Projet tutoré : programmation du jeu Takuzu

L'objectif de ce projet tutoré est de coder en langage C le jeu Takuzu, aussi appelé binairo. 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 Takuzu. 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.

Remarque : Certains tests doivent être évalués avec la fonction `assert`. Cette fonction est une fonction du langage C dont la déclaration se trouve dans le fichier d'en-tête `assert.h`. Elle prend en paramètre un test. Lors de l'appel de cette fonction, si le test est évalué à `0`, un message d'erreur est affiché et le programme s'arrête. Un exemple d'utilisation de la fonction `assert` est donné dans le fichier `utilisation_assert.c` sur la page Web :
`http://lipn.univ-paris13.fr/~lacroix/enseignement.php`.

1 Description du jeu Takuzu

Le Takuzu [1] est un jeu de réflexion d'origine belge créé par Frank Coussement et Peter De Schepper en 2009, et déposé à l'époque sous le nom de Binairo. Il est basé sur la logique un peu comme le sudoku. Dans notre cas, il s'agit de grilles carrées de taille 4×4, 6×6 ou 8×8. Chaque grille ne contient que des 0 et des 1, et doit être complétée en respectant ces règles :

— autant de 1 et de 0 sur chaque ligne et sur chaque colonne ;
— pas plus de 2 chiffres identiques côte à côte ;
— 2 lignes ne peuvent pas être identiques ;
— 2 colonnes ne peuvent pas être identiques.

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

	1		0
		0	
	0		
1	1		0

0	1	1	0
1	0	0	1
0	0	1	1
1	1	0	0

FIGURE 1 – Grille de départ et solution du jeu Takuzu

Par la suite, une case de la grille sera appelée *cellule*. Les cellules contenant au départ des nombres seront appelées *cellules initiales*.

1. Explications et exemple issus du site Web Wikipedia.

2 Travail à réaliser

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

2.1 Partie 1

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

Question 1 : Définir le type structuré `cellule` contenant les champs suivants :
- un entier `val`,
- un entier `initial`.

Le type structuré `cellule` permet de représenter une cellule de la grille du Takuzu. L'entier `val` représente le nombre contenu dans la cellule (il sera égal à -1 si la cellule est vide). L'entier `initial` est égal à 1 si c'est une cellule initiale, et 0 sinon.

Question 2 : Définir le type structuré `grille` contenant les champs suivants :
- un pointeur de cellules `tab`,
- un entier `n`.

Le type structuré `grille` permet de représenter une instance du jeu Takuzu. L'entier `n` représente le nombre de cellules par ligne et par colonne de la grille. La grille peut alors être vue comme un carré de `n*n` cellules. Ce carré représentant la grille sera codé par le champ `tab` qui sera un **tableau dynamique (de cellules) à une dimension**. Les lignes seront codées les unes à la suite des autres. Les `n` premières cases correspondront à la première ligne, les `n` suivantes à la deuxième, etc.

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

-1	1	-1	0	-1	-1	0	-1	-1	0	-1	-1	1	1	-1	0
0	1	0	1	0	0	1	0	0	1	0	0	1	1	0	1

Chaque cellule du tableau correspond à une cellule. Le nombre du haut correspond à la valeur de `val`, le nombre du bas à la valeur de `initial`.

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

0	1	1	0	1	0	0	1	0	0	1	1	1	1	0	0
0	1	0	1	0	0	1	0	0	1	0	0	1	1	0	1

Question 3 : Définir la fonction `creer_grille` prenant en paramètre un entier. La fonction allouera dynamiquement une variable de type `grille`, dont le champ `n` sera égal à la valeur passée en paramètre et le champ `tab` représentera une grille `n` × `n` de cellules vides (sans cellule initiale). La fonction retournera l'adresse de la variable de type `grille` allouée dynamiquement. La fonction vérifiera d'abord avec la fonction `assert` que le nombre passé en paramètre vaut 4, 6 ou 8.

```
1  /**
2   * Fonction allouant dynamiquement une grille dont l'adresse est retournée.
3   * @param n : nombre de lignes/colonnes (4, 6, ou 8)
4   * @return  : adresse de la grille, NULL en cas d'erreur
5   */
6  grille * creer_grille(int n);
```

Question 4 : Définir la fonction `detruire_grille` prenant en paramètre un pointeur sur une `grille` allouée dynamiquement et libérant la mémoire pointée par ce dernier.

```c
/**
 * Fonction désallouant dynamiquement la grille passée en paramètre.
 * @param g : grille à désallouer
 */
void detruire_grille(grille * g);
```

Question 5 : Définir la fonction `est_indice_valide` prenant en paramètre un pointeur sur une `grille` et un entier. La fonction retourne 1 si l'entier correspond à un indice de ligne ou de colonne valide (donc compris entre 0 et le nombre de lignes/colonnes de la grille -1).

```c
/**
 * Fonction retournant 1 si l'indice est valide par rapport à la grille.
 * @param g      : grille
 * @param indice : entier
 * @return       : 1 si indice est un indice valide pour g, 0 sinon
 */
int est_indice_valide(grille * g, int indice);
```

Question 6 : Définir la fonction `est_cellule` prenant en paramètre un pointeur sur une `grille` et deux entiers `i` et `j`. La fonction retourne 1 si `(i,j)` correspond à une cellule de la grille (c'est-à-dire qu'il existe une cellule ligne `i` et colonne `j`), et 0 sinon.

```c
/**
 * Fonction retournant 1 si (i,j) est une cellule de la grille.
 * @param g : grille
 * @param i : numéro de ligne
 * @param j : numéro de colonne
 * @return  : 1 si (i,j) correspond à une cellule de g, 0 sinon
 */
int est_cellule(grille * g, int i, int j);
```

Question 7 : Définir la fonction `get_val_cellule` prenant en paramètre un pointeur sur une variable de type `grille`, un indice de ligne `i` et un indice de colonne `j`, et retournant la valeur de l'attribut `val` de la cellule `(i,j)`. La fonction vérifiera d'abord à l'aide de la fonction `assert` que `(i,j)` correspond à une cellule de la grille.

```c
/**
 * Fonction retournant la valeur de la cellule (i,j) de la grille g.
 *
 * @param g : grille
 * @param i : numéro de ligne
 * @param j : numéro de colonne
 */
int get_val_cellule(grille * g, int i, int j);
```

Exemple : Supposons que `p_ex` est un pointeur sur une variable de type `grille` correspondant à la grille de départ de la figure 1. L'exécution du code :

```c
printf("Valeur sur la cellule (0,1) : %2d\n", get_val_cellule(p_ex,0,1));
printf("Valeur sur la cellule (1,1) : %2d\n", get_val_cellule(p_ex,1,1));
printf("Valeur sur la cellule (3,3) : %2d\n", get_val_cellule(p_ex,3,3));
```

affichera alors :

```
Valeur sur la cellule (0,1) :  1
Valeur sur la cellule (1,1) : -1
Valeur sur la cellule (3,3) :  0
```

En effet, la cellule `(0,1)`, correspondant à la deuxième cellule de la première ligne, contient la valeur 1. La cellule `(1,1)` est vide donc elle contient la valeur -1. La cellule `(3,3)` (dernière cellule de la dernière ligne) contient la valeur 0.

Question 8 : Définir la fonction `set_val_cellule` prenant en paramètre trois entiers `i`, `j` et `valeur`. La fonction vérifie d'abord, avec la fonction `assert`, que `(i,j)` est une cellule de la grille et `valeur` est compris entre -1 et 1. Elle modifie le champ `val` de la cellule `(i,j)` avec la valeur `valeur`.

```c
/**
 * Fonction modifiant la valeur de la cellule (i,j) de la grille g avec
 * la valeur passée en paramètre.
 * @param g : grille
 * @param i : indice de ligne
 * @param j : indice de colonne
 * @param valeur : valeur à mettre dans le champ val de la cellule (i,j)
 */
void set_val_cellule(grille * g, int i, int j, int val);
```

Exemple : Supposons que `p_ex` est un pointeur sur une variable de type `grille` correspondant à la grille de départ de la figure 1. L'exécution du code :

```c
set_val_cellule(p_ex,0,0,0);     set_val_cellule(p_ex,0,2,1);
set_val_cellule(p_ex,1,0,1);     set_val_cellule(p_ex,1,1,0);
set_val_cellule(p_ex,1,3,1);     set_val_cellule(p_ex,2,0,0);
set_val_cellule(p_ex,2,2,1);     set_val_cellule(p_ex,2,3,1);
set_val_cellule(p_ex,3,2,0);
```

modifiera la grille qui correspondra alors à la grille solution de la figure 1.

Question 9 : Définir la fonction `est_cellule_initiale` prenant en paramètre deux entiers `i` et `j`. La fonction retournera 1 si la cellule `(i,j)` est une cellule initiale, et 0 sinon. La fonction vérifiera d'abord à l'aide de la fonction `assert` que `(i,j)` correspond à une cellule de la grille.

```c
/**
 * Fonction retournant 1 si la cellule (i,j) est une cellule initiale,
 * et 0 sinon.
 * @param g : grille
 * @param i : indice de ligne
 * @param j : indice de colonne
 */
int est_cellule_initiale(grille * g, int i, int j);
```

Question 10 : Définir la fonction `est_cellule_vide` prenant en paramètre deux entiers `i` et `j`. La fonction retournera 1 si la cellule `(i,j)` est une cellule vide, et 0 sinon. La fonction vérifiera d'abord à l'aide de la fonction `assert` que `(i,j)` correspond à une cellule de la grille.

```c
/**
 * Fonction retournant 1 si la cellule (i,j) de la grille g est vide,
 * et 0 sinon.
 * @param g : grille
```

```
 * @param i : indice de ligne
 * @param j : indice de colonne
 */
int est_cellule_vide(grille * g, int i, int j);
```

Question 11 : Définir la fonction `afficher_grille`. Cette fonction devra afficher à l'écran la grille passée en paramètre.

```
/**
 * Fonction affichant la grille sur le terminal.
 * @param g : pointeur sur la grille que l'on souhaite afficher
 */
void afficher_grille(grille * g);
```

Il existe plusieurs façons d'afficher la grille du Takuzu. On peut distinguer 3 manières :

— Affichage facile : on affiche les valeurs de la grille sous forme d'un tableau à deux dimensions. On affiche le caractère `'I'` devant la valeur si la cellule est initiale. Un exemple est donné par la figure 2.

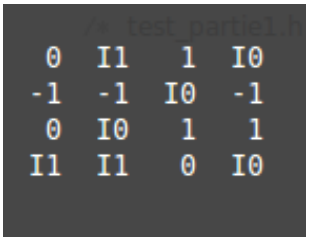

FIGURE 2 – Affichage facile de la grille solution de la figure 1 où les valeurs non initiales de la deuxième ligne ont été supprimées.

— Affichage moyen : Les valeurs -1 (cellules vides) ne sont pas affichées et chaque cellule prend plusieurs lignes et plusieurs colonnes. Les valeurs des cellules initiales sont doublées. Un exemple est donné par la figure 3.

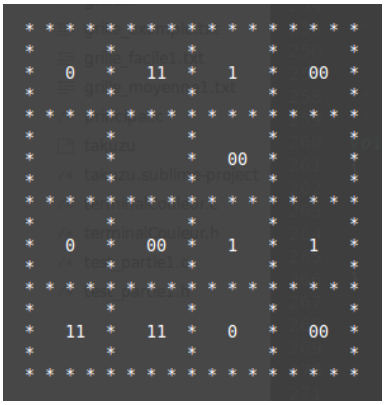

FIGURE 3 – Affichage moyen de la grille solution de la figure 1 où les valeurs non initiales de la deuxième ligne ont été supprimées.

— Affichage difficile : on réalise un affichage en couleurs. Chaque cellule prend plusieurs lignes et colonnes. Les cellules sont différenciées grâce à la couleur de fond. De plus, les valeurs initiales sont affichées d'une couleur différente des autres valeurs. On affiche également les

numéros de ligne et de colonne (les lignes et colonnes 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.

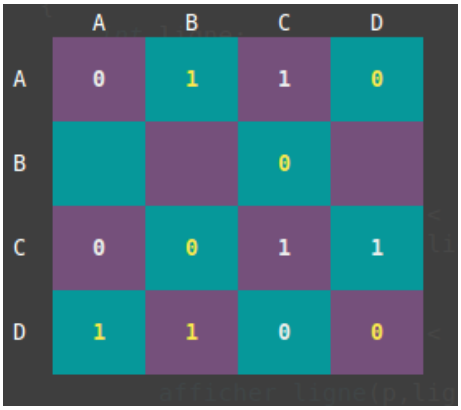

FIGURE 4 – Affichage difficile de la grille solution de la figure 1 où les valeurs non initiales de la deuxième ligne ont été supprimées.

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 de lire une instance dans un fichier et de déterminer si une partie est gagnée.

Question 12 : Définir la fonction `rendre_cellule_initiale` prenant en paramètre un pointeur de grille et deux entiers `i` et `j`. Cette fonction rend la cellule `(i,j)` initiale, c'est-à-dire qu'elle met à 1 l'attribut `initial` de cette cellule. La fonction vérifiera d'abord à l'aide de la fonction `assert` que `(i,j)` correspond à une cellule de la grille et que celle-ci n'est pas une cellule initiale.

```c
/**
 * Fonction modifiant une cellule pour la rendre initiale.
 * @param g : pointeur sur la grille
 * @param i : indice de ligne
 * @param j : indice de colonne
 */
void rendre_cellule_initiale(grille * g, int i, int j);
```

Question 13 : Définir la fonction `initialiser_grille` prenant en paramètre un nom de fichier (chaîne de caractères). La fonction initialise une grille avec les informations contenues dans le fichier et retourne l'adresse de la grille créée.

```c
/**
 * Fonction retournant une grille avec les informations contenues
 * dans le fichier passé en paramètre.
```

```
 * @param nom_fichier : fichier contenant l'instance du problème
 * @return            : pointeur sur la grille créée
 */
grille * initialiser_grille(char nom_fichier[]);
```

Remarque : La structure du fichier de données doit respecter la syntaxe suivante. La première ligne doit contenir le nombre de lignes/colonnes de la grille. La deuxième ligne doit contenir le nombre de valeurs initiales. Chaque ligne suivante contient une valeur initiale. Elle est donnée par le numéro de ligne, le numéro de colonnes et la valeur. À titre d'exemple, le fichier de données correspondant à la grille de départ de la figure 1 est :

```
4
7
0 1 1
0 3 0
1 2 0
2 1 0
3 0 1
3 1 1
3 3 0
```

Question 14 : Définir la fonction `est_grille_pleine` prenant en paramètre un pointeur sur une grille. La fonction retourne 1 si la grille est entièrement remplie (aucune cellule vide), et 0 sinon.

```
/**
 * Fonction testant si la grille est entièrement remplie.
 * @param g : grille à tester
 * @return  : 1 si la grille est pleine, 0 sinon
 */
int est_grille_pleine(grille * g);
```

Question 15 : Définir la fonction `pas_zero_un_consecutifs` prenant en paramètre un pointeur sur une grille. La fonction retourne 1 s'il n'existe pas 3 nombres identiques (0 ou 1) consécutifs sur une ligne ou sur une colonne, et 0 sinon.

```
/**
 * Fonction vérifiant qu'il n'existe pas 3 zéro ou 3 un consécutifs dans
 * la grille (ligne ou colonne).
 * @param g : grille à tester
 * @return  : 1 si c'est le cas, 0 sinon
 */
int pas_zero_un_consecutifs(grille * g);
```

Question 16 : Définir la fonction `meme_nombre_zero_un` prenant en paramètre un pointeur sur une grille. La fonction retourne 1 si chaque ligne et chaque colonne contient autant de 0 que de 1, et 0 sinon.

```
/**
 * Fonction testant si le nombre de zéros est égal au nombre de uns dans
 * chaque ligne/colonne.
 * @param g : grille à tester
 * @return  : 1 si c'est le cas, 0 sinon
 */
int meme_nombre_zero_un(grille * g);
```

Question 17 : Définir la fonction `lignes_colonnes_distinctes` prenant en paramètre un pointeur sur une grille. La fonction retourne 1 s'il n'existe pas deux lignes identiques ni deux colonnes identiques, et 0 sinon.

```
/**
 * Fonction testant qu'il n'existe pas deux lignes identiques ou deux
 * colonnes identiques.
 * @param  g : grille à tester
 * @return   : 1 si c'est le cas, 0 sinon
 */
int lignes_colonnes_distinctes(grille * g);
```

Question 18 : Définir la fonction `est_partie_gagnee` prenant en paramètre un pointeur sur une grille. La fonction retourne 1 si la partie est gagnée et 0 sinon.

```
/**
 * Fonction déterminant si la partie est gagnée.
 * @param  g : grille à tester
 * @return   : 1 si la partie est gagnée, 0 sinon
 */
int est_partie_gagnee(grille * g);
```

2.3 Partie 3

Les fonctions à implémenter dans cette partie permettent de jouer au Takuzu.

Les mouvements saisis par l'utilisateur doivent respecter une syntaxe précise. L'utilisateur doit saisir une chaîne de deux ou trois caractères. Le premier caractère indique le numéro de ligne. Ce numéro est codé sous forme d'une lettre majuscule (`'A'` pour la première ligne, `'B'` pour la seconde, etc). Le deuxième caractère indique le numéro de colonne. Ce numéro est également codé sous forme d'une lettre majuscule (`'A'` pour la première colonne, `'B'` pour la seconde, etc). Le troisième caractère indique la valeur que l'utilisateur souhaite mettre dans la cellule. Les valeurs possibles sont `0` et `1`. Pour supprimer une valeur dans une cellule, l'utilisateur rentre les caractères correspondant aux ligne et colonne de la cellule et n'indique aucune valeur.
Exemple : L'utilisateur saisit `"AB1"` pour mettre un 1 dans la deuxième cellule de la première ligne. Il saisit `"DD0"` pour mettre un 0 dans la quatrième cellule de la quatrième ligne. Il saisit `"AA"` pour supprimer la valeur de la première cellule de la première ligne.

Question 19 : Définir la fonction `est_mouvement_valide` prenant en paramètre un pointeur sur une grille, une chaîne de caractères et trois entiers passés par adresse. La fonction retourne 1 si le mouvement indiqué par la chaîne de caractères est valide, et 0 sinon. Par ailleurs, si le mouvement est valide, les entiers passés par adresse doivent contenir les numéros de ligne et de colonne et la valeur correspondant au mouvement passé en paramètre.

Le mouvement est valide si la cellule existe et n'est pas une cellule initiale. Dans le cas de la suppression d'une valeur, la cellule doit contenir une valeur. On ne regarde pas si les autres conditions (autant de un que de zéro dans chaque ligne/colonne, etc) sont vérifiées.
Exemple : Supposons que l'on ait l'appel `est_mouvement_valide(g,ch,&l,&c,&v);` où `g` est un pointeur sur la grille de départ de la figure 1 et `ch` est une chaîne valant `"DC1"`. Cet appel doit retourner 1. De plus, après appel, `l`, `c` et `v` doivent respectivement contenir les valeurs 3, 2 et 1.

```
/**
 * Fonction déterminant si un mouvement est valide. Si c'est le cas,
 * elle met à jour les indices de ligne et colonne et la valeur en
```

```
 * fonction de la saisie.
 * @param  g       : pointeur sur la grille
 * @param  mouv    : chaîne de caractères contenant le mouvement
 * @param  ligne   : indice de ligne à modifier en fonction de mouv
 * @param  colonne : indice de colonne à modifier en fonction de mouv
 * @param  val     : valeur à modifier en fonction de mouv
 * @return         : 1 si le mouvement est valide, et 0 sinon
 */
int est_mouvement_valide(grille * g, char mouv[], int * ligne, int * colonne,
                         int * val);
```

Question 20 : Définir la fonction `tour_de_jeu` prenant en paramètre un pointeur sur une grille. Cette fonction demandera à l'utilisateur de saisir un mouvement. Cette saisie sera répétée jusqu'à ce que l'utilisateur saisisse un mouvement valide. La fonction modifiera la grille en conséquence.

```
/**
 * Fonction effectuant un tour de jeu :
 *    - saisie jusqu'à ce que l'utilisateur saisisse un mouvement valide,
 *    - modification de la grille en fonction de la saisie.
 * @param  g : pointeur sur la grille
 */
void tour_de_jeu(grille * g);
```

Question 21 : Définir la fonction `jouer` prenant en paramètre une chaîne de caractères `ch`. Cette fonction permet de jouer au Takuzu. Elle commence par créer une grille en fonction des informations contenues dans le fichier dont le nom est donné en paramètre (chaîne `ch`). Elle permet à l'utilisateur de saisir des mouvements jusqu'à ce que la partie soit gagnée.

```
/**
 * Fonction permettant de jouer au Takuzu.
 * @param ch : Chaîne de caractères contenant le nom du fichier
 *             correspondant à l'instance du Takuzu
 */
void jouer(char ch[]);
```

Question 22 : On suppose que les fichiers d'instances du Takuzu sont stockés dans un répertoire `Grilles`. Celui-ci contient 3 sous-répertoires `G4`, `G6` et `G8` contenant respectivement 5 instances du Takuzu de taille 4×4, 6×6 et 8×8. Le nom de chaque instance est `grilleX.txt` où `X` vaut 1, 2, 3, 4 ou 5. Vous pouvez télécharger les instances sur la page Web :

`https://lipn.univ-paris13.fr/~lacroix/enseignement.php`.

Définir la fonction `choisir_grille` permettant de choisir alétoirement une grille de Takuzu. La fonction demande à l'utilisateur de saisir un nombre parmi 4, 6 ou 8 (la saisie est répétée jusqu'à ce que l'utilisateur rentre l'une de ces trois valeurs). Elle choisit aléatoirement un nombre entre 1 et 5 inclus puis modifie la chaîne passée en paramètre pour qu'elle contienne le nom de l'instance choisie (chemin relatif).
Exemple : Si l'utilisateur saisit la valeur 6 et que le nombre aléatoire généré est 2, après appel de la fonction, la chaîne passée en paramètre doit contenir la valeur `"Grilles/G6/grille2.txt"`.
Remarque : Il est possible d'utiliser la fonction `strcpy` définie dans la libriairie `string.h`.

```
/**
 * Fonction permettant de choisir aléatoirement une grille dont la taille
 * est saisie par l'utilisateur.
```

```
4    * @param s : chaîne de caractères contenant le nom de la grille
5    *            choisie aléatoirement
6    */
7   void choisir_grille(char s[]);
```

Question 23 : Écrire le programme principal permettant de jouer au Takuzu. La taille de la grille est demandée à l'utilisateur et la grille est choisie aléatoirement.

Question 24 : Séparer le code en plusieurs fichiers sources et créer un Makefile permettant de compiler le 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/tests 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.

Pour faciliter le début du code, vous trouverez sur le site le fichier `squelette_partie1.c`. Ce fichier est un squelette de code source pour la partie 1 qu'il faut compléter. Il contient les inclusions de fichiers nécessaires, la définition des fonctions pour afficher à l'écran en couleur, les en-têtes des fonctions à définir ainsi que les définitions des fonctions de test de la partie 1. Vous pouvez télécharger ce fichier et écrire le code de la partie 1 dedans.