

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 écrirez, 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`.
2. Écrire une fonction `int* nextRiverElements(int r, int n)` qui calcule les n premiers éléments d'une rivière avec $r_0 = r$.
3. Écrire une fonction `void printRiverElements(int* r, int n)` qui affiche les n premiers éléments d'une rivière sous la forme $r_0 \rightarrow r_1 \rightarrow r_2 \rightarrow \dots$.
4. Écrire une fonction `int riverCrossing(int r1, int r2, int n)` qui renvoie le point d'intersection de 2 rivières commençant en r_1 et r_2 . 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).
5. Quelle est la complexité algorithmique de la fonction `riverCrossing`?
6. Écrire le programme principal qui
 1. inclura les bibliothèques 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.

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 crée et renvoie une Carte (vide). Tous les éléments de la grille seront des ' '.
- (b) Écrire une fonction `void freeCarte(Carte* c)` qui détruit une Carte.
- (c) Écrire une fonction `void afficherCarte(Carte c)` qui affiche la grille d'une Carte.

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) Écrire une fonction `Carte lireCarte(char* fName)` qui lit une Carte depuis un fichier texte.
 - (b) 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 pourra utiliser la fonction `Carte lireCarte(char* fName)` même si la question précédente n'a pas été traitée.
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 coordonnées (0, 0) du labyrinthe correspond à la case en haut à gauche, les lignes correspondent à la coordonnées 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.
 - (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" ('`.`').
 - (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.
4. On suppose que toutes les fonctions précédentes ont été implémentées dans la librairie `"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'agit d'un piège !
- On pensera à inclure les bibliothèques nécessaires au fonctionnement du projet. On supposera qu'une fonction `void ecrireCarte(char* fName, Carte c)` existe et pourra être utilisée sans donner son implémentation. Cette fonction écrit une `Carte` dans un fichier texte dont le nom est donné en entrée.

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