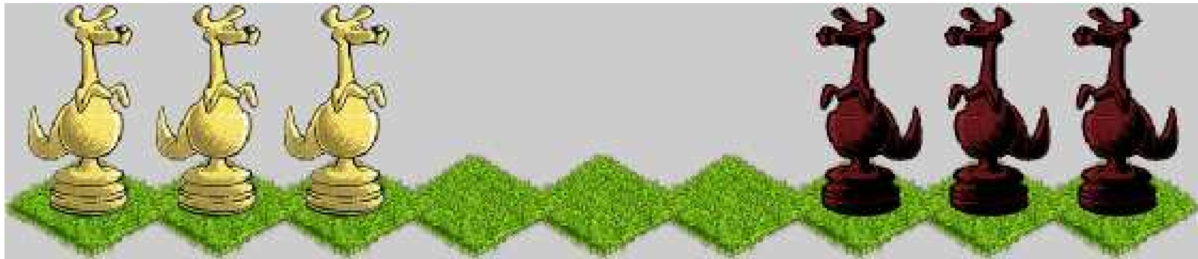
Aix-Marseille Université | Licence Math-Info – Semestre 2
Faculté des Sciences de Luminy | Mardi 12 Mai 2015

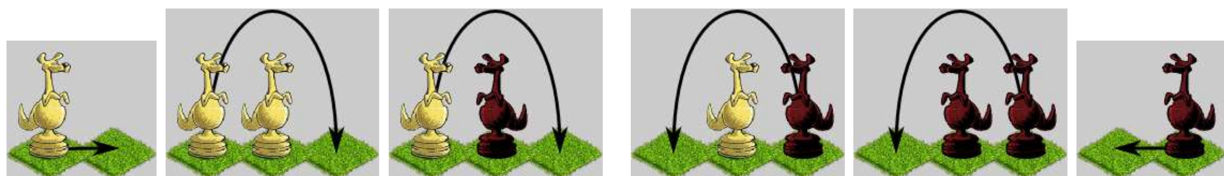
Programmation — Examen

Durée : 2 heures, Calculatrices et Documents Interdits



Le Jeu des Kangourous est un casse-tête pour un joueur (ou deux joueurs coopératifs), se jouant sur un plateau linéaire à N cases, avec P pions blancs et P pions noirs. Initialement, le plateau est trié **"blancs-noirs"**, *c'est-à-dire : les blancs sont groupés sur les P cases de gauche et les noirs sur les P cases de droite. Les deux groupes sont séparés par les $N-2P$ cases libres restantes. Le but est d'obtenir un plateau trié* **"noirs-blancs"**.

Pour cela, on doit avancer un pion de chaque groupe en alternance, en commençant par un pion blanc. Les blancs avancent vers la droite, les noirs vers la gauche. Un pion peut **avancer d'une case** *si la case d'arrivée est libre. Un pion peut* **faire un bond de deux cases** *si la case d'arrivée est libre et que la case intermédiaire est occupée par un pion (quelle que soit sa couleur). Sinon, il est bloqué. Aucun pion ne peut reculer, ne peut sortir du plateau, ou ne peut aller sur une case déjà occupée. La figure ci-dessous résume les coups possibles :*



Dans la figure ci-dessous, le plateau de gauche est trié **"noirs-blancs"**, *et la partie est gagnée. Dans le plateau de droite, les blancs sont bloqués. Si c'est au tour des noirs de jouer, seul le pion de gauche peut avancer d'une case, les autres étant bloqués. Mais cela ne débloquera pas les blancs, et la partie sera perdue au coup suivant.*

On souhaite modéliser la résolution de ce casse-tête par la force brute : on tente tous les coups possibles depuis un plateau donné en mémorisant les plateaux obtenus dans un historique à capacité fixe. Puis on tente à nouveaux tous les coups possibles sur ces nouveaux plateaux, et ainsi de suite, jusqu'à obtenir le plateau final, épuiser tous les coups (s'il n'y a pas de solution pour les N et P fixés), ou saturer la capacité de l'historique.

Il peut arriver que l'on génère deux plateaux jumeaux via deux séquences de coups différentes. Par exemple, si l'on note `[WWW---BBB]` l'état du plateau en début de partie (`W` pour *White*, `B` pour *Black*, `-` pour *Libre*), alors on peut atteindre l'état `[-WWW-B-BB]` par deux séquences de trois coups qui donnent le tour aux noirs :

```
séquence 1: [WWW---BBB] -> [W-WW--BBB] -> [W-WW-B-BB] -> [-WWW-B-BB] -> ...
séquence 2: [WWW---BBB] -> [WW-W--BBB] -> [WW-W-B-BB] -> [-WWW-B-BB] -> ...
```

On évitera alors d'explorer les coups possibles d'un plateau lorsqu'elles ont déjà été explorées à partir d'un plateau jumeau, afin d'éviter une explosion combinatoire inutile lors de la recherche de la solution.

Dans tout le devoir, pour implémenter une fonction, vous <u>devez RÉUTILISER</u> les fonctions des questions précédentes lorsque cela est possible, même si vous ne les avez pas implémentées. Vous pouvez écrire des fonctions auxilliaires non demandées explicitement dans l'énoncé.

<u>La question 15, Section 3, est une question de cours.</u>

1 Pions et Plateau

On se donne le type énumératif `Pawn` pour représenter les pions, qui définit les trois constantes énumératives `PAWN_NONE` (absence de pion), `PAWN_WHITE` (pion blanc), `PAWN_BLACK` (pion noir).

```
typedef enum Pawn {
  PAWN_NONE, PAWN_WHITE, PAWN_BLACK
} Pawn;
```

Question 1. Implémenter `Pawn_ToChar()` qui retourne la représentation caractère d'un pion : le caractère `W` pour un blanc, le caractère `B` pour un noir, le caractère `-` pour l'absence de pion, le caractère `?` pour les valeurs inconnues. Utiliser pour ce faire l'instruction `switch`.

```
char Pawn_ToChar (Pawn pawn);
```

On se donne trois constantes paramétrant le plateau : `BOARD_GROUP_SIZE` pour le nombre de pions par groupe, `BOARD_GAP_SIZE` pour le nombre de cases libres. `BOARD_SIZE` pour le nombre total de cases du plateau.

```
#define BOARD_GROUP_SIZE  3
#define BOARD_GAP_SIZE    3
#define BOARD_SIZE       (2 * BOARD_GROUP_SIZE + BOARD_GAP_SIZE)
```

Question 2. Implémenter la fonction `Board_PawnAtPosWhenSorted()` qui retourne le pion occupant la position `pos` dans tout plateau trié `"left-right"`, pour deux couleurs de pions `left` et `right` données. La fonction retourne `PAWN_NONE` pour les positions libres et les positions en dehors du plateau.

```
Pawn Board_PawnAtPosWhenSorted (int pos, Pawn left, Pawn right);
```

On se donne le type structuré `Board` représentant l'état d'un plateau :

```
typedef struct Board {
  Pawn squares [BOARD_SIZE];
  Pawn turn;
  int id, parent;
  int nb_moves;
} Board;
```

Le champ `squares` décrit l'occupation des cases par les pions. Le champ `turn` indique la couleur dont c'est le tour de jouer pour le coup suivant. Le champ `nb_moves` indique le nombre de coups déjà joués. Le champ `id` est un index localisant ce plateau dans l'historique de la partie (toujours `0` pour le plateau de début de partie). Le champ `parent` est un index localisant le plateau dont il est issu dans l'historique de la partie, à travers un coup joué (par convention, `-1` pour le plateau de début de partie sans parent). L'historique est étudié en Section 2.

Question 3. Implémenter la fonction `Board_Init()` qui initialise un plateau de début de partie `b`, c'est-à-dire trié dans l'ordre **"blancs-noirs"**, avec le tour aux blancs, sans aucun coup joué et sans parent.

```
void Board_Init (Board * b);
```

Un plateau est final lorsque le tour est aux blancs, et que ses cases sont triées dans l'ordre **"noirs-blancs"**.

Question 4. Implémenter le prédicat `Board_IsFinal()` qui teste si un plateau `b` est final.

```
int Board_IsFinal (Board const * b);
```

On considère le programme suivant :

```
int main (void)
{
  Board board;
  Board_Init (& board);
  printf ("%s\n", Board_ToString (& board)); // à implémenter (Question 5.)
  return 0;
}
```

La sortie produite par ce programme est :

```
{ id=0 squares=[WWW---BBB] parent=-1 turn=W nb_moves=0 }
```

Question 5. En utilisant `sprintf()`, implémenter la fonction `Board_ToString()` qui retourne un pointeur sur une chaîne statique représentant l'état d'un plateau `b` conformément au format de la sortie de `main()`.

```
char * Board_ToString (Board const * b);
```

Un entier `pos` représente une position valide si sa valeur est dans la plage des index de cases du plateau :

```
int Board_PosIsValid (int pos) { return 0 <= pos && pos < BOARD_SIZE; }
```

Un mouvement déplaçant un pion de la position `start` à la position `end` est valide sur le plateau `b` si `start` est une position valide occupée par un pion dont c'est le tour et si `end` est une position valide libre. (Ni la distance entre les positions ni l'occupation des cases intermédiaires n'interviennent dans la validité du mouvement telle que définie ici).

Question 6. Implémenter le prédicat `Board_MoveIsValid()` qui teste si un mouvement de la position `start` à la position `end` est valide sur le plateau `b`.

```
int Board_MoveIsValid (Board const * b, int start, int end);
```

Question 7. Implémenter `Board_PlayMove()` qui déplace un pion de la position `start` à la position `end` sur un plateau `b`. La fonction vérifie préalablement via `assert()` que le mouvement est valide au sens défini ci-dessus. Le tour passe à la couleur adverse, et le nombre de coups est incrémenté (mais l'id et le parent restent inchangés). La fonction renvoie un booléen indiquant si le plateau est devenu final.

```
int Board_PlayMove (Board * b, int start, int end);
```

Deux plateaux `b1` et `b2` sont jumeaux si et seulement si chaque case de `b1` a un état identique à sa case homologue dans `b2`, et si `b1` donne le tour au même groupe que `b2`.

Question 8. Implémenter le prédicat `Board_IsTwinOf()` qui teste si le plateau `b1` est le jumeau de `b2`.

```
int Board_IsTwinOf (Board const * b1, Board const * b2);
```

2 Historique

Un historique est une structure de type `History` permettant de mémoriser jusqu'à `HISTORY_CAPACITY` états de plateaux lors de l'exploration des coups d'une partie. Dans son état initial, un historique est de longueur `1` : il contient un unique plateau de début de partie à l'index `0`.

```
#define HISTORY_CAPACITY 10000

typedef struct History {
  Board boards [HISTORY_CAPACITY];
  int length;
} History;
```

```
void History_Init (History * h)
{
  Board_Init (& h->boards [0]);
  h->length= 1;
}
```

Les `length` plateaux d'un historique `h` sont stockés dans le tableau `boards` aux index `0` à `length-1`. On ajoute toujours un nouveau plateau à la fin de l'historique, et son `id` devient l'index le localisant dans `h`. Un historique est plein lorsque sa capacité est atteinte.

```
int History_IndexIsValid (History const * h, int id) { return 0 <= id   &&   id < h->length; }
int History_IsFull       (History const * h)         { return h->length == HISTORY_CAPACITY; }
```

Lorsque l'on joue un coup d'une position `start` à une position `end` sur un plateau d'index `id` dans un historique `h`, on procède en deux temps. D'abord, on ajoute une copie du plateau avant le coup à la fin de `h` via une fonction `History_AddCopyOf()`. Le plateau d'index `id` devient à cette occasion le parent de la copie ajoutée. Cette copie est ensuite altérée via la fonction `Board_PlayMove()` déjà codée, afin de refléter l'état du plateau après le coup. Le plateau d'index `id` reste inchangé.

```
int History_PlayMove (History * h, int id, int start, int end)
{
  History_AddCopyOf (h, id); // à implémenter (Question 9.)
  Board * b= & h->boards [h->length -1];
  return Board_PlayMove (b, start, end);
}
```

Question 9. Implémenter `History_AddCopyOf()` qui rajoute une copie du plateau d'index `id` à la fin de l'historique `h`. La copie diffère en deux points de l'original : leur champ `id` puisque les deux plateaux sont à des index différents dans `h`, et leur champ `parent` puisque l'original devient le parent de la copie. Par ailleurs, la fonction vérifie préalablement via `assert()` que `id` est un index valide, et que `h` n'est pas déja plein.

```
void History_AddCopyOf (History * h, int id);
```

Le pas d'un mouvement d'une position `start` à une position `end` est la différence `end-start`. Les pas autorisés par les règles sont `-1`, `-2` (pour les noirs), `+1`, `+2` (pour les blancs). Par convention, on représente un blocage par le pas `0`. Pour une position donnée, il y a un unique pas parmi ces cinq valeurs qui soit possible selon les règles.

Question 10. Implémenter la fonction `Board_StepAtPos` qui retourne l'unique pas qui soit possible à la position `start` d'un plateau `b`. En cas d'impossibilité (non possession du tour par le pion de la case, cases candidates pour l'arrivée hors plateau ou occupées) le pas `0` est retourné. Par ailleurs, la fonction vérifie préalablement via `assert()` que `start` est un index valide.

```
int Board_StepAtPos (Board const * b, int start);
```

On cherche à jouer tous les coups du tour courant d'un plateau. Par exemple, depuis le plateau initial `[WWW---BBB]` avec le tour aux blancs, on peut jouer les deux coups blancs suivants :
`[W-WW--BBB]` (mouvement de la position `1` à `3`, avec pas de `+2`) et
`[WW-W--BBB]` (mouvement de la position `2` à `3`, avec pas de `+1`).

Question 11. Implémenter la fonction `History_PlayBoard()` qui joue tous les coups possibles du tour courant du plateau d'index `id` dans l'historique `h` avec `History_PlayMove()`. Cependant, la fonction s'arrête dès que la capacité de `h` est atteinte (échec) ou lorsqu'un plateau final est obtenu après un coup (succès). La fonction retourne un booléen indiquant son succès. Par ailleurs, la fonction vérifie préalablement via `assert()` que `id` est un index valide.

```
int History_PlayBoard (History * h, int id);
```

Question 12. Implémenter la fonction `History_FindMostRecentTwin()` qui retourne l'index du jumeau le plus récent du plateau d'index `id`, et situé avant lui dans l'historique `h`. On retourne `-1` si aucun jumeau n'est trouvé. Par ailleurs, la fonction vérifie préalablement via `assert()` que `id` est un index valide.

```
int History_FindMostRecentTwin (History const * h, int id);
```

Question 13. Implémenter la fonction `History_PlayAll()` qui joue tous les coups possibles pour tous les plateaux de l'historique `h` qui n'on pas de jumeau avant eux. Les plateaux ainsi rajoutés subissent le même sort et ainsi de suite. On s'arrête de jouer lorsque l'historique est plein (échec mémoire), lorsque tous les coups sont épuisés (échec sur blocage général) ou lorsque l'on atteint un plateau final après un coup (succès). La fonction retourne un booléen indiquant son succès.

```
int History_PlayAll (History * h);
```

Question 14. Implémenter la fonction `History_PrintGame()` qui affiche sur le flux de sortie `file` le plateau d'index `id` et tous ses `nb_moves` ancêtres (parent, grand-parent, arrière-grand-parent, etc) dans l'historique `h`, jusqu'au plateau de début de partie, à raison d'un plateau par ligne.

```
void History_PrintGame (History const * h, int id, FILE * file);
```

Le programme final est :

```
int main (void)
{
  History history;
  History_Init (& history);
  int is_final= History_PlayAll (& history);

  if (is_final) History_PrintGame (& history, history.length-1, stdout);
  else          fprintf (stdout, "no solution found.\n");

  return 0;
}
```

Pour un plateau de 9 cases et 3 pions par groupe, une solution en 24 coups est trouvée après l'exploration 2652 états depuis l'état initial. On obtient la sortie suivante (les nombres ont été alignés pour plus de clarté) :

```
{ id=2652 squares=[BBB---WWW] parent=2615 turn=W nb_moves=24 }
{ id=2615 squares=[B-BB--WWW] parent=2565 turn=B nb_moves=23 }
{ id=2565 squares=[B-BB-WW-W] parent=2502 turn=W nb_moves=22 }
{ id=2502 squares=[B-B-BWW-W] parent=2417 turn=B nb_moves=21 }
{ id=2417 squares=[B-BWB-W-W] parent=2273 turn=W nb_moves=20 }
{ id=2273 squares=[-BBWB-W-W] parent=2071 turn=B nb_moves=19 }
{ id=2071 squares=[-BBWBW--W] parent=1849 turn=W nb_moves=18 }
{ id=1849 squares=[-BBW-WB-W] parent=1651 turn=B nb_moves=17 }
{ id=1651 squares=[-BBW-WBW-] parent=1435 turn=W nb_moves=16 }
{ id=1435 squares=[-B-WBWBW-] parent=1184 turn=B nb_moves=15 }
{ id=1184 squares=[-BW-BWBW-] parent= 927 turn=W nb_moves=14 }
{ id= 927 squares=[-BW-BW-WB] parent= 711 turn=B nb_moves=13 }
{ id= 711 squares=[-BWWB--WB] parent= 527 turn=W nb_moves=12 }
{ id= 527 squares=[-BWW-B-WB] parent= 383 turn=B nb_moves=11 }
{ id= 383 squares=[WB-W-B-WB] parent= 273 turn=W nb_moves=10 }
{ id= 273 squares=[WB-W--BWB] parent= 190 turn=B nb_moves= 9 }
{ id= 190 squares=[WB-W-WB-B] parent= 134 turn=W nb_moves= 8 }
{ id= 134 squares=[WB-W-W-BB] parent=  87 turn=B nb_moves= 7 }
{ id=  87 squares=[WBW--W-BB] parent=  52 turn=W nb_moves= 6 }
{ id=  52 squares=[W-WB-W-BB] parent=  24 turn=B nb_moves= 5 }
{ id=  24 squares=[W-WBW--BB] parent=   9 turn=W nb_moves= 4 }
{ id=   9 squares=[W-W-WB-BB] parent=   3 turn=B nb_moves= 3 }
{ id=   3 squares=[W-WW-B-BB] parent=   1 turn=W nb_moves= 2 }
{ id=   1 squares=[W-WW--BBB] parent=   0 turn=B nb_moves= 1 }
{ id=   0 squares=[WWW---BBB] parent=  -1 turn=W nb_moves= 0 }
```

Pour un plateau de 8 cases et 3 pions par groupe, il n'y a pas de solution (échec sur blocage général).

3 Question de Cours

Question 15. Citer les trois grandes catégories de fonctions que l'on utilise dans un langage impératif. Pour chacune d'elles, énoncer brièvement la règle de nommage à laquelle elle obéit (catégorie grammaticale et conjugaison), et citer en exemple un identificateur trouvé dans cet examen.