Colle n°6

Tous vos fichiers sont à **renommer** comme d'habitude (en y ajoutant votre nom en majuscule) et à **déposer sur Moodle** dans la section `Informatique - Devoirs`, tout en bas dans 2ème session.

Vous devrez terminer l'exercice qui vous a été attribué et me remettre l'ensemble des fichiers corrigés sur Moodle au même endroit avant dimanche soir.

Pensez à toutes les bonnes pratiques de programmation : réfléchir sur papier, tester dès que cela est possible, méthode des petits pas, programmation défensive...

Si vous avez terminé en moins d'1h, venez me demander un exercice supplémentaire.

Exercice 1 (Listes doublement chaînées (AIT-EL-AMIRI, GRELY)).

Vous n'êtes pas autorisé à réutiliser le travail effectué en TP mardi, vous devez être capable de reconstruire de zéro. Toutefois, si vous êtes bloqué, appelez moi pour pouvoir continuer à travailler.
Une liste doublement chaînée (LDC) est une liste implémentée sous la forme de maillons chaînés qui peuvent se parcourir dans les deux sens : chaque maillon est donc relié au maillon suivant mais aussi au maillon précédent. **On considérera ici uniquement des listes de nombres réels.**

Toutes les fonctions de cet exercice seront écrites en C dans un fichier `NOM_ldc.c`

1. Proposer des structures de données en C qui implémentent concrètement ce type de listes
2. Implémenter une fonction `ldc_create` permettant de créer un objet de ce nouveau type
3. Implémenter une fonction `ldc_push` permettant d'ajouter un élément **en tête** de votre LDC
4. Implémenter une fonction `ldc_print` qui affiche tous les éléments de votre LDC
5. Tester et déboguer ces premières fonctions, en faisant des manipulations « en dur » dans la fonction principale
6. Implémenter et tester une fonction `ldc_free` qui libère tout l'espace mémoire qui a été alloué pour le stockage de la LDC.
7. Implémenter une fonction `ldc_pop` qui retourne la valeur stockée en tête de liste et supprime l'élément de tête de liste.
8. Tester abondamment cette fonction.
9. Implémenter une fonction `ldc_insert_ith` qui insère une nouvelle valeur en i-ème position (on commence la numérotation des éléments à 0)
10. Implémenter une fonction `ldc_delete_ith` qui supprime le ième élément de la liste
11. Implémenter une fonction `ldc_reverse` qui renverse l'ordre des éléments d'une liste doublement chaînée. La fonction doit travailler **en place**, c'est-à-dire sans allouer de nouveaux maillons.
12. Implémenter une fonction `ldc_tri_insertion` qui trie une liste doublement chaînée par valeurs croissantes avec l'algorithme de tri par insertion.

Exercice 2 (Labyrinthe (ESCUDER, GIRAULT, HOSFORD)).

Le but de cet exercice est de créer un algorithme permettant de résoudre une énigme de type labyrinthe. Pour un labyrinthe 2D donné, avec une entrée et une sortie, l'algorithme doit trouver un chemin permettant d'aller de l'entrée jusqu'à la sortie.

Un labyrinthe 2D peut être vu comme une grille dimensionnelle, chaque case de la grille représentant une case du labyrinthe. Chaque case est repérée par son indice ligne/colonne, en numérotant à partir de 0. Par exemple $(0, 0)$ dans l'exemple ci-dessus pour la case tout en haut à gauche et $(5, 5)$ pour la case tout en bas à droite. On choisit donc d'indexer les cases en partant du coin supérieur gauche et en considérant les cases ligne par ligne (la première ligne est tout en haut, la dernière tout en bas), jusqu'à la dernière case, qui représente le coin inférieur droit. Les indices de ligne i pour une case sont donc numérotés de haut en bas et les indices de colonnes j de gauche à droite.

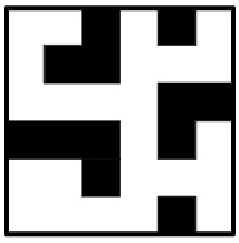
Dans chaque case de la grille 2D, on stocke un 0 s'il s'agit d'un passage, un 1 s'il s'agit d'un mur.

A partir d'une case de la grille, on peut se déplacer sur une case libre voisine horizontalement ou verticalement (pas de déplacement en diagonale).

En fait, pour des raisons d'implémentation, on représentera un labyrinthe en mémoire par un tableau uni-dimensionnel, version linéarisée de la grille 2D. Une case d'indice (i, j) du labyrinthe 2D correspond à la case d'indice `siz_lab` $\times i + j$ dans le tableau linéarisé.

Voici un exemple d'association entre une tableau et un labyrinthe 2D, prenez bien le temps de l'étudier pour bien comprendre les conventions choisies pour représenter un labyrinthe.

```c
int siz_lab = 6; // taille du labyrinthe carre
int test_lab[36] =
  {
   0, 0, 1, 0, 1, 0,
   0, 1, 1, 0, 0, 0,
   0, 0, 0, 0, 1, 1,
   1, 1, 1, 0, 1, 0,
   0, 0, 1, 0, 0, 0,
   0, 0, 0, 0, 1, 0,
  };
```



Le but de l'exercice est d'écrire une fonction qui détermine un chemin de l'entrée à la sortie du labyrinthe par une recherche en profondeur utilisant une pile.

- Nous devrons implémenter une fonction qui donnera, à partir des coordonnées d'une case, la liste des cases voisines libres et non déjà visitées.
- De chaque case, on se déplacera sur une telle case voisine libre et non déjà visitée.
- Dès que l'on visitera une case, on mettra la valeur -1 dans la case associée dans le tableau du labyrinthe pour indiquer que cette case a été visitée. Ainsi, à tout moment, les seules cases non déjà visitées et qui ne sont pas des murs sont des cases dont la valeur est à 0.
- Une pile constituera le fil d'Ariane de notre recherche vers la sortie : on empilera les cases visitées successivement. Dès qu'une case n'aura plus aucune case voisine qui n'a pas déjà été visitée, on la dépilera, pour poursuivre le trajet à partir de la case précédente.
- La pile finale obtenue, si l'on a atteint la case de sortie, correspond au chemin parcouru depuis l'entrée et constitue donc une solution de l'algorithme. On l'affichera à l'écran.

Toutes les fonctions de cet exercice seront écrites en C dans un fichier `NOM_labyrinthe.c`

1. Reprendre le fichier implémenté lors du TP précédent, implémentant une structure de données de pile avec un grand tableau, et copiez là dans votre répertoire de travail, dans un fichier nommé `NOM_labyrinthe.c`
2. Adaptez le `main` pour représenter en dur dans la fonction principal le labyrinthe de la figure ci-dessus, que nous utiliserons comme test.
3. Coder la fonction `idx2couple` qui permet de passer de l'indice unidimensionnel d'une case dans le tableau linéarisé au couple d'indice (i, j). On réfléchira bien à la manière de retourner les deux indices i et j.
4. Coder la fonction `neighbours`, qui liste les cases voisine d'une case donnée, accessibles et non déjà visitées. Le prototype de la fonction est le suivant :

   ```
   int neighbours(int idx, int *lab, int siz_lab, int *neigh)
   ```

   Vous réfléchirez bien en amont à ses arguments d'entrée, et à la manière de retourner ses sorties. *Une case a au maximum 4 voisins.*

   Tester !
5. Écrire en pseudo-code sur papier une algorithme qui prendra en entrée le tableau unidimensionnel du labyrinthe, l'indice de l'entrée et l'indice de la sortie du labyrinthe dans ce tableau, et retournera la pile des indices unidimensionnels des cases successivement visitées constituant un trajet de l'entrée à la sortie.
6. Coder cet algorithme dans une fonction `compute_solution`.

   ```
   bool compute_solution(int idx_start, int idx_end, int *lab, int siz_lab)
   ```

   Cette fonction renvoie la valeur vraie si une solution a été trouvée, et fausse sinon. Elle affichera le chemin solution s'il existe.

   Vous testerez sur le labyrinthe donné en exemple et dont vous avez implémenté le tableau dans la fonction principale, en prenant comme entrée la case $(0, 0)$ (indice linéarisé 0) et en sortie la case $(5, 5)$ (indice linéarisé 35).
7. Améliorations :
   - **a.** Écrire une fonction permettant de générer un labyrinthe carré aléatoire dont la taille est donnée
   - **b.** Écrire une fonction d'affichage permettant d'afficher un labyrinthe dans le terminal