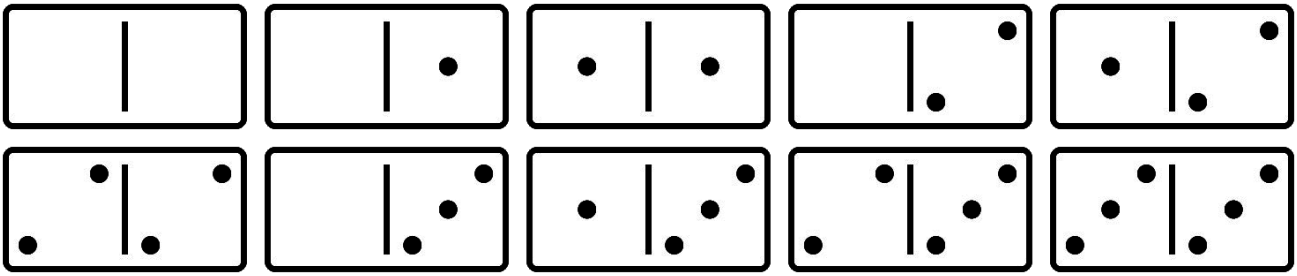
Programmation — Examen

Durée : 2 heures, Calculatrices et Documents Interdits



On souhaite simuler une partie de dominos à 2 joueurs aux règles simplifiées. L'ordinateur joue contre lui-même (pas d'interaction humaine) et de façon aléatoire (pas de stratégie pour selectionner un coup).

Dans tout le devoir, pour implémenter une fonction, vous devez RÉUTILISER les fonctions des questions précédentes lorsque cela est possible, même si vous ne les avez pas implémentées.

Par ailleurs, dans tout l'examen, on utilisera pour les Booléens le type `bool`, et les constantes `true` et `false`, tous trois définis dans `<stdbool.h>`, que l'on supposera inclus comme les autres entêtes standards.

1 Tuile de domino : `struct Tile`

Une tuile de domino (*tile*), notée $[a : b]$, est constituée d'un côté gauche et d'un côté droit, portant respectivement les valeurs entières a et b choisies parmi V valeurs possibles (typiquement comprises entre 0 et 6 pour $V = 7$). La renversée de la tuile $[a : b]$ est la tuile $[b : a]$. Le type structuré `Tile` représente une tuile :

```
typedef struct Tile {
  int left, right;
} Tile;
```

Question 1. Implémenter la fonction `Tile_Make()` qui fabrique la tuile `[left:right]`. En déduire la fonction `Tile_Reversed()` qui fabrique et retourne la renversée d'une tuile.

```
Tile Tile_Make (int left, int right);
Tile Tile_Reversed (Tile tile);
```

Question 2. Implémenter la fonction `Tile_String()` qui retourne la représentation texte d'une tuile sous la forme d'une chaîne statique. L'exemple ci-dessous montre le résultat produit pour une tuile et sa renversée.

```
char * Tile_String (Tile tile);
```

```
int main (void)
{
  Tile tile1= Tile_Make (1, 6);
  Tile tile2= Tile_Reversed (tile1);
  fprintf (stdout, "%s"  , Tile_String (tile1));
  fprintf (stdout, "%s\n", Tile_String (tile2));
  return 0;
}
```

```
[1:6][6:1]
```

Question 3. Implémenter le prédicat `Tile_Contains()` qui teste si une tuile porte la valeur `value` sur au moins l'un de ses côtés.

```
bool Tile_Contains (Tile tile, int value);
```

2 Jeu de tuiles : `struct Set`

S'il y a V valeurs possibles, alors un jeu de tuiles complet comprend $N = V(V+1)/2$ tuiles, à raison d'un exemplaire de toutes les tuiles $[a:b]$ tel que $a \leq b$. (Dans un jeu classique, il y a $N = 28$ tuiles et $V = 7$ valeurs).

Le type structuré `Set` permet de représenter un jeu de tuiles par un tableau de capacité maximale `MAX_TILES`. Si `s` est une variable type `Set`, alors `s.length` est la cardinalité du jeu, et ses tuiles sont rangées dans les cases `s.tiles[k]` pour les index `k` compris dans l'intervalle `0 <= k < s.length`.

```
#define MAX_VALUES   7                                          /* borne max sur V */
#define MAX_TILES    ((MAX_VALUES * (MAX_VALUES + 1)) / 2)      /* borne max sur N */

typedef struct Set {
  Tile tiles [MAX_TILES];
  int length;
} Set;
```

On se donne le prédicat `Set_IsEmpty()` qui teste si un jeu de tuiles est vide, ainsi que le prédicat `Set_IsFull()` qui teste si un jeu de tuiles a atteint la capacité du tableau :

```
bool Set_IsEmpty (Set const * set) { return set->length == 0; }
bool Set_IsFull  (Set const * set) { return set->length == MAX_TILES; }
```

Question 4. Implémenter la fonction `Set_InitEmpty()` qui initialise à vide un jeu de tuiles. Lorsqu'un jeu est non-vide, on considère que ses tuiles sont rangées de la gauche vers la droite selon les index croissants : implémenter la fonction `Set_AddRight()` qui ajoute une tuile à un jeu sur sa droite. Cette dernière fonction vérifie préalablement avec `assert()` que la capacité du tableau n'était pas déja atteinte.

```
void Set_InitEmpty (Set * set);
void Set_AddRight  (Set * set, Tile tile);
```

Question 5. Implémenter la fonction `Set_Print()` qui affiche un jeu de tuiles sur la sortie `file`. Les tuiles sont affichées sur une seule ligne terminée par un retour à la ligne, de la gauche vers la droite selon les index croissants. (L'exemple ci-dessous montre la sortie produite pour un jeu de 3 tuiles).

```
void Set_Print (Set const * set, FILE * file);
```

```
int main (void)
{
  Set set;
  Set_InitEmpty(& set);
  Set_AddRight (& set, Tile_Make (1,6));
  Set_AddRight (& set, Tile_Make (2,4));
  Set_AddRight (& set, Tile_Make (3,0));
  Set_Print (& set, stdout);
  return 0;
}
```

```
[1:6][2:4][3:0]
```

Question 6. Implémenter la fonction `Set_Init()` qui initialise un jeu complet pour $V =$ `nb_values`. La fonction commence par construire un jeu vide, puis ajoute par la droite chacune des tuiles $[a:b]$ vérifiant $a \leq b$. (L'exemple ci-dessous montre le jeu complet pour $V = 4$).

```
void Set_Init (Set * set, int nb_values);
```

```
int main (void)
{
  Set set;
  Set_Init  (& set, 4);
  Set_Print (& set, stdout);
  return 0;
}
```

```
[0:0][0:1][1:1][0:2][1:2][2:2][0:3][1:3][2:3][3:3]
```

3 Pioche et rateliers (stock and racks)

Le ratelier (*rack*) est le jeu de tuiles d'un joueur. Afin de pouvoir lui faire jouer un coup, il faut être capable d'extraire une tuile d'un jeu de tuiles à un index choisi.

Question 7. Implémenter la fonction `Set_RemoveAt()` qui retire la tuile située à un index donné dans un jeu de tuiles. Les tuiles suivantes sont décalées d'une case vers la gauche. La tuile retirée est retournée par la fonction. La fonction vérifie préalablement avec `assert()` que l'index est bien valide.

```
Tile Set_RemoveAt (Set * set, int index);
```

```
int main (void)
{
  Set set; Tile tile;
  Set_Init (& set, 4);
  fprintf (stdout, "before: "); Set_Print (& set, stdout);
  tile= Set_RemoveAt(& set, 1);
  fprintf (stdout, "after : "); Set_Print (& set, stdout);
  fprintf (stdout, "result: %s", Tile_String (tile));
  return 0;
}
```

```
before: [0:0][0:1][1:1][0:2][1:2][2:2][0:3][1:3][2:3][3:3]
after : [0:0][1:1][0:2][1:2][2:2][0:3][1:3][2:3][3:3]
result: [0:1]
```

La pioche (*stock*) est le jeu de tuiles dans laquelle les joueurs prélèvent des tuiles au hasard pour les ajouter à leur rack (lors de la distribution initiale, et lorsqu'ils doivent passer leur tour). On préfère ici ne pas mélanger la pioche, mais retirer une tuile à un index aléatoire dans la pioche lors d'un prélèvement.

Question 8. Implémenter la fonction `Set_RemoveRandom()` qui choisit une tuile au hasard dans un jeu, la retire et la retourne. La fonction vérifie préalablement avec `assert()` que le le jeu n'était pas déjà vide.

```
Tile Set_RemoveRandom (Set * set);
```

4 Chaîne de tuiles (chain)

La chaîne (*chain*) est le jeu de tuiles commun au centre la table qui est construit par tous les joueurs. Dans la chaîne, si une tuile $[a_1 : b_1]$ est suivie immédiatement par une tuile $[a_2 : b_2]$ alors elle doivent être connectées, c'est à dire vérifier l'égalité $b_1 = a_2$. On peut ajouter une nouvelle tuile à une chaîne non-vide par l'une des deux extrémités de la chaîne, en renversant éventuellement la tuile afin qu'elle se connecte correctement à l'extrémité. Les valeurs des extrémités d'une chaîne non-vide sont obtenues par `Chain_Left()` et `Chain_Right()` :

```
int Chain_Left  (Set const * chain) { return chain->tiles [      0         ].left;  }
int Chain_Right (Set const * chain) { return chain->tiles [chain->length-1].right; }
```

Question 9. On sait déja ajouter une tuile dans un jeu par la droite avec la fonction `Set_AddRight()`, il faut maintenant être capable d'insérer une tuile par la gauche. Implémenter à cet effet la fonction `Set_AddLeft()`. Toutes les tuiles déjà présentes sont décalées d'une case vers la droite. La fonction vérifie préalablement avec`assert()` que la capacité du jeu n'était pas déjà atteinte.

```
void Set_AddLeft (Set * set, Tile tile);
```

Question 10. Implémenter la fonction `Chain_FindConnectable()` qui cherche une tuile dans le ratelier `rack` qui soit connectable à l'une des extrémités de la chaîne `chain`. La fonction retourne l'index de la première tuile trouvée dans `rack->tiles`, ou `-1` si aucune n'est trouvée. (Lorsque la chaîne est vide, toutes les tuiles conviennent et on peut choisir arbirairement la tuile d'index `0`).

```
int Chain_FindConnectable (Set const * chain, Set const * rack)
```

Question 11. Implémenter la fonction `Chain_Connect()` qui connecte la tuile `tile` à la chaîne `chain` par la première extrémité trouvée qui vérifie la contrainte de connexion, en renversant la tuile si nécessaire. La chaîne peut être vide et dans ce cas, il n'y a pas de contrainte de connexion car il n'y a pas d'extrémité. (Lorsque la tuile n'est pas connectable, la tuile n'est pas ajoutée.)

```
void Chain_Connect (Set * chain, Tile tile);
```

5 Coups et Partie : `enum Move` et `struct Game`

On cherche à programmer une partie de Dominos à deux joueurs, l'ordinateur jouant contre lui-même, et sans stratégie. On utilise les règles simplifiées suivantes : Au départ, la pioche (*stock*) est constituée du jeu complet à V valeurs. La chaîne (*chain*) est vide ainsi que les rateliers (*racks*) des deux joueurs. Lors de la distribution, les joueurs piochent une tuile chacun leur tour jusqu'à ce que leur rack possède $V-1$ tuiles. Les joueurs jouent ensuite chacun leur tour. Quatre cas se présentent lors d'un tour :

1. Le joueur possède dans son rack une tuile connectable à la chaîne. Alors, il JOUE cette tuile.
2. Même cas, mais ce faisant, le joueur vide son rack. La partie s'arrête sur la VICTOIRE de ce joueur.
3. Sinon, le joueur PASSE en piochant une tuile dans le stock (sans pouvoir la jouer dans ce tour).
4. Même cas, mais la pioche est vide. La partie s'arrête sur la DÉFAITE de ce joueur.

Ces quatre coups sont représentés par le type énumératif `Move` :

```
typedef enum Move {
  MOVE_PLAY, MOVE_VICTORY, MOVE_PASS, MOVE_DEFEAT
} Move;
```

Le type structuré `Game` représente une partie en cours, et est constitué de 4 sets (la pioche, la chaine et un tableau de 2 racks). Le tour courant (*turn*) est le numéro du rack (0 ou 1) dont c'est le tour de jouer.

```
typedef struct Game {
  Set racks [2], stock, chain;
  int turn;
} Game;
```

Une partie pour $V =$ `nb_values` s'initialise avec la fonction `Game_Init()` :

```
void Game_Init (Game * game, int nb_values)
{
  Set_Init      (& game->stock, nb_values);
  Set_InitEmpty (& game->chain);
  Set_InitEmpty (& game->racks [0]);
  Set_InitEmpty (& game->racks [1]);
  game->turn= 0;
}
```

Question 12. Implémenter la fonction `Game_Deal()` qui distribue les tuiles aux deux racks d'une partie `game` déjà initialisée, en leur faisant chacun leur tour piocher une tuile au hasard dans le stock, jusqu'à ce qu'ils aient chacun `nb_tiles` tuiles.

```
void Game_Deal (Game * game, int nb_tiles);
```

Question 13. Implémenter la fonction `Game_PlayTurn()` qui fait exécuter un coup au joueur courant de la partie `game` (le joueur joue, gagne, passe ou perd). Il n'y a aucune interaction avec l'utilisateur ni affichage : l'ordinateur joue sans stratégie en fonction du résultat de `Chain_FindConnectable()`. Les racks, le stock et la chaîne sont modifiés, mais le tour courant reste inchangé. Le coup exécuté est retourné sous la forme d'un `Move`.

```
Move Game_PlayTurn (Game * game);
```

Question 14. Implémenter la fonction `Game_ShiftTurn()` qui fait passer le tour courant d'une partie `game` à l'autre joueur. Aucun coup n'est exécuté par cette fonction.

```
void Game_ShiftTurn (Game * game);
```

Question 15. Écrire un programme principal qui initialise une partie pour $V = 4$, distribue $V-1$ tuiles à chaque joueur, puis fait jouer ces derniers jusqu'à ce que la partie s'arrête sur la victoire ou la défaite de l'un d'eux. (Pour simplifier, on ne demande aucun affichage).

```
int main (void);
```