TP 3 : Fonctions simples en C

Notions : Buts : Fonctions et tableaux

Ce TP se réalise en 2 parties sur 2 séances encadrées : une première partie avec des fonctions élémentaires, une deuxième partie avec des fonctions manipulant des tableaux. Les extensions facultatives sont aussi possibles.

1 Première Partie : séance 1

1.1 Fonctions simples

Ecrire les fonctions suivantes et les tester une à une avec un programme principal que vous écrirez aussi :

Maximum de 2 entiers : Ecrire une fonction qui retourne le maximum de 2 entiers. Le prototype de cette fonction sera :

```
int max2Entiers(int x, int y)
```

Maximum de 3 entiers : Ecrire une fonction qui retourne le maximum de 3 entiers. Cette fonction doit utiliser la fonction `int max2Entiers` que vous venez d'écrire. Le prototype de cette fonction sera :

```
int max3Entiers(int x, int y, int z)
```

Maximum d'un tableau Ecrire une fonction qui retourne le maximum des éléments d'un tableau de `n` réels. Le prototype de cette fonction sera :

```
double maximum Tableau(double tab[], int n) ;
```

Calcul du pgcd : Ecrire une fonction qui calcule le pgcd de 2 entiers et un programme principal qui utilise cette fonction.

L'algorithme est :

---

repeat
  $r \leftarrow$ reste de la division entiere de a par b (operateur % en C)
  Changer le rôle de a, b, et r
  $a \leftarrow b$
  $b \leftarrow r$
until $r = 0$

Le prototype de cette fonction sera : `int pgcd(int a, int b) ;`

Rq : pour cette fonction, il est préférable que $a > b$ pour des raisons d'efficacité

1.2 Fonctions et tableaux

Nombres parfaits : Un nombre parfait est un nombre qui est égal à la somme de ses diviseurs propres ;

— Ecrire une fonction qui identifie tous les diviseurs de l'entier `n` (y compris n lui-meme) et les range dans le tableau `diviseurs` dans l'ordre croissant. La fonction retourne le nombre de diviseurs trouvés. On supposera que $n > 2$. L'algorithme naïf consiste simplement à tester si chaque entier compris entre 1 et n est un diviseur de n. L'entête de la fonction est :

`int diviseurs(int n, int diviseurs[])`

Note : `a` et `b` étant des expressions entières, le quotient et le reste de la division entière de `a` par `b` s'obtiennent respectivement par les expressions `a / b` et `a % b`.

— En utilisant la fonction précédente, écrire une fonction qui retourne la somme des diviseurs propres d'un nombre (tous les diviseurs sauf le nombre lui meme). L'entête de la fonction est : `int somme(int diviseurs[], int nbdiviseurs)`

— En utilisant les fonctions précédentes, écrire une fonction qui retourne 1 si le nombre est parfait, 0 sinon. L'entête de la fonction est :

`int perfection(int n)`

— En utilisant les fonctions précédentes, écrire un programme qui demande un nombre entier au clavier et indique si ce nombre est parfait ou non.

2 Deuxième Partie : fonctions et passage des tableaux en paramètre

L'objectif est ici de réaliser la gestion du jeu Master Mind. : vous devez deviner quelles sont les N (5) couleurs, choisies au hasard parmi M (7) couleurs possibles par votre programme. Une même couleur peut être tirée plusieurs fois. Le joueur fait une proposition de configuration de couleurs pour deviner la configuration initiale. Le programme doit répondre

— combien de couleurs sont à leur place,

— combien de couleurs sont effectivement présentes mais mal placées.

A partir de cette réponse, le joueur propose une nouvelle configuration à laquelle le programme répond et ainsi de suite jusqu'à ce que la bonne réponse soit trouvée par le joueur. On ne demande pas de réaliser un programme qui devine la solution, mais simplement un programme qui indique quelles sont les couleurs correctement devinées ou placées.

Les couleurs sont représentées par un entier (1..M) et une configuration est un tableau de N entiers. N et M sont des constantes définies initialement à l'aide des instructions du préprocesseur par :

```
#define N 5
#define M 7
```

2.1 Algorithme

Une configuration est donc un tableau d'entiers. Vous utiliserez deux tableaux : un pour la configuration initiale à deviner, un pour la configuration du joueur. Ce dernier changera donc à chaque tour de jeu. Vous décomposerez votre problème en plusieurs étapes plus simples. Les différentes étapes à réaliser (ou l'algorithme) sont les suivantes :

Initialisation : créer les variables utiles
Créer 2 tableaux : machine et joueur
Créer 3 entiers : nbcoups, bp et mp :
Initialiser le tableau à deviner
repeat
  Lire la configuration proposée par le joueur
  Incrémenter nbcoups
  Calculer le nombre de couleurs devinées et bien placées et mettre ce nombre dans bp
  Calculer le nombre de couleurs devinées et mal placées et mettre ce nombre dans mp
  Afficher les valeurs de bp et mp
until le nombre de bien placées est N
Afficher le nombre de coups

Pour ce premier programme utilisant plusieurs fonctions, nous n'utiliserons qu'**UN** seul fichier. Toutes les fonctions, y compris le programme principal, seront écrites dans cet unique fichier `master.c`. Nous vous donnons plusieurs fonctions que vous n'avez pas à écrire et que vous pouvez récupérer sur le site `tdinfo.phelma.grenoble-inp.fr` et télécharger dans les fichier `master.c` et `master.h` :

— Une fonction qui remplit un tableau de n cases (couleurs) tirées au hasard parmi les m couleurs possibles pour créer la configuration initiale tab (celle que l'ordinateur choisi). L'entête de la fonction est `void init(int tab[], int n, int m);`.

— Une fonction qui affiche un tableau tab de n cases (m couleurs) à l'écran. L'entête de la fonction est `void affiche(int tab[], int n, int m);`.

— Une fonction qui retourne combien de couleurs sont présentes mais mal placées dans une configuration proposée par le joueur par rapport à la configuration initiale vous est donnée sur le site. Elle retourne le nombre de couleurs mal placées. L'entête de la fonction est `int malplace(int joueur[], int machine[], int n, int m)`.

2.2 Travail à réaliser

Il vous reste à écrire les fonctions suivantes dans le fichier `master.c` et leurs prototypes dans le fichier `master.h` :

1. Ecrire une fonction qui lit une configuration au clavier (celle du joueur). Pour simplifier, cette fonction doit lire des entiers au clavier. Elle vérifie aussi que les nombres entrés sont bien dans les limites autorisées (entre 0 et m-1). C'est la fonction d'affichage qui fera la conversion entre un entier et les couleurs des pions. L'entête de la fonction est `void lire(int tab[], int n, int m);`. Ne pas oublier d'ajouter l'entête de cette fonction dans le fichier `master.h`.
2. Ecrire une fonction qui indique combien de couleurs sont bien placées dans une configuration joueur proposée par le joueur par rapport à la configuration initiale machine. Elle retourne le nombre de couleurs bien placées. L'entête de la fonction sera `int bienplace(int joueur[], int machine[], int n);`. Ne pas oublier d'ajouter l'entête de cette fonction dans le fichier `master.h`.
3. Ecrire un programme qui réalise les différentes opérations pour jouer au master mind : tirage initial, lecture de la configuration du joueur au clavier, réponse du programme, boucle et gestion de la fin de la partie. Ne pas oublier d'Inclure le fichier `master.h`.
4. Vous pouvez créer votre exécutable avec la commande unix : `clang master.c -o master` puis l'exécuter avec la commande unix : `./master`

Attention : utilisez une approche incrémentale pour développer ce programme. Vérifiez chaque fonction une à une. Ecrivez une première fonction lire par exemple et vérifier son fonctionnement, puis passer à la fonction bien placée, vérifier, etc..

2.3 Facultatif : version graphique

Ne vous lancez dans cette partie que si vous avez terminé la partie précédente

Vous pouvez utiliser la bibliothèque graphique SDL pour réaliser une version graphique couleur de votre master mind à la manière de la figure 1.

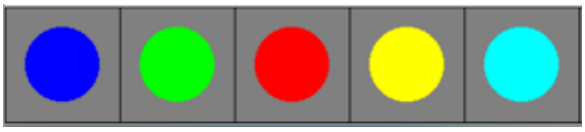


FIGURE 1 – Affichage graphique

4 étapes sont nécessaires pour afficher une configuration :

1. Créer une variable de type `SDL_PHWindow*`
2. Initialisation du système graphique avec la fonction `SDL_Init`
3. Création d'une fenêtre vide par la fonction `SDL_PH_CreateWindow`
4. Affichage d'une configuration `SDL_PH_DrawMasterMind`

Vous retrouvez ces 3 étapes dans le programme contenu dans le fichier `grmastermind.c`. Il réalise les actions suivantes

1. Il crée une fenêtre vide ;
2. Il remplit une configuration
3. Il affiche dans cette fenêtre la configuration.
4. Il attend que l'utilisateur tape sur une touche,
5. Il réalise 5 fois les 3 actions suivantes
   (a) Il remplit la configuration au hasard
   (b) Il l'affiche à nouveau.
   (c) Il attend que l'utilisateur tape sur une touche

Pour compiler, il faut copier le fichier `Makefile` (voir le site des tp) dans son répertoire et utiliser la commande unix `make grmastermind` pour créer un exécutable. On lance ensuite l'exécution par la commande `./grmastermind`. Vous pouvez aussi gérer la souris (voir le site tdinfo, onglet bibliothèque graphique, gestion des évènements et de la souris, fichier `demo3.c`).

Inspirez vous du programme ci dessous pour réaliser une version semi graphique très simple : affichage graphique qui remplace l'affichage texte, mais entrée des configurations au clavier. Le fichier `Makefile` proposé est prévu pour que vous fassiez une version graphique dans un seul fichier `mastergraphique.c` dans lequel vous aurez recopié toutes les fonctions exceptée la fonction `main` du fichier précédent `master.c` [1].

Fichier `grmastermind.c`

```
#include <stdio.h>
#include <SDL2/SDL_phelma.h>

/*      Nombre de point du master mind */
#define N 7
/*      Nombre de couleurs du master mind */
#define M 7
/*      Taille d'un rayon d'un point. La taille d'une case est alors 3R+1 */
#define R 30


int main(int a, char** b) {
  int i,j;
  int t[N] ; /* Tableau representant le master mind */
  SDL_PHWindow* f1=NULL;

    /* initialisation du systeme video,audio de la SDL */
  if ( SDL_Init(SDL_INIT_VIDEO | SDL_INIT_AUDIO| SDL_INIT_EVENTS) !=0) {
        fprintf(stderr,"initialisation de la SDL impossible : %s\n", SDL_GetError());
        exit(EXIT_FAILURE);
  }
        /* Creation de la fenetre d'affichage graphique */
  f1=SDL_PH_CreateWindow((N)*(3*R+1)+1,3*R+3);
  if ( f1==NULL) {printf("Creation de fenetre impossible \n"); exit(EXIT_FAILURE); }
        /* Fenetre vide  */
  SDL_PH_ClearWindow(f1);
        /* On dessine une grille */
  SDL_PH_DrawGrid(f1,1,N,3*R+1,3*R+1,0,0);

        /* Initialisation du tableau */
  for (i=0; i<N; i++) t[i]=i;

        /* Affichage du master mind et attente  */
  SDL_PH_DrawMasterMind(f1,t,N,R);
  printf("Tapez une touche pour continuer \n"); getchar();

        /* 5 autres masterMind affiches successivement  mais avec couleurs au hazard*/
  for (j=0; j<5; j++) {
    for (i=0; i<N; i++) t[i]=rand()%M;
```

1. il est bien sur préférable de travailler avec plusieurs fichiers, mais cela sera pour une autre fois

```
    SDL_PH_DrawMasterMind(f1,t,N,R);
    printf("Tapez une touche pour continuer \n"); getchar();
  }
        /* On libere la memoire utilisee*/
  if (f1) SDL_PH_DestroyWindow(f1);
  SDL_Quit();
  exit(EXIT_SUCCESS);
}
```

Pour information, voici le code de la fonction `SDL_PH_DrawMasterMind`, qui dessine un cercle dans chaque case de la grille. Il n'est pas à recopier, car il existe dans la bibliothèque de fonctions que nous fournissons.

```
void SDL_PH_DrawMasterMind(SDL_PHWindow* f1,int tab[], int nb, int taille) {  int i, coul;
  for (i=0; i<nb; i++) {
    switch(tab[i])  {
      case 0: coul=SDL_PH_RED; break; /* PH_GetPixelRed */
      case 1: coul=SDL_PH_GREEN; break; /* PH_GetPixelGreen */
      case 2: coul=SDL_PH_BLUE; break; /* PH_GetPixelBlue */
      case 3: coul=SDL_PH_YELLOW; break;
      case 4: coul=SDL_PH_MAGENTA; break;
      case 5: coul=SDL_PH_CYAN; break;
      default: coul=SDL_PH_BLACK; break;
    }
      /* un pion est represente par un cercle */
    filledCircleColor (f1->rendu,i*(3*taille+1)+3*taille/2,3*taille/2,taille,coul) ;
  }
    /* On fait l'affichage a l'ecran */
  SDL_RenderPresent(f1->rendu);
}
```

3 Troisième partie : pour ceux qui vont vraiment vite

Pour ceux qui veulent aller plus loin, 2 autres sujets au choix plus ou moins graphiques dans lesquels les tableaux et pointeurs interviennent :

La corde vibrante : un exemple de simulation numérique graphique

Puissance 4 : un exemple de jeu

3.1 Corde vibrante

Cet exercice est un exemple de simulation numérique simple et simplifié de phénomènes physiques. Le code complet de la simulation, y compris les aspects graphiques, prend moins de 50 lignes. Un exemple de la sortie est présent sur le site des tp.

La simulation des equations différentielles décrivant le phénomène se fait par différences finies.

3.2 Les équations régissant la corde vibrante et les approximations de la simulation numérique

L'équation de l'onde transversale se propageant le long d'une corde en vibration s'écrit :

$$\frac{\partial^2 f}{\partial x^2} = \frac{1}{v^2} * \frac{\partial^2 f}{\partial t^2}$$

où f représente l'onde, $x$la distance le long de la corde, t le temps, v représente la vitesse de propagation de l'onde qui dépend du milieu ambiant et de la nature du matériau.

La méthode de résolution utilisée ici consiste à calculer la solution point par point, ces points constituant une solution discrète dans le domaine choisi. C'est une solution numérique, moins précise que les solutions analytiques que vous verrez en cours sur les équations différentielles, mais dont le principe se révèle indispensable lorsqu'il n'existe pas de solution analytique. Nous utilisons ici un schéma numérique aux différences finies, qui exploite le développement en série de Taylor de la fonction f.

Si Δt et Δx désignent les pas de discrétisation respectivement dans le temps et dans l'espace :

$$\begin{aligned} f(x+\Delta x,t) &= f(x,t) + \Delta x * \frac{\partial f(x,t)}{\partial x} + \frac{\Delta x^2}{2} * \frac{\partial^2 f(x,t)}{\partial x^2} + o(\Delta x^2) \\ f(x-\Delta x,t) &= f(x,t) - \Delta x * \frac{\partial f(x,t)}{\partial x} + \frac{\Delta x^2}{2} * \frac{\partial^2 f(x,t)}{\partial x^2} + o(\Delta x^2) \end{aligned}$$

d'où

$$\frac{\partial^2 f(x,t)}{\partial x^2} = \frac{1}{\Delta x^2} * (f(x+\Delta x,t) - 2 * f(x,t) + f(x-\Delta x,t))$$

En procédant de même dans le temps et en négligeant les termes en o(t^2), on obtient

$$\frac{\partial^2 f(x,t)}{\partial t^2} = \frac{1}{\Delta t^2} * (f(x,t+\Delta t) - 2 * f(x,t) + f(x,t-\Delta t))$$

En posant r =(v.Δt/Δx) l'équation initiale devient :

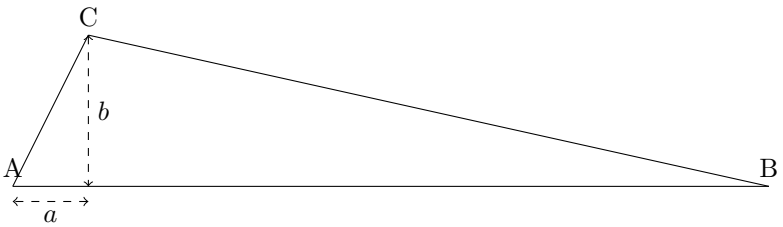
$$f(x,t+\Delta t) = r^2 * (f(x+\Delta x,t) + f(x-\Delta x,t)) \; + \; 2 * (1-r^2) * f(x,t) - f(x,t-\Delta t)$$

On retrouve donc ici une suite de fonctions : on peut calculer pour tout x la fonction $f(x,t+\delta t)$ i.e. la courbe à l'instant $t+\delta t$ en connaissant la fonction en $f(x,t)$, $f(x+\delta x,t)$, $f(x-\delta x,t)$ i.e. la courbe à l'instant t et la fonction en $f(x,t-\delta t)$ i.e. la courbe à l'instant $t-\delta t$.

Cette relation de récurrence est similaire à celle des suites vues au TP précédent avec comme conditions iniitiales la connaissance de la courbe à $t=0$ et $t=-\delta t$ et les points fixes en $x=0$ et $x=1$.

On peut montrer que, pour des raisons de stabilité, la contrainte Δx>=v*Δt doit être vérifiée. La meilleure approximation de la solution est obtenue pour Δx=v*Δt ;

Les conditions initiales valables pour t<=0 (donc pour les 2 premières cordes) sont définies sur le schéma suivant :



La corde ACB est fixée à ses deux extrémités A et B. A l'instant initial, cette corde est disposée suivant le profil indiqué sur la figure, et se trouve libérée de la contrainte appliquée en C. A partir des conditions initiales à l'instant t=0 pour lesquelles on connaît la position de la corde en tout point, on peut calculer à l'aide de l'équation précédente la valeur de f(x, Δt) en tout point de la corde, puis la corde à l'instant 2Δt et ainsi de suite pour tout t.

3.3 Travail à réaliser :

Le calcul des cordes dans le temps est donc un calcul dont le principe est identique à celui des suites et 3 tableaux seulement seront nécessaires, représentant respectivement les cordes aux instants t, $t-1$,$t-2$.

On définira et utilisera au moins les fonctions suivantes, écrites dans le fichier `corde.c` :

— une fonction initialisation, dont le rôle est de donner des valeurs initiales à tous les points de la courbe de longueur L=1m, échantillonnée sur `n` points et donc `dx=1/n` . Son prototype sera :
`void init(double tab[], int n, double a, double b) ;`

— une fonction calcul, dont le rôle est de calculer une nouvelle corde `cordet` de longueur 1m, échantillonnée sur `n` points à partir des deux cordes précédentes `cordet1` et `cordet2`. Elle retourne 1 si le calcul est possible (condition de stabilité respectée) ou 0 sinon. Au début de la simulation, les 2 cordes `cordet1` et `cordet2` seront 2 cordes identiques (on lâche la corde avec une vitesse nulle). Son prototype sera
`int calcul(double cordet[],double cordet1[] , double cordet2[], int n, double dt) ;`

— Une fonction qui recopie le tableau `cordet1` dans le tableau `cordet2`. Cette fonction sera utile pour préparer l'itération suivante de la suite de cordes [2]. Son prototype sera
`int copieTableau(double cordet1[],double cordet2[], int n);`

— une fonction d'affichage : son prototype sera `void affiche(double tab[], int n)` ;

— un programme principal qui effectuera le calcul et l'affichage des courbes entre t=0 et t=1 par pas de temps de Δt.

Valeurs numériques par défaut : v=300m/s ; L=1m ; a=0.15m ; b=0,0lm ; $\Delta t = 10\mu s$

3.4 Version graphique

Pour réaliser un affichage graphique de la suite des courbes (figure 2), vous pouvez vous inspirer du programme suivant qui affiche la courbe $log(1+x)$.

FIGURE 2 – Affichage graphique d'une courbe représentée par un tableau de réels

Pour compiler, il faut copier le fichier `Makefile` (voir le site des tp) et utiliser la commande unix `make exemple` pour créer un exécutable et executer avec la commande `./exemple`.

Le Makefile est aussi prévu pour que vous écriviez le code dans un fichier `corde.c`. La compilation se fera avec la commande `make corde` pour créer un exécutable et l'execution avec la commande `./corde`.

Fichier `exemple.c`

```
#include <stdio.h>
#include <stdlib.h>
#include <SDL2/SDL_phelma.h>
#include <math.h>

  /* Nombres de points dans le tableau */
#define N 300

int main() {
  int i, dimx=N*2, dimy=200 ;
  double y[N];
  SDL_PHWindow*  f1;

  /* Courbe log(i+1) dans le tableau x*/
  for (i=0; i<N; i++) y[i]=log(i+1);

    /* initialisation du systeme video,audio de la SDL */
  if ( SDL_Init(SDL_INIT_VIDEO | SDL_INIT_AUDIO| SDL_INIT_EVENTS) !=0) {
        fprintf(stderr,"initialisation de la SDL impossible : %s\n", SDL_GetError());
        exit(EXIT_FAILURE);
  }
  /* Creation d'une fenetre graphique */
  f1=SDL_PH_CreateWindow(dimx,dimy);
    /* Fenetre cree ? */
  if (f1==NULL) { puts("impossible d'ouvrir une fenetre graphique"); exit(EXIT_FAILURE); }
  else {
        /* Effacer la fenetre */
      SDL_PH_ClearWindow(f1);
```

2. il existe cependant une solution plus rapide qu'une copie des contenus des tableaux en utilisant des pointeurs

```
        /* affichage de la courbe y de N points, abscisses entre 0 et N, ordonnees entre 0 et log(1+N),
    couleur Rouge */
      SDL_PH_DrawTabDoubleScaleY(f1, y,N,0,log(1+N),SDL_PH_RED);
        /* On attend que l'utilisateur tape sur une touche pour terminer le programme */
      printf("Taper une touche pour continuer\n");
      getchar();
        /* Liberer la memoire */
      SDL_PH_DestroyWindow(f1);
  }
  SDL_Quit();
  return EXIT_SUCCESS;
}
```

3.5 Puissance4, pour ceux qui préfèrent les jeux

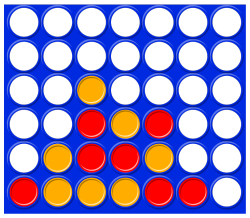


FIGURE 3 – Une partie de Puissance 4 en cours

Puissance 4 est un jeu de société ressemblant au jeu du morpion dont le but est de faire aligner 4 jetons de sa propre couleur sur une grille (figure 3) positionnée verticalement et dont les jetons subissent donc la gravité.

Le but de votre programme est de permettre au joueur d'entrer la colonne dans laquelle il veut placer son pion, puis choisir le coup de la machine et continuer jusqu'à ce qu'il y ait un gagnant ou qu'il n'y ait plus de place disponible.

Dans cet exercice, par simplicité, seule une succession de 4 jetons de même type en colonne ou en ligne permet de gagner (le placement en diagonal sera être mis en œuvre à la fin).

Nous représenterons la grille du jeu par un tableau unidimensionnel Lig*Col (lignes*colonnes). Lige tableau sera de type char et contiendra soit le caractère point `'.'` (case vide), soit le caractère `'x'` (jeton du joueur) soit le caractère `'o'` (jeton de la machine).

L'affichage se fera en mode texte selon le modèle présenté figure 4

.	.	.	.	.	.	.
.	.	o	.	.	.	.
.	.	x	o	x	.	.
.	o	x	x	o	.	.
x	o	o	o	x	x	.

FIGURE 4 – Affichage en mode texte

3.6 Représenter un tableau multi-dimensionnel

10	11	12	ligne d'indice 3
7	8	9	ligne d'indice 2
4	5	6	ligne d'indice 1
1	2	3	ligne d'indice 0

FIGURE 5 – Une matrice

On peut représenter un tableau multidimensionnel dans un tableau unidimensionnel en l'aplatissant. Prenons l'exemple d'une matrice de Lig=4 lignes et Col=3 colonnes dont les cases sont numérotées sur le tableau de la figure 5

Cette matrice peut être représentée par le tableau linéaire suivant :
`char tab[] = {1,2,3,4,5,6,7,8,9,10,11,12};`

Tous les *Col* éléments nous changeons en fait de ligne. Ainsi, l'élément (i,j) de la matrice peut être accédé par la notation `tab[i*Col+j]` ;

Attention : pour l'affichage, il faut veiller à effectuer l'affichage par ordre décroissant des lignes, i.e. d'abord la ligne 3, puis la ligne 2, etc.

3.7 Travail à réaliser

Dans le fichier `puissance.c` (à télécharger sur le site des td info) analysez le main() puis écrivez les fonctions suivantes :

— `char element(int l, int c, char grille[], int colonnes)` : fonction qui retourne l'élément de la colonne d'indice c et de la ligne d'indice l de la grille

— `void init(char grille[], int lignes, int colonnes)` qui initialise la grille (c.-à-d. met des '.' partout).

— `void affiche(char grille[], int lignes, int colonnes)` qui affiche la grille à l'écran (attention la ligne zéro doit être en bas de l'écran).

— `int coup_possible(int numero_col, char grille[], int lignes, int colonnes)` fonction qui vérifie si un coup est possible (hors cases ou case occupée) et retourne le numéro de la ligne si possible ou -1 si impossible.

— `void lire_joueur(char grille[], int lignes, int colonnes)` fonction qui acquiert le coup du joueur et vérifie si le coup est possible.

Les fonctions `calculer_coup_machine` et `est_gagnant` rudimentaires sont fournies :

— `char est_gagnant(char joueur,char grille[], int lignes, int colonnes)` fonction qui vérifie si le coup est gagnant (4 jetons alignés en lignes ou colonnes)

— `void calculer_coup_machine(char grille[], int lignes, int colonnes)~` : fonction qui joue pour la machine de façon très simple.

3.8 Pour aller plus loin :

— améliorez la fonction est_gagnant pour prendre en compte les configurations en diagonale

— modifiez le programme pour pouvoir joueur à deux joueurs humains

— améliorez le choix du coup machine