DURÉE: 1h30, UNE FEUILLE A5 RECTO-VERSO MANUSCRITE AUTORISÉE

REMARQUES:
- Les exercices sont indépendants et peuvent être réalisés dans l'ordre voulu.
- Dans l'implémentation d'une méthode, on pourra utiliser n'importe quelle autre méthode définie auparavant même si celle-ci n'a pas été implémentée.
- Dans toutes les implémentations que vous écrivez, pensez à respecter le guide de syntaxe pour la programmation (règles de nommage, présentation des blocs, *etc.*).

EXERCICE 1: SUITES ET RIVIÈRES (9 POINTS)

Une rivière (digitale) est une suite mathématique dont chaque élément $r_n \in \mathbb{N}$ correspond à l'élément précédent r_{n-1} auquel on ajoute la somme des chiffres de r_{n-1}.
Par exemple, avec $r_0 = 42$, on aura $r_1 = r_0 + 4 + 2 = 48$, $r_2 = r_1 + 4 + 8 = 60$, ...

1. Écrire une fonction `int nextRiverElement( int r )` qui calcule l'élément suivant d'une rivière sachant l'élément courant `r`.

Solution: (1 point)
```
int nextRiverElement( int r ) {
  int res = r;
  while( r != 0 ) {
    res += r % 10;
    r /= 10;
  }
  return res;
}
```

2. Écrire une fonction `int* nextRiverElements( int r, int n )` qui calcule les n premiers éléments d'une rivière avec $r_0 =$`r`.

Solution: (2 points) : 1 point pour le malloc / calloc, 1 point pour le reste de la fonction
```
int* nextRiverElements( int r, int n ) {
  int* res = malloc( n * sizeof( int ) );
  for( int i = 0; i < n; i++ ) {
    res[i] = r;
    r = nextRiverElement( r );
  }
  return res;
}
```

3. Écrire une fonction `void printRiverElements( int* r, int n )` qui affiche les n premiers éléments d'une rivière sous la forme $r_0 \to r_1 \to r_2 \to \ldots$

Solution: (1 point)
```
void printRiverElements( int *r, int n ) {
  int i;
  for( i = 0; i < n-1; i++ )
    printf( "%d -> ", r[i] );
  printf( "%d\n", r[i] );
}
```

4. Écrire une fonction `int riverCrossing( int r1, int r2, int n )` qui renvoie le point d'intersection de 2 rivières commençant en $r1$ et $r2$. Si les rivières ne se sont pas croisées après n calculs d'éléments suivants, on supposera qu'il n'existe pas de solution et on renverra -1.
Pour trouver le point d'intersection des 2 rivières, on calculera l'élément suivant de la plus petite des 2 (jusqu'à trouver la même valeur ou dépasser le nombre de calculs autorisés).

Solution: (2 points)

```
int riverCrossing( int r1, int r2, int n ) {
  int t = 0;
  while( t < n && r1 != r2 ) {
    if( r1 < r2 )       r1 = nextRiverElement( r1 );
    else                r2 = nextRiverElement( r2 );
    t++;
  }
  return ( r1 == r2 ) ? r1 : -1;
}
```

5. Quelle est la complexité algorithmique de la fonction `riverCrossing` ?

Solution: (1 point)
$\mathcal{C}_{\text{riverCrossing}} = n \times \mathcal{C}_{\text{nextRiverElement}} = n \times \mathcal{O}(n) = \mathcal{O}(n^2)$

6. Écrire le programme principal qui
   1. inclura les librairies nécessaires,
   2. définira 3 constantes (2 entiers $R1\_0$ et $R2\_0$ correspondant à la valeur initiale de 2 rivières et un nombre d'itérations N)
   3. puis implémentera une fonction `main`. Celle-ci devra
      (a) calculer et afficher les N premiers termes de la rivière 1
      (b) trouver l'intersection des rivières 1 et 2.

Solution: (2 points) : 1 point pour la manipulation correcte du tableau d'entier (*i.e.* appel des fonctions écrites en Q2 et Q3, libération mémoire à la fin).

```
#include <stdio.h>
#include <stdlib.h>

#define R1_0 471
#define R2_0 480
#define N 100

/* Les fonctions vont ici */

int main( ) {
  int *r = nextRiverElements( R1_0, N );
  printRiverElements( r, N );
  free( r );
  printf( "%d\n", riverCrossing( R1_0, R2_0, N ) );
  return 0;
}
```

EXERCICE 2: LABYRINTHE (13 POINTS)

Vous avez récemment fait l'acquisition de plusieurs cartes d'un même labyrinthe pour maximiser vos chances d'y trouver le trésor (et d'en ressortir vivant . . .). Le problème est que chaque carte indique un chemin différent et certaines sont des pièges et ne vous mènent pas au trésor ! Vous décidez donc de faire la seule chose logique dans cette situation : écrire un programme qui lira ces cartes, trouvera le chemin vers le trésor et vous dira quelle carte choisir de sorte à y arriver le plus rapidement possible.
Pour cette exercice, vous disposez de la structure ci-dessous et définie dans une librairie "`maps.h`" qui sera progressivement complétée.

```
typedef struct {
  int dimX, dimY; // Dimensions du labyrinthe
  int xS, yS;     // Entree du labyrinthe
  char** grille;  // Carte du labyrinthe
} Carte;
```

1. **Création, destruction et affichage**
   (a) Écrire une fonction `Carte creerCarte( int dimX, int dimY, int xS, int yS )` qui créé et renvoie une `Carte` (vide). Tous les éléments de la grille seront des `'.'`.

Solution: (2 points) : 1 point pour l'alloc dynamique correcte, 0.5 point pour le remplissage de la grille

```
Carte creerCarte( int dimX, int dimY, int xS, int yS ) {
  int i, j;
  Carte res = {dimX, dimY, xS, yS}; // Creation de la structure
  // Allocation dynamique de la grille
  res.grille = malloc( dimX * sizeof( char* ) );
  for( i = 0; i < dimX; i++ ) {
    res.grille[i] = calloc( dimX, sizeof( char ) );
    for( j = 0; j < dimY; j++ )
      res.grille[i][j] = '.';
  }
  return res;
}
```

   (b) Écrire une fonction `void freeCarte( Carte* c )` qui détruit une `Carte`.

Solution: (1 point) : ne pas pénaliser l'absence de la partie "sécurité"

```
void freeCarte( Carte* c ) {
  int i;
  // Destruction de la grille
  for( i = 0; i < c->dimX; i++ )
    free( c->grille[i] );
  free( c->grille );
  // Securite
  c->dimX = 0;
  c->dimX = 0;
  c->grille = NULL;
}
```

   (c) Écrire une fonction `void afficherCarte( Carte c )` qui affiche la grille d'une `Carte`.

Solution: (1 point) :

```
void afficherCarte( Carte c ) {
  int i, j;
  for( i = 0; i < c.dimX; i++ ) {
    for( j = 0; j < c.dimY; j++ )
      printf( " %c", c.grille[i][j] );
    printf( "\n" );
  }
}
```

2. **Lecture, écriture :** Chaque carte est stockée dans un fichier texte, dont la première ligne contient les dimensions en x et en y de la carte, la deuxième ligne contient l'entrée du labyrinthe (xS et yS) et les n lignes suivantes correspondent à la grille du labyrinthe.
   (a) **Sujet A :** Écrire une fonction `Carte lireCarte( char* fName )` qui lit une `Carte` depuis un fichier texte.

Solution: (2 points) : 1 point pour la bonne gestion des fichiers (ouverture, fermeture et test). Ne pas pénaliser l'absence de lecture du `"n"`.

```
Carte lireCarte( char* fName ) {
  Carte c;
  // Ouverture en lecture d'un fichier texte
  FILE *fTxt = fopen( fName, "r" );
  if( fTxt != NULL ) {
    // Lecture de la taille et de l'entree de la carte
```

```
    int dimX, dimY, xS, yS;
    fscanf( fTxt, "%d %d\n", &dimX, &dimY );
    fscanf( fTxt, "%d %d\n", &xS, &yS );
    // Creation de la carte
    c = creerCarte( dimX, dimY, xS, yS );
    // Lecture de la carte
    int i, j;
    for( i = 0; i < dimX; i++ ) {
      for( j = 0; j < dimY; j++ )
        fscanf( fTxt, "%c", &(c.grille[i][j]) );
      fscanf( fTxt, "\n" );      // Pour lire le \n
    }
    // Fermeture du fichier
    fclose( fTxt );
  }
  return c;
}
```

(b) **Sujet B :** Écrire une fonction `void ecrireCarte( char* fName, Carte c )` qui écrit une `Carte` depuis un fichier texte.

Solution: (2 points) : 1 point pour la bonne gestion des fichiers (ouverture, fermeture et test).

```
void ecrireCarte( char* fName, Carte c ) {
  // Ouverture en ecriture d'un fichier binaire
  FILE *fTxt = fopen( fName, "w" );
  if( fTxt != NULL ) {
    // Ecriture de la taille et de l'entree de la carte
    fprintf( fTxt, "%d %d\n", c.dimX, c.dimY );
    fprintf( fTxt, "%d %d\n", c.xS, c.yS );
    // Ecriture de la carte
    int i, j;
    for( i = 0; i < c.dimX; i++ ) {
      for( j = 0; j < c.dimY; j++ )
        fprintf( fTxt, "%c", c.grille[i][j] );
      fprintf( fTxt, "\n" );
    }
    // Fermeture du fichier
    fclose( fTxt );
  }
}
```

(c) Écrire une fonction `Carte* lireCartes( char* fName, int* nCartes )` qui lit le contenu d'un fichier texte `fName` et renvoie un tableau de `Carte` et le nombre de cartes lues dans le fichier.
Chaque ligne de ce fichier contient le nom d'un fichier contenant une `Carte` à lire. Le nombre de lignes dans le fichier est inconnu (il s'agit du nombre de cartes que vous devez renvoyer) mais ne pourra pas dépasser `NMAX`, une constante définie au début de la librairie `"maps.h"`. On supposera qu'une fonction `Carte lireCarte( char* fName )` existe et elle pourra être utilisée sans donner son implémentation.

Solution: (2 points) : 1 point pour le test correct dans la boucle while

```
Carte* lireCartes( char* fName, int* nCartes ) {
  // Resultats en cas de mauvaise lecture
  Carte* c = NULL;
  *nCartes = 0;
  // Ouverture en lecture d'un fichier texte
  FILE *fTxt = fopen( fName, "r" );
  if( fTxt != NULL ) {
    char fCarte[50];
    c = calloc( NMAX, sizeof( Carte ) );
    // Tant qu'on arrive a lire un nom de fichier et qu'il y a de la place
       dans le tableau
```

```
    while( *nCartes < NMAX && fscanf( fTxt, "%s\n", fCarte ) == 1 ) {
      c[*nCartes] = lireCarte( fCarte );  // On lit une carte
      (*nCartes)++;                       // On incremente le nombre de cartes lues
    }
  }
  return c;
}
```

3. **Recherche du chemin :** Le chemin sur la carte est indiqué par les caractères `'<'` (gauche), `'>'` (droite), `'v'` (bas), `'^'` (haut), le trésor par `'T'`, les murs par `'#'` et les autres cases par `'.'`.
La coordonnée (0, 0) du labyrinthe correspond à la case en bas à gauche , les lignes correspondent à la coordonnée x, les colonnes à y.

   (a) Écrire une fonction `int checkBound( int x, int y, Carte c )` qui vérifie que les coordonnées x et y sont bien à l'intérieur de la carte.

Solution: (0.5 point) :
```
int checkBound( int x, int y, Carte c ) {
  int res = x >= 0 && x < c.dimX;          // x est dans la grille
  res = res && y >= 0 && y < c.dimY;       // y est dans la grille
  return res;
}
```

   (b) Écrire une fonction `int trouverChemin( Carte c )` qui renvoie le nombre de cases à parcourir pour aller de l'entrée du labyrinthe jusqu'au trésor. La fonction renverra -2 si le chemin vous fait sortir de la carte et -1 si le chemin vous amène dans un mur (`'#'`) ou sur une case "vide" (`'.'`).

Solution: (1,5 points) :
```
int trouverChemin( Carte c ) {
  int i = 0;
  int x = c.xS;
  int y = c.yS;
  while( checkBound( x, y, c ) ) {
    char m = c.grille[x][y];
    if( m == '^' )          x -= 1;   // x += 1;  (sujet B)
    else if( m == 'v' )     x += 1;   // x -= 1;  (sujet B)
    else if( m == '<' )     y -= 1;
    else if( m == '>' )     y += 1;
    else if( m == 'T' )     return i;
    else                    return -1;
    i++;
  }
  return -2;
}
```

   (c) Écrire une fonction `int trouverMeilleureCarte( Carte* allMaps, int nCartes )` qui renvoie la meilleure carte, c'est à dire celle reliant l'entrée du labyrinthe au trésor le plus rapidement. On renverra -1 si aucune carte ne permet d'accéder au trésor.
On pourra utiliser la fonction `int trouverChemin( Carte c )` même si la question précédente n'a pas été traitée.

Solution: (1.5 points) : 0.5 point pour le test, 0.5 point pour la boucle.
```
int trouverMeilleureCarte( Carte* allMaps, int nCartes ) {
  int res = -1;
  int minChemin = -1;
  int i;
  for( i = 0; i < nCartes; i++ ) {
    int lChemin = trouverChemin( allMaps[i] );
    if( lChemin >= 0 && ( lChemin < minChemin  || res == -1 ) ) {
```

```
        minChemin = lChemin;
        res = i;
      }
    }
    return res;
  }
```

4. On suppose que toutes les fonctions précédentes ont été implémentées dans la libraire ”maps.h” et on pourra donc s’en servir même si la question correspondante n’a pas été traitée.
   Écrire le programme principal qui
   1. lira l’ensemble des cartes depuis un fichier. Le nom du fichier (`"data/maps.txt"`) sera défini comme une constante au début du programme.
   2. affichera l’ensemble des cartes lues
   3. trouvera la meilleure carte, affichera un message contenant son numéro et écrira son contenu dans un fichier texte. Le nom du fichier (`"results/bestMap.txt"`) sera défini comme une constante au début du programme.
      Si aucune carte ne permet d’accéder au trésor, on affichera un message pour prévenir qu’il s’agît d’un piège !

   On pensera à inclure les bibliothèques nécessaires au fonctionnement du projet.

Solution: (1,5 points) : 0.5 point pour la destruction des cartes à la fin, 1 point pour le reste.

```
#include "maps.h"

#define FNAME_MAPS "data/maps.txt"
#define FNAME_OUT "results/bestMap.txt"

int main( ) {
  int nCartes, i;
  Carte* allMaps = lireCartes( FNAME_MAPS, &nCartes );
  for( i = 0; i < nCartes; i++ )
    afficherCarte( allMaps[i] );

  // Recherche du chemin et affichage de la meilleur carte
  int nCarte = trouverMeilleureCarte( allMaps, nCartes );
  if( nCarte == -1 )
    printf( "It's a trap!\n" );
  else {
    printf( "Vous devriez utiliser la carte %d\n", nCarte );
    ecrireCarte( FNAME_OUT, allMaps[nCarte] );
  }

  // Destruction des cartes
  for( i = 0; i < nCartes; i++ )
    freeCarte( &(allMaps[i]) );
  free( allMaps );

  return 0;
}
```

Cet exercice est inspiré du puzzle ”Dungeons and Maps” de CodinGame.