TP noté

1 – Exercice I : Sortir du labyrinthe

L'objectif de cet exercice est de trouver comment sortir d'un labyrinthe. La structure du labyrinthe est la suivante :

```
typedef struct labyrinthe {
    struct labyrinthe *portes[4];
    int distance;
} *Labyrinthe;
```

Chaque structure `Labyrinthe` contient un tableau de portes qui vont permettre de se déplacer dans toutes les directions. La porte 0 va vers le haut, la porte 1 va vers la droite, la porte 2 va vers le bas, et la porte 3 va vers la gauche (sens horaire).

La variable `distance` contient la distance qui sépare cette case de la sortie. Un exemple de labyrinthe est illustré sur la figure 1, où le personnage et la sortie sont représentés.

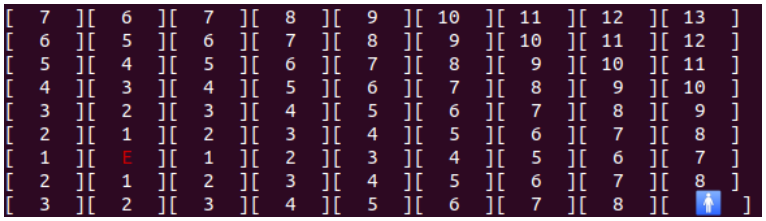

FIGURE 1 – Exemple de labyrinthe

Le labyrinthe est déjà construit. Son code, **qui ne doit pas être modifié**, se situe dans les fichiers `labyrinthe.c` et `labyrinthe.h` du dossier `labyrinthe`.

Vous entrerez vos réponses dans le fichier `solution.c` qui se trouve lui aussi dans le dossier `labyrinthe`. Toutes les définitions des fonctions qui vous seront demandées dans les questions ci-dessous sont déjà dans ce fichier avec la documentation qui leur est associée. Vous trouverez aussi un `Makefile` déjà fait qui vous permettra de compiler vos fichiers, et un fichier `test.c` pour tester votre solution.

Questions :

1. Codez la fonction `meilleure_porte` qui retournera la direction (i.e. l'identifiant de la porte) dans laquelle le joueur doit se déplacer pour atteindre l'objectif.

   ```
   int meilleure_porte(Labyrinthe l);
   ```

2. Codez la fonction `sortir` qui retournera une liste de **toutes les directions à suivre pour sortir du labyrinthe**. Dans l'exemple de la figure 1, les deux listes valides seraient $0-0-3-3-3-3-3-3-3-3$ ou $3-3-3-3-3-3-3-0-0$. Vous trouverez l'implémentation d'une liste doublement chaînée circulaire dans le dossier `list`.

   ```
   Liste sortir(Labyrinthe l);
   ```

3. La fonction `sortir` retourne **un** des plus courts chemins vers la sortie. Quels changements seraient nécessaires pour que la fonction `sortir` retourne **tous** les plus courts chemins ?
4. (Bonus) Codez la fonction `toutes_sorties` qui retournera tous les plus courts chemins vers la sortie du labyrinthe.

2 – Exercice II : Classement sportif

L'objectif de cet exercice est de créer une structure de donnée qui simulera un classement sportif, où l'ordre des équipes dans la structure devra être déterminé par le nombre de points que possède chaque équipe.

La structure de données d'un `Classement` est la suivante :

```
typedef struct classement{
    int id_equipe;
    int points;
    struct classement *suivant;
    struct classement *precedent;
} *Classement;
```

Les pointeurs `suivant` et `précédent` servent à relier une équipe aux autres équipes du classement. La variable `id_equipe` servira à identifier une équipe unique, et la variable `points` donnera le nombre de points possédé par cette équipe.

Des exemples de classement sont présentés sur les tables 2 à 4.

TABLE 1 – Différentes opérations sur les classements

TABLE 2 – Classement original

ID équipe	Points
0	20
1	10

TABLE 3 – ajouter_equipe(c, 2, 15)

ID équipe	Points
0	20
2	15
1	10

TABLE 4 – ajouter_points(c, 1, 7)

ID équipe	Points
0	20
1	17
2	15

Vous entrerez vos réponses dans le fichier `classement.c` qui se trouve dans le dossier `classement`. Toutes les définitions des fonctions qui vous seront demandées dans les questions ci-dessous sont déjà dans ce fichier avec la documentation qui leur est associée. Vous trouverez aussi la définition et la spécification des fonctions qui vont être utilisées dans le fichier `test.c`. Vous trouverez aussi un `Makefile` déjà fait qui vous permettra de compiler vos fichiers.

Questions :

1. Quel type de structure de donnée et quel type de chaînage vous semblent être les plus adéquats pour modéliser un classement ? Comment modifieriez-vous une structure de donnée vue en cours pour satisfaire les contraintes de cet exercice ? Justifiez votre réponse.
2. Codez la fonction `ajouter_equipe`. Cette fonction ajoutera **nouvelle équipe** dans le classement **à la bonne position** (le score le plus haut en première position, le deuxième plus haut en deuxième position, etc...). Voir exemple sur la table 3.

   ```
   Classement ajouter_equipe(Classement c, int id_equipe, int points);
   ```

3. Codez la fonction `ajouter_points`. Cette fonction ajoutera des points à une **équipe existante**. L'ordre du classement devra rester cohérent, c'est-à-dire que l'équipe à laquelle on a ajouté les points devra être mise à la bonne position dans le classement. Voir transition entre la table 3 et la table 4. **Attention : il est aussi possible que l'ajout de points fasse gagner plusieurs places à une équipe**.

   ```
   Classement ajouter_points(Classement c, int id_equipe, int points);
   ```