Synthèse : Puissance 4
Corrigé

Le jeu « Puissance 4 » est proposé par MB Jeux. Sur la boîte, on peut lire les indications suivantes.

> Puissance 4 est un jeu de stratégie verticale passionnant et plein d'astuce. C'est un jeu facile à apprendre et amusant à jouer. Ses règles sont simples. Chaque joueur essaie de faire une rangée de quatre pions dans le cadre – horizontalement, verticalement ou diagonalement – tout en essayant d'empêcher son adversaire de faire de même. Croyez-moi ce n'est pas aussi facile que cela en a l'air ! La position verticale du jeu demande au joueur beaucoup de concentration et de réflexion.
>
> **Montage :**
>
> 1. Suivez les instructions illustrées pour l'assemblage des parties en plastiques (non reproduites ici).
> 2. Placez le jeu entre les joueurs.
> 3. Chaque joueur choisit une couleur de pions.
>
> **But du jeu :**
> Être le premier joueur à placer quatre de ses pions sur une ligne horizontale, verticale ou diagonale continue.
>
> **Règles du jeu :**
>
> 1. Choisissez le premier joueur. Le joueur qui commence la première partie sera le deuxième joueur au cours de la deuxième partie.
> 2. À tour de rôle, chaque joueur fait tomber un de ses pions dans une des fentes au sommet de la grille.
> 3. Le jeu continue jusqu'à ce qu'un des joueurs ait une ligne continue de quatre pions de sa couleur. L'alignement peut être vertical, horizontal ou en diagonale.
> 4. Pour vider la grille, poussez la barre de retenue qui se trouve au vase de celle-ci et les pions tomberont. Vous pouvez maintenant commencer la partie suivante.

Conseil : Lire attentivement les questions suivantes et ébaucher une solution au brouillon avant de commencer la rédaction.

Exercice 1 : Préparation du jeu
Le jeu « puissance 4 » se joue sur une grille verticale de six lignes et sept colonnes avec 21 pions rouges et 21 pions jaunes.

Pour faciliter les calculs ultérieurs, nous allons ajouter une bordure à cette grille. Ses dimensions sont donc de huit lignes et neuf colonnes. Bien entendu, ces cases supplémentaires ne peuvent pas contenir de pions (qu'ils soient jaunes ou rouges). Elles sont donc toujours vides.

Définir les types nécessaires pour représenter la grille de jeu.

C

```c
/* Définir le type pour une case de la grille */
enum Case { ROUGE, JAUNE, VIDE };
typedef enum Case Case;

/* La grille de jeu avec une bordure supplémentaire */
typedef Case Grille[NB_LIGNES+2][NB_COLONNES+2];
```

Exercice 2 : Vider la grille

Pour commencer chaque partie, il faut commencer par vider la grille des jetons qu'elle contient. Écrire un sous-programme qui réalise cette opération.

Solution : Ce sous-programme a un seul paramètre, la grille, en **out**. Il s'agit donc d'une initialisation. Nous en faisons donc une procédure.

```
Procédure vider(grille: out Grille) Est
    -- vider la grille.
```

Ce sous-programme consiste à mettre toutes les cases de la grille à VIDE, y compris les cases de la bordure. On le fait donc naturellement avec deux boucles imbriquées, l'une qui balaie les lignes, l'autre les colonnes.

C Notons qu'en C, nous n'avons pas fait un passage par adresse car la grille est un tableau. La grille est donc déjà passée par adresse (adresse du premier élément du tableau).

```c
/* Vider la grille.  Ceci correspond à actionner la barre se situant
 * sous la grille du jeu réel.
 *
 * Assure :
 *      -- la grille est vide
 */
void vider(Grille grille)
{
    int i, j;   /* parcourir les cases de la grille */

    for (i = 0; i <= NB_LIGNES+1; i++) {
        for (j = 0; j <= NB_COLONNES+1; j++) {
            grille[i][j] = VIDE;
        }
    }
}
```

Exercice 3 : Afficher la grille

Écrire un sous-programme qui affiche le contenu de la grille. Par convention, les jetons de couleur rouge seront représentés par un astérisque (*) et les jetons jaunes par la lettre o minuscule.

Pour faciliter, la localisation d'une case de la grille, les numéros de lignes et de colonnes seront affichés comme sur la figure 1.

```
   1 2 3 4 5 6 7
6        *        6
5        o        5
4      o o   *    4
3      * o   o o  3
2    * o *   o *  2
1  * o * * o o *  1
   1 2 3 4 5 6 7
```

FIG. 1 – Affichage de la grille

Solution : Ce sous-programme prend un seul paramètre, la grille. C'est un paramètre en entrée. De plus ce sous-programme fait de l'affichage. Nous en faisons donc une procédure.

```
1 Procédure afficher(grille: in Grille) Est
2     -- Afficher la grille.
```

L'implémentation de ce sous-programme n'est pas très facile à structurer, en particulier si on veut essayer de limiter les redondances qui rendent plus difficiles les évolutions ultérieures.

Comme on ne sait pas se positionner sur l'écran, il faut donc afficher ligne par ligne. On se rend compte rapidement qu'il nous faudra donc faire deux boucles imbriquées : l'une pour les lignes, l'autre pour les colonnes. On peut réaliser une boucle qui s'appuie sur l'affichage à l'écran et qui inclut donc la bordure correspondant aux numéros. Il suffit alors de tester les valeurs des indices de boucle des lignes et colonnes pour savoir si :

- on n'affiche rien (les quatre coins) ;
- on affiche un numéro de ligne ou de colonne (les bordures) ;
- on afficher le contenu d'une case (les autres cases).

Une autre solution, celle retenue ici, consiste à séparer la première et la dernière ligne qui consiste à afficher les numéros de colonne. Notons que pour afficher les lignes, on commence par celle de numéro NB_LIGNES car les indices de notre tableau correspondent aux numéros de ligne du puissance 4 (on aurait pu prendre la convention inverse).

```
1  R1 : Raffinage De « afficher »
2     | Afficher les numéros de colonnes
3     | Pour i ← NB_LIGNES Décrémenter JusquÀ i = 1_LIGNES Faire
4     |    | Afficher la ligne i
5     | FinPour
6     | Afficher les numéros de colonnes
7
8  R2 : Raffinage De « Afficher la ligne i »
9     | Afficher le numéro de ligne
10    | Pour j ← 1 JusquÀ j = NB_COLONNES Faire
11    |    | afficher_case(grille[i][j])
12    | FinPour
13    | Afficher le numéro de ligne
```

Comme l'affichage des numéros de colonne est à faire en début et fin, nous en faisons un

sous-programme pour factoriser le code.

C

```c
/* Afficher les numéros de colonnes */
void afficher_numeros_colonnes()
{
    int j;      /* parcourir les colonnes de la grille */
    printf("%3s", "");

    for (j = 1; j <= NB_COLONNES; j++) {
        printf("%1d ", j);
    }
    printf("\n");
}

/* Afficher la grille et son contenu */
void afficher(Grille grille)
{
    int i, j;   /* parcourir les cases de la grille */

    printf("\n");
    afficher_numeros_colonnes();

    for (i = NB_LIGNES; i >= 1; i--) {

        /* afficher le numéro de ligne à gauche */
        printf("%2d", i);

        /* afficher la ligne i */
        for (j = 1; j <= NB_COLONNES; j++) {
            printf(" %c", symbole_case(grille[i][j]));
        }

        /* afficher le numéro de ligne à droite */
        printf("%2d", i);
        printf("\n");
    }

    afficher_numeros_colonnes();
    printf("\n");
}
```

Exercice 4 : Décider si un coup est possible

Écrire un sous-programme qui indique s'il est possible de jouer dans une colonne donnée.

Exemple : Sur l'exemple de la figure 1, il est possible de jouer les colonnes 1, 2, 3, 5, 6 ou 7. Il n'est pas possible de jouer la colonne 4.

Solution : Ce sous-programme prend en paramètre :

- la grille (**in**) ;
- le numéro de colonne envisagé (**in**) ;
- l'information de si le coup est possible (**out**).

Un seul paramètre en **out**, d'autres paramètres en **in**. On en fait donc une fonction.

```
Fonction est_coup_possible(grille: in Grille; colonne: in Entier): Booléen Est
    -- Est-ce qu'il est possible de jouer dans la colonne donnée
    -- de la grille ?
```

Pour savoir s'il est possible de jouer, il suffit de regarder si la colonne est pleine ou non, c'est-à-dire s'il sa dernière case, celle du haut, est vide ou non. Si elle est vide, on peut jouer dans la colonne. Si elle n'est pas vide, on ne peut pas y jouer. Notons que si elle est vide, ça ne veut pas dire que le pion restera dans cette case. Il tombera...

On en déduit donc naturellement le code.

C

```c
/* est-ce que la colonne peut être jouée sur cette grille ? */
bool est_coup_possible(Grille grille, int colonne)
{
   return grille[NB_LIGNES][colonne] == VIDE;
}
```

Exercice 5 : Lâcher un jeton

Lorsqu'un joueur joue, il lâche un jeton de sa couleur au dessus d'une colonne de la grille. En raison de la force gravitationnelle, le jeton tombe dans la grille jusqu'à ce qu'il soit arrêté par un autre jeton ou le fond de la grille.

Écrire un sous-programme qui réalise cette opération : il calcule le nouvel état de la grille lorsqu'un jeton est lâché au dessus d'une colonne donnée.

Exemple : Partant de la figure 1, si les rouges jouent la colonne 5, le nouvel état de la grille est celui donné par la figure 2.

```
   1 2 3 4 5 6 7
6        *        6
5        o        5
4      o o   *    4
3      * o   o o  3
2    * o * * o *  2
1  * o * * o o *  1
   1 2 3 4 5 6 7
```

FIG. 2 – Les rouges ont joué la colonne 5

Solution : Le sous-programme prend en paramètre :

- la grille (**in out**) ;
- la colonne donnée (**in**) ;
- le jeton, ou plutôt sa couleur (**in**).

Comme il y a un paramètre en **in out**, nous en faisons une procédure.

```
Procédure lâcher(grille: in out Grille;
                 colonne: in Entier;
```

```
3                 couleur: in Case) Est
4      -- La couleur joue un pion dans la colonne.
5      --
6      -- Nécessite
7      --  est_coup_possible(grille, colonne)
```

Le principe de l'algorithme est alors de regarder jusqu'où va tomber le jeton, soit il n'y a encore aucun pion et il est donc en fond de grille, soit il est dans la dernière case qui est vide. En fait, le plus simple est de partir du fond de la colonne et de chercher la première case vide en remontant. Comme on sait qu'il est possible de jouer dans la colonne (précondition), on n'a pas besoin de tester le débordement de la colonne.

```
R1 : Raffinage De « lâcher »
  | Déterminer la ligne de chute
  | Mettre le pion dans cette case

R2 : Raffinage De « Déterminer la ligne de chute »
  | ligne ← 1
  | TantQue grille[ligne, colonne] <> VIDE Faire
  |   | ligne ← ligne + 1
  | FinTQ
```

Remarque : On décide de faire un sous-programme, en l'occurrence une fonction, de « Déterminer la ligne de chute » car nous aurons par la suite l'utilité de trouver cette ligne de chute. Il n'est pas évident ici, du premier coup, de voir qu'il peut être utile d'en faire un sous-programme.

C

```c
/* ligne sur laquelle tombe un jeton lâché à la colonne spécifiée
 *
 * Nécessite
 *      est_coup_possible(grille, colonne)      -- coup possible
 */
int ligne_de_chute(Grille grille, int colonne)
{
    int l;      /* parcourir les lignes de la colonne considérée */

    assert(est_coup_possible(grille, colonne));

    l = 1;
    while (grille[l][colonne] != VIDE) {
        l++;
    }

    return l;
}


/* Lacher sur la grille à la colonne précisée un jeton de la couleur
 * donnée.
 *
 * Nécessite
 *      est_coup_possible(grille, colonne)      -- coup possible
```

```
 *
 * Assure
 *      -- un pion de plus sur la colonne
 *      -- le nouveau pion est de la couleur donnée
 */
void lacher(Grille grille, int colonne, Case couleur)
{
        assert(est_coup_possible(grille, colonne));

        grille[ligne_de_chute(grille, colonne)][colonne] = couleur;
}
```

Exercice 6 : Compter les jetons alignés

Écrire un sous-programme qui indique quelle est la longueur de l'alignement le plus long qui inclut un jeton donné (repéré par son numéro de ligne et son numéro de colonne). Les alignements doivent être cherchés dans toutes les directions (horizontale, verticale ou en diagonale).

Ce sous-programme sera ensuite utilisé pour déterminer la fin d'une partie (alignement ≥ 4) ou aider l'ordinateur à choisir la colonne où jouer (exercice 8).

Sur l'exemple de la figure 2, la case (1,1) correspond à un alignement de trois jetons en diagonale ; la case (3,4) correspond à un alignement de deux jetons horizontalement ; la case (6,2) correspond à trois jetons alignés verticalement ou en diagonale...

Indication : Quel est l'intérêt d'avoir défini une bordure supplémentaire autour de la grille ?

Solution : Ce sous-programme prend en paramètre une grille (**in**), une ligne (**in**) et une colonne (**in**) et détermine le plus long alignement (out). Un paramètre en sortie. Tous les autres en entrée. Nous pouvons donc en faire une fonction.

```
Fonction nb_pions_alignés(grille: in Grille;
                ligne: in Entier;
                colonne: in Entier): Entier
    -- Plus grand nombre de jetons de la même couleur alignés
    -- horizontalement, verticalement ou en diagonale et incluant
    -- le jeton de la position (ligne, colonne).
    --
    -- Nécessite
    --  grille[ligne, colonne] <> VIDE
```

L'algorithme est un peu compliqué, ou tout au moins fastidieux. Il faut regarder dans les quatre directions (horizontalement, verticalement et en diagonale) pour compter l'alignement le plus grand. Pour chaque direction, il faut compter la case (ligne, colonne) et les deux sens de la direction.

Supposons que l'on sache calculer un nombre de pions alignés selon une direction. On peut alors en déduire le code de « nb_pions_alignés »

```
R1 : Raffinage De « nb_pions_alignés »
  | Résultat ← max(nb_pions_dir(grille, ligne, colonne, 1, 1),
  |                max(nb_pions_dir(grille, ligne, colonne, 1, 0),
  |                    max(nb_pions_dir(grille, ligne, colonne, 1, -1),
  |                        nb_pions_dir(grille, ligne, colonne, 0, 1))))
```

Il ne reste plus qu'à écrire nb_pions_dir. C'est une fonction.

```
Fonction nb_pions_dir(grille: in Grille; ligne, colonne: in Entier;
                      dir_x, dir_y: in Entier) Est
    -- Compter le nombre de pions dans la direction (dir_x, dir_y)
    -- et incluant le pion en (ligne, colonne).
    --
    -- Nécessite
    --  grille[ligne, colonne] <> VIDE
    --  (dir_x <> 0) Ou (dir_y <> 0)    -- direction valide
    --
    -- Assure
    --  Résultat >= 1           -- au moins le jeton de (ligne, colonne)
```

Pour son implémentation, il suffit de compter le jeton de (ligne, colonne) puis compter les jetons de même couleur dans la direction (dir_x, dir_y) et ceux dans la direction opposée (-dir_x, -dir_y).

```
R1 : Raffinage De « nb_pions_dir »
  | couleur ← grille[ligne, colonne]
  | Résultat ← 1
  | Comptabiliser les jetons dans la direction (dir_x, dir_y)
  | Comptabiliser les jetons dans la direction (-dir_x, -dir_y)
```

C

```
/* Plus grand nombre de jetons de la même couleur alignés suivant la
 * direction (dir_x, dir_y) et incluant le jeton de la case
 * (ligne,colonne).
 *
 * Nécessite
 *      Grille[ligne][colonne] != VIDE  -- un jeton en (ligne, colonne)
 *      dir_x != 0 || dir_y != 0        -- une vraie direction
 *
 * Assure
 *      Résultat >= 1
 */
int nb_pions_dir(Grille grille, int ligne, int colonne, int dir_x, int dir_y)
{
    int l;      /* ligne dans la direction +/- dir_y */
    int c;      /* colonne dans la direction +/- dir_x */
    Case couleur;       /* la couleur de l'alignement */
    int resultat;

    assert(grille[ligne][colonne] != VIDE);
    assert(dir_x != 0 || dir_y != 0);

    couleur = grille[ligne][colonne];
    resultat = 1;               /* le jeton en (ligne, colonne) */


    /* Comptabiliser les jetons dans la direction (dir_x, dir_y) */
    l = ligne + dir_y;
```

```
28      c = colonne + dir_x;
29      while (grille[l][c] == couleur) {
30          resultat++;
31          l = l + dir_y;
32          c = c + dir_x;
33      }
34
35      /* Comptabiliser les jetons dans la direction (-dir_x, -dir_y) */
36      l = ligne - dir_y;
37      c = colonne - dir_x;
38      while (grille[l][c] == couleur) {
39          resultat++;
40          l = l - dir_y;
41          c = c - dir_x;
42      }
43
44      assert(resultat >= 1);
45
46      return resultat;
47  }
48
49  /* Le plus grand de deux entiers */
50  int max(int a, int b) {
51      if (a > b) {
52          return a;
53      }
54      else {
55          return b;
56      }
57  }
58
59
60  /* Plus grand nombre de jetons de la même couleur alignés
61   * horizontalement, verticalement ou en diagonale et incluant le jeton
62   * de la position (ligne, colonne)
63   *
64   * Assure
65   *      Résultat >= 0
66   */
67  int nb_pions_alignes(Grille grille, int ligne, int colonne)
68  {
69      int resultat;
70
71      if (grille[ligne][colonne] == VIDE) {
72          resultat = 0;
73      }
74      else {
75          /* Déterminer le plus grand alignement */
76          resultat =
77              max(nb_pions_dir(grille, ligne, colonne, 1, 1),
78                  max(nb_pions_dir(grille, ligne, colonne, 1, 0),
```

```
79                      max(nb_pions_dir(grille, ligne, colonne, 1, -1),
80                          nb_pions_dir(grille, ligne, colonne, 0, 1))));
81
82      }
83
84      assert(resultat >= 0);
85
86      return resultat;
87  }
```

Exercice 7 : Conseiller une colonne où jouer

Écrire un sous-programme qui étant donné une grille propose une colonne où jouer. Le numéro de la colonne sera choisi aléatoirement.

Solution : Ce sous-programme est une fonction car il a un seul paramètre en sortie (la colonne conseillée) et un paramètre en entrée (la grille de jeu), éventuellement deux si l'on considère aussi la couleur de celui que l'on doit conseiller (couleur qui n'est pas exploité dans cette version très naïve du conseil).

```
1  Fonction colonne_conseillée_aléatoire(grille: in Grille; couleur: in Case) Est
2      -- colonne conseillée pour jouer
```

Le principe est de tirer une colonne au hasard jusqu'à ce qu'elle correspondent à une colonne où l'on peut jouer.

C

```
1  /* colonne conseillée pour jouer.
2   *
3   * Principe : La colonne conseillée est choisie au hasard entre les
4   * colonnes possibles.
5   *
6   * Nécessite
7   *      -- la partie n'est pas finie
8   *
9   * Assure
10  *      est_coup_possible(grille, Résultat);
11  */
12 int colonne_conseillee_aleatoire(Grille grille, Case couleur)
13 {
14      int resultat;
15      do {
16          resultat = rand() % NB_COLONNES + 1;
17      } while (!est_coup_possible(grille, resultat));
18
19      return resultat;
20 }
```

Exercice 8 : Améliorer le conseil

Pour améliorer le conseil, on ajoute une contrainte supplémentaire : conseiller la colonne qui permet l'alignement le plus long.

Remarque : Cette contrainte implique que si un coup gagnant existe, la colonne conseillée conduira à la victoire.
Exemple : Partant de la figure 2, les jaunes joueront en colonne 5. Ceci provoque un alignement de 4 pions et donc la victoire.
Solution : Les paramètres sont les mêmes que pour le conseil précédent sauf que la couleur du joueur est ici nécessaire.

```
1  Fonction colonne_conseillée(grille: in Grille; couleur: in Case) Est
2      -- colonne conseillée pour jouer
```

Le principe du conseil est un peu plus compliqué car il faut proposer le coup correspondant à l'alignement le plus grand. On a déjà écrit un sous-programme pour calculer un alignement mais il suppose que l'on ait déjà jouer dans la case. Il nous faut donc jouer dans la case. Et si l'on joue, il faut également pouvoir annuler le coup pour ne pas fausser la partie.

```
1  R1 : Raffinage De « colonne_conseillée »
2    | max ← 0              -- alignement maximal trouvé
3    | Pour c ← 1 JusquÀ c = NB_COLONNES Faire
4    |   | Jouer en c pour couleur
5    |   | Si alignement > max Alors
6    |   |   | max ← alignement
7    |   |   | Résultat ← c
8    |   | FinSi
9    |   | Annuler le coup en c
10   | FinPour
```

C

```
1  /* Annuler le dernier coup jouer sur la colonne spécifiée.
2   *
3   * Nécessite
4   *      grille[1][colonne] != VIDE      -- coup possible
5   *
6   * Assure
7   *      -- un pion de moins sur la colonne
8   */
9  void annuler(Grille grille, int colonne)
10 {
11     assert(grille[1][colonne] != VIDE);
12
13     grille[ligne_dernier_jeton(grille, colonne)][colonne] = VIDE;
14 }
15
16
17 /* Colonne conseillée pour jouer.
18  *
19  * Principe : La colonne conseillée est celle qui permet l'alignement
20  * le plus long.  Ainsi, si un coup est gagnant, la colonne proposée
21  * est gagnante.
22  *
23  * Nécessite
24  *      -- la partie n'est pas finie
```

```
25   *
26   * Assure
27   *      est_coup_possible(grille, Résultat);
28   *      -- si un coup gagnant existe, alors Résultat est gagnant
29   */
30  int colonne_conseillee(Grille grille, Case couleur)
31  {
32      int c;      /* pour chercher une colonne gagnante */
33      int nb_pions;       /* Nombre de pions alignés pour c */
34      int resultat;
35      int max;    /* le nb max de pions alignés (donc de resultat) */
36      int ligne;  /* conserver la ligne de chute du pion lâché */
37
38      /* chercher une colonne permettant le plus long alignement */
39      max = 0;    /* nécessairement un minorant strict */
40      for (c = 1; c <= NB_COLONNES; c++) {
41          if (est_coup_possible(grille, c)) {
42              /* déterminer la ligne de chute */
43              ligne = ligne_de_chute(grille, c);
44
45              /* jouer le coup pour voir */
46              lacher(grille, c, couleur);
47
48              /* évaluer le coup */
49              nb_pions = nb_pions_alignes(grille, ligne, c);
50
51              /* choisir le coup (par rapport au précédent) */
52              if (nb_pions > max          /*{ meilleur coup }*/
53                  || (nb_pions == max     /*{ coup équivalent }*/
54                      && rand() % 2 == 0))        /*{ un peu d'aléatoire }*/
55              {
56                  max = nb_pions;
57                  resultat = c;
58              }
59
60              /* annuler le coup joué */
61              annuler(grille, c);
62          }
63      }
64      return  resultat;
65  }
```