettent notamment de déterminer si le placement des immeubles dans le quartier correspond à une solution du jeu.

Question 9 : Définir la fonction `est_case_bordure` prenant en paramètre un pointeur sur un gratte-ciel et deux entiers `i` et `j`, et retournant 1 si la case `(i,j)` est une case de la bordure, et 0 sinon.
Attention : La fonction doit retourner 0 si la case est une des quatre cases inutilisées de la bordure.

```
/*!
 * Fonction retournant 1 si (i,j) est une case de la bordure (n'est pas une case inutilisée),
 * 0 sinon
 *
 * \param p : pointeur sur gratte-ciel
 * \param i : indice de ligne
 * \param j : indice de colonne
 */
int est_case_bordure(Gratte_ciel * p, int i, int j) { }
```

Question 10 : Définir la fonction `nb_immeubles_visibles` prenant en paramètre un pointeur sur un gratte-ciel et une case `(i,j)` de la bordure. La fonction doit retourner le nombre d'immeubles visibles depuis la case `(i,j)`.
Attention : Si `(i,j)` n'est pas une case de la bordure ou si elle correspond à une case inutilisée, la fonction doit renvoyer -1.

```
/*!
 * Fonction retournant le nombre d'immeubles visibles depuis la case (i,j)
 *
 * si (i,j) n'est pas une case de la bordure ou si c'est une case inutilisée,
 * la fonction retourne -1
 *
 * \param p : pointeur sur gratte-ciel
 * \param i : indice de ligne
 * \param j : indice de colonne
 */
int nb_immeubles_visibles(Gratte_ciel * p, int i, int j) { }
```

Question 11 : Définir la fonction `calcul_bordure` prenant en paramètre un pointeur sur un gratte-ciel. La fonction met à jour les valeurs de toutes les cases de la bordure (sauf les quatre cases inutilisées). Dans chaque case, elle met le nombre d'immeubles visibles depuis cette case.

```
/*!
 * Fonction mettant à jour la bordure avec le nombre d'immeubles du quartier visibles
 * depuis chaque case.
 *
 * \param p : pointeur sur gratte-ciel
 */
void calcul_bordure(Gratte_ciel *p) { }
```

Question 12 : Définir la fonction `valeurs_differentes_ligne` prenant en paramètre un pointeur sur un gratte-ciel et un indice de ligne. La fonction retourne 1 si tous les immeubles de cette ligne sont de hauteur différente, et 0 sinon.
Attention : Si l'indice ne correspond pas à un indice de ligne du quartier, la fonction retourne 0.

```
/*!
 * Fonction retournant 1 si les immeubles de la ligne ind sont tous de taille différente
 *
 *  \param p : pointeur sur gratte-ciel
 *  \param ind : indice de ligne
 */
int  valeurs_differentes_ligne (Gratte_ciel * p, int ind) { }
```

Question 13 : Définir la fonction `valeurs_differentes_colonne` prenant en paramètre un pointeur sur un gratte-ciel et un indice de colonne. La fonction retourne 1 si tous les immeubles de cette colonne sont de hauteur différente, et 0 sinon.
Attention : Si l'indice ne correspond pas à un indice de colonne du quartier, la fonction retourne 0.

```
/*!
 * Fonction retournant 1 si les immeubles de la colonne ind sont tous de taille différente
 *
 *  \param p : pointeur sur gratte-ciel
 *  \param ind : indice de colonne
 */
int  valeurs_differentes_colonne (Gratte_ciel * p, int ind) { }
```

Question 14 : Définir la fonction `valeurs_differentes_quartier` prenant en paramètre un pointeur sur un gratte-ciel. La fonction retourne 1 si dans le quartier, il n'y a pas deux immeubles de même hauteur sur une même ligne ou sur une même colonne.

```
/*!
 * Fonction retournant 1 s'il n'y a pas deux immeubles de même hauteur
 * dans une ligne ou une colonne du quartier
 *
 *  \param p : pointeur sur gratte-ciel
 */
int  valeurs_differentes_quartier (Gratte_ciel * p) { }
```

Question 15 : Définir la fonction `bordure_correcte` prenant en paramètre un pointeur sur un gratte-ciel. La fonction teste, pour chaque case de la bordure contenant une valeur non-nulle, si cette valeur correspond au nombre d'immeubles que l'on voit dans le quartier à partir de cette case. Si les valeurs sont cohérentes pour toutes les cases, la fonction retourne 1. Elle retourne 0 s'il y a la moindre erreur.

```
/*!
 * Fonction retournant 1 si les valeurs non-nulles de la bordure correspondent
 * au nombre d'immeubles que l'on aperçoit dans le quartier à partir de la bordure,
 * et 0 sinon.
 *
 *  \param p : pointeur sur gratte-ciel
 */
int bordure_correcte(Gratte_ciel * p) { }
```

Question 16 : Définir la fonction `quartier_est_solution` prenant en paramètre un pointeur sur un gratte-ciel. La fonction retourne 1 si le placement des immeubles dans le quartier correspond à

une solution du jeu. Autrement dit, la fonction retourne 1 si le quartier est plein, il n'y a pas deux immeubles ayant la même hauteur dans une même ligne ou une même colonne, et si le placement des immeubles correspond aux informations données dans la bordure. La fonction retourne 0 sinon.

```c
/*!
 * Fonction retournant 1 si le placement des immeubles dans le quartier correspond
 * à une solution du jeu du gratte-ciel, et 0 sinon.
 *
 *  \param p : pointeur sur gratte-ciel
 */
int quartier_est_solution(Gratte_ciel * p) { }
```

2.3 Partie 3

Les fonctions à implémenter dans cette partie permettent de générer aléatoirement des problèmes du jeu Gratte-ciel et de jouer à ce jeu.
Remarque : La génération de parties de Gratte-ciel est assez complexe. La grille est normalement faite de telle manière qu'il y a exactement une seule solution. De plus, il est possible que certaines cases de la bordure ne contiennent pas de valeur. Dans ce cas, on ne sait pas combien d'immeubles sont visibles depuis cet endroit. Dans ce projet, on utilisera un processus de génération aléatoire simplifié. Les problèmes que l'on générera pourront admettre plusieurs solutions. De plus, aucune information ne manquera sur la bordure.

Question 17 : Définir la fonction `permutation_lignes` prenant en paramètre un pointeur sur un gratte-ciel et deux entiers. Si les deux entiers sont différents et correspondent à des indices de lignes du quartier, alors la fonction permutera les deux lignes dont les indices sont donnés par les entiers passés en paramètre. Autrement, la fonction ne fera aucune modification sur le gratte-ciel.
Remarque : La fonction ne modifiera que le quartier. La bordure ne sera pas modifiée.

```c
/*!
 * Fonction permutant les lignes i et j dans la grille si i et j correspondent à des indices
 * différents de lignes du quartier.
 *
 * \param p : pointeur sur gratte-ciel
 * \param i : indice de ligne
 * \param j : indice de ligne
 */
void permute_lignes(Gratte_ciel * p, int i, int j) { }
```

Question 18 : Définir la fonction `permutation_colonnes` prenant en paramètre un pointeur sur un gratte-ciel et deux entiers. Si les deux entiers sont différents et correspondent à des indices de colonnes du quartier, alors la fonction permutera les deux colonnes dont les indices sont donnés par les entiers passés en paramètre. Autrement, la fonction ne fera aucune modification sur le gratte-ciel.
Remarque : La fonction ne modifiera que le quartier. La bordure ne sera pas modifiée.

```c
/*!
 * Fonction permutant les colonnes i et j dans la grille si i et j correspondent à des indices
 * différents de colonnes du quartier.
 *
 * \param p : pointeur sur gratte-ciel
 * \param i : indice de colonne
 * \param j : indice de colonne
 */
void permute_colonnes(Gratte_ciel * p, int i, int j) { }
```

Question 19 : Définir la fonction `permute_nombres` prenant en paramètre un pointeur sur un gratte-ciel et deux entiers. S'il n'y a pas de cases libres dans le quartier et si les deux entiers passés en paramètre sont différents et sont des nombres compris entre 1 et n, alors la fonction permutera dans le quartier les valeurs. À titre d'exemple, l'appel de cette fonction avec les entiers 1 et 2 modifiera la grille en inversant les immeubles de 10 étages avec les immeubles de 20 étages.
Remarque : La fonction ne modifiera que le quartier. La bordure ne sera pas modifiée.

```
/*!
 * Fonction permutant les valeurs nb1 et nb2 dans le quartier si nb1 et nb2 correspondent à des
 * valeurs différentes pour des immeubles (entre 1 et n inclus) et si le quartier est entièrement
 * rempli.
 *
 * \param p : pointeur sur gratte-ciel
 * \param nb1 : première valeur
 * \param nb2 : deuxième valeur
 */
void permute_nombres(Gratte_ciel * p, int nb1, int nb2) { }
```

Question 20 : Définir la fonction `remplir_quartier` prenant en paramètre pointeur sur un gratte-ciel. La fonction remplira tout le quartier de la manière suivante :

- les immeubles de 10 étages sont sur la diagonale (en haut à gauche/en bas à droite),
- l'immeuble voisin de droite fait toujours 10 étages de plus (sauf si c'est l'immeuble de 10 étages),
- l'immeuble de la première case d'une ligne (sauf pour la première ligne) fait 10 étages de plus que l'immeuble de la dernière case de cette ligne.

Pour un quartier de taille 4 * 4, le remplissage de la fonction est donné dans la figure 5.

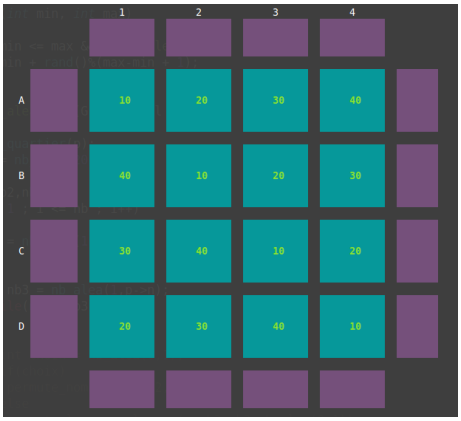

FIGURE 5 – Remplissage du quartier avec la fonction `remplir_quartier`

Remarque : La fonction ne modifiera que le quartier. La bordure ne sera pas modifiée.

```
/*!
 * Fonction remplissant le quartier selon un schéma précis
 * (10 étages sur la diagonale et +10 étgages pour le voisin de droite)
 *
 * \param p : pointeur sur gratte-ciel
 */
void remplir_quartier(Gratte_ciel * p) { }
```

Question 21 : Définir la fonction `nombre_aleatoire` prenant en paramètre deux entiers `min` et `max`. Cette fonction retournera un entier aléatoire compris entre `min` et `max` inclus. Si `min` est strictement supérieur à `max`, alors la fonction retournera -1.

```
/*!
 * Fonction retournant un nombre entier aléatoire entre min et max inclus, -1 si min > max.
 *
 * \param min : borne inférieure de l'intervalle
 * \param max : borne supérieure de l'intervalle
 */
int nombre_aleatoire(int min, int max) { }
```

Question 22 : Définir la fonction `quartier_aleatoire` remplissant un quartier de manière aléatoire sans qu'il y ait deux immeubles de même taille sur une même ligne ou colonne. Pour cela, la fonction remplira le quartier avec la fonction `remplir_quartier` définie avant. Elle tirera ensuite un nombre aléatoire `nb` entre 50 et 100 inclus. Elle fera alors `nb` fois : permuter deux lignes, permuter deux colonnes ou permuter deux valeurs (avec une probabilité égale pour chacune de ces possibilités).
Remarque : La fonction ne modifiera que le quartier. La bordure ne sera pas modifiée.

```
/*!
 * Fonction générant aléatoirement un quartier de manière qu'il n'y ait pas deux immeubles
 * de même taille sur une même ligne ou colonne.
 *
 * \param p : pointeur sur gratte-ciel
 */
void quartier_aleatoire (Gratte_ciel * p);
```

Question 23 : Définir la fonction `creer_gratte_ciel_aleatoire` prenant en paramètre un entier `n`. La fonction allouera un gratte-ciel de taille `(n+2) * (n+2)` (`n*n` pour le quartier). Elle remplira le quartier de manière aléatoire en utilisant la méthode `quartier_aleatoire` définie précédemment, mettra à jour la bordure avec le nombre d'immeubles visibles à chaque endroit puis effacera les immeubles dans le quartier. La fonction retournera finalement l'adresse du gratte-ciel ainsi créé.

```
/*!
 * Fonction créant une instance du jeu Gratte-ciel. Le quartier est généré aléatoirement,
 * la bordure est mise à jour et le quartier est finalement effacé.
 * La fonction renvoie l'adresse du gratte-ciel ainsi créé.
 *
 * \param n : Nombre d'immeubles par ligne et par colonne du quartier
 */
Gratte_ciel * creer_gratte_ciel_aleatoire (int n) { }
```

Question 24 : Définir la fonction `mouvement`. Cette fonction prend en paramètre une partie. Elle permet au joueur de saisir un emplacement et une taille d'immeuble (0 pour supprimer un immeuble) et met l'immeuble à l'emplacement indiqué dans le quartier. La saisie sera répétée jusqu'à ce que l'utilisateur saisisse un mouvement valide ou la valeur `stop`. Si l'utilisateur saisit `stop`, aucun mouvement n'est effectué et la fonction retourne 0. Sinon, la fonction effectue le mouvement et renvoie 1.

Important : Le format de saisie d'un mouvement valide est précis. L'utilisateur doit saisir une chaîne. Le premier caractère est une lettre pour désigner le numéro de ligne (`A` pour la première ligne du quartier, `B` pour la deuxième ligne, etc). Le deuxième caractère est le numéro de colonne de quartier (`1` pour la première colonne, etc). L'utilisateur saisit ensuite un espace puis une taille

d'immeuble (10, 20, ...). Ainsi, pour mettre un immeuble de 40 étages dans la case de la première ligne et de la première colonne du quartier, l'utilisateur doit saisir la chaîne `A1 40` au clavier.

Remarque : Comme il y a moins de 10 colonnes dans le quartier, le numéro de colonne s'écrit avec un seul caractère. Pour lire une saisie clavier comportant des espaces, on utilisera la fonction `saisie_avec_espaces` donnée dans le fichier `saisie_avec_espaces.c`. L'en-tête et les explications de cette fonction sont dans le fichier `saisie_avec_espaces.h`.

```
/*!
 * Fonction permettant au joueur de saisir un emplacement d'immeuble et modifiant le quartier en
 * conséquence. Si le joueur veut arrêter, il saisit la chaîne "stop". La saisie est répétée
 * jusqu'à ce que la saisie corresponde à une saisie d'un immeuble sur une case.
 *
 * Format : "B1 30" pour mettre un immeuble de 30 étages sur la case du quartier de la 2ème ligne
 * et de la 1ère colonne
 */
int mouvement(Gratte_ciel * p) { }
```

Question 25 : Définir la fonction `jouer`. Cette fonction prend en paramètre un entier `n`. Elle crée une partie aléatoire dont le quartier est de taille `n * n`. Le joueur saisit des mouvements jusqu'à ce qu'il saisisse `stop` ou que la grille soit pleine. Si l'utilisateur a saisi `stop`, la fonction affiche `jeu arrêté`. Autrement, elle affiche `partie gagnée` ou `partie perdue` suivant le résultat de la grille.

```
/*!
 * Fonction permettant de jouer au jeu du gratte-ciel.
 *
 * \param n : Taille du quartier de la partie à créer
 */
void jouer(int n) { }
```

Question 26 : Écrire le programme principal permettant de jouer au jeu Gratte-ciel. La taille de la grille est demandée à l'utilisateur jusqu'à ce qu'il saisisse un nombre compris entre 2 inclus et 8 inclus.

Question 27 : Séparer le code en plusieurs fichiers sources et créer un Makefile permettant la compilation du programme.

3 Évaluation du projet

Le projet tutoré sera évalué au cours de 3 séances, chaque séance correspondant à l'évaluation d'une partie (autrement dit, le travail à présenter lors de la première séance correspond à celui décrit dans la partie 1, etc). L'évaluation portera sur :

- **l'implémentation** des fonctions demandéees à chaque séance,
- **la pertinence des jeux d'essai réalisés** permettant de tester les différents cas de figure pour chaque fonction demandée (pour les parties 2 et 3),
- **les réponses aux questions** posées lors de l'évaluation.

La partie relatives aux tests (ou jeux d'essai) est très importante. Pour avoir une idée précise des tests demandés, vous trouverez un exemple de tests pour la première partie dans le fichier `test_partie1.c` disponible sur le site. Assurez-vous que votre code passe tous les tests avec succès. Il faut ensuite effectuter des tests similaires pour les parties 2 et 3.