# Devoir à la maison

Julien JACQUET 21400579

7 décembre 2016

**Note :** Pour la durée de l'exercice, les tableaux débutent à la case 1.

**Note :** Pour la durée de l'exercice, la lecture hors du tableau n'entraînera pas d'erreur fatale et renverra simplement une réponse négative, les tests continueront indépendamment.

# 1

Pour stocker l'état du plateau de solitaire, on va utiliser un tableau à doubles entrées qui représentera le plateau, il servira à stocker l'état des cases et donc du plateau dans son intégralité. La taille du tableau dépendra du type de solitaire (*Wikipedia*), on utilisera un modèle européen à 37 trous pour cet exercice.

```
struct SOLITAIRE(){
  int PLATEAU[7][7];
  //le tabelau est de type int car on stockera l'etat des
      cases avec des entiers
};
```

**Etat des cases :**

-1 : la case n'est pas une case (le solitaire européen n'est pas un carré, contrairement à notre représentation, il faut donc attribuer les cases qui ne seront pas utilisées).
0 : la case est vide.
1 : il y a un pion dans la case.

Pour représenter un pion on va simplement lui attribuer des coordonnées du plateau. Donc des entiers.

```
struct POS(){
  int x;
  int y;
};
```

# 2

Pour savoir si le mouvement d'un pion $p1$ a un pion $p2$ est possible (selon les règles du solitaire) il faut tester plusieurs choses. $p1$ et $p2$ doivent tenir dans $s$ (leurs coordonnées ne peuvent pas être inférieures à 1 ni supérieures a 7).

$p1$ et $p2$ sont ils dans la zone jouable ?
$p1$ est il un pion ?
$p2$ est une case vide ?
$p2$ est il suffisamment proche de $p1$ ? (à une case d'écart)

Notre fonction renverra un booléen.

```
BOOL mouvement_test (SOLITAIRE s, POS p1, POS p2){

//p1 et p2 sont ils dans la zone jouable?
  if( (s[p1.x][p1.y] == -1) && (s[p2.x][p2.y] == -1)){
  return false;
  }

//p1 est un pion?
  if( s[p1.x][p1.y] != 1){
  return false;
  }

//p2 pret a recevoir p1?
  if( s[p2.x][p2.y] != 0){
  return false;
  }

//verifcation de la condition de distance (doit remplir une
    des conditions suivantes)
  if( s[p2.x][p2.y] != (s[p1.x+2][p1.y] || s[p1.x][p1.y+2]
      || s[p1.x-2][p1.y] || s[p1.x][p1.y-2]) ){
  return false
  }
//dans tous les autres cas
return true;
}
```

La complexité de cet algorithme est dans le pire des cas de 4, puisque le pire des cas est le cas où le mouvement est possible les 4 tests sont alors exécutés, la complexité vaut alors 4.

## 3

Pour qu'il y ait un déplacement possible de pion sûr $s$ il faut qu'il y ait une case vide et qu'à deux cases de distance, il y ait un pion. On va donc chercher une case vide et ensuite tester s'il y un pion qui remplit les bonnes conditions.

```
BOOL deplacement_possible(SOLITAIRE s){
  int i,j;
  for(i=1; i < 7; i++){
      for(j=1; j < 7; j++){
        if( (s[i][j] == 0) && (s[i+2][j] == 1 || s[i][j+2]
            == 1 || s[i-2][j] == 1|| s[i][j-2] == 1))
        return true;
        //les conditions sont remplies, un deplacement est
            possible
      }
  }
  return false;
  //les conditions ne sont pas remplies, pas de deplacement.
}
```

La complexité de cet algorithme est directement liée à la taille des doubles boucles imbriquées, elles sont de tailles 7, ce qui nous donne $7 * 7 = 7^2 = 49$. Si l'on remplace 7 par $n$ en fonction des différents types de solitaires ont obtient une complexité en $O(n^2)$.

# 4

On peut fortement s'inspirer de l'algorithme précédent pour cette question.

```
SOLITAIRE jouer_tant_qu_on_peut_encore (SOLITARE s){
    //l'algorithme prends en argument un solitaire et renvoi
    un solitaire modifie.

int i, j;
int un_mouvement_a_ete_effectue = 1; //pour savoir quand
    s'arreter
POS p1, p2;


while(un_mouvement_a_ete_effectue == 1){ //l'alogrithme
    doit s'arreter lorsqu'a la completion des boucles aucun
    mouvement n'a ete effectue.

  for(i=1; i < 7; i++){
      for(j=1; j < 7; j++){
        if( (s[i][j] == 0) && (s[i+2][j] == 1 || s[i][j+2]
            == 1 || s[i-2][j] == 1|| s[i][j-2] == 1)){
        p1.x = i; p1.y = j;

          if(s[i+2][j] == 1){
            p2.x= i+2; p2.y = j;
            effectuer_mouvement(p1, p2);
            un_mouvement_a_ete_effectue = 1;
            }

          else if(s[i][j+2] == 1){
            p2.x= i; p2.y = j+2;
            effectuer_mouvement(p1, p2);
            un_mouvement_a_ete_effectue = 1;
            }

          else if(s[i-2][j] == 1){
            p2.x= i-2; p2.y = j;
            effectuer_mouvement(p1, p2);
            un_mouvement_a_ete_effectue = 1;
            }

          else if(s[i][j-2] == 1){
            p2.x= i; p2.y = j-2;
            effectuer_mouvement(p1, p2);
            un_mouvement_a_ete_effectue = 1;
            }
         }
```

```
39          }
40        }
41      }
42  }
```

L'idée est qu'une fois que des mouvements ont été effectués, il faut vérifier si de nouveaux mouvements ne sont pas possibles. D'où la condition d'arrêt. Quand aucun mouvement n'a été effectué par la boucle, il n'y a pas d'autres mouvements possibles.

La complexité de cet algorithme est bien plus difficile à évaluer : Les deux boucles internes ont toujours la même complexité qu'a la question précédente. Mais le while est difficile à évaluer. Dans le pire des cas le while s'effectuera le nombre de cases qu'il y a sur le plateau, définissons le nombre de cases par $n$.

Si $n$ est le nombre de cases alors la double boucle intérieure à une complexité de $n$, le while également. Ce qui donnne $n * n = n^2$. La complexité de cet algorithme pour $n$ valant le nombre de cases est de l'ordre de $O(n^2)$.

Cependant si l'on prend $n$ la largeur des tableaux (comme à la question précédente) la complexité de cet algorithme est alors de $n*n*n*n = n^4.O(n^4)$.

# 5

Pour représenter l'ensemble des parties de solitaire possible, on peut utiliser un arbre des possibilités. Il suffit alors de suivre l'arbre de la racine jusqu'aux feuilles les branches représentants différents mouvements possibles. La feuille de l'arbre que l'on a suivie donnerait le nombre minimal de pions restants sur le plateau.