IN 101 - Examen
11 janvier 2022
François Pessaux

Éléments de correction

1 Jeu de dames bizarre ($\sim 60\%$)

1.1 Q1.1

Identifiez les grandes étapes du programme, sans descendre dans le détail des fonctions que vous allez écrire. Cela représentera en quelque sorte votre futur `main`.

Solution

```
programme :
  Vérifier qu'il y a des arguments et les récupérer
  Charger le damier
  Vérifier que la position initiale est compatible avec la taille du damier.
  Vérifier qu'il y a bien un pion à la position initiale
  Lancer l'exploration
  Afficher le résultat
  Libérer la mémoire
```

1.2 Q1.2

Identifiez les grandes étapes de la fonction de chargement du damier. Quels sont ses domaines d'entrée et de sortie ? La gestion des cas de fichier mal formé sera un bonus.

Solution

Elle doit prendre en entrée le nom du fichier. En sortie, elle doit retourner le damier mais aussi la taille. Pour cette dernière on utilisera un entier que l'on passera par adresse, ce qui rajoute en fait un paramètre.

```
load_game (fname, *size) :
  Ouvrir fichier
  Vérifier succès
  Lire taille du damier -> size
  Allouer la matrice carrée de taille size
  Vérifier allocation
  Lire une chaîne du fichier
  y = 0
  Tant que pas fin de fichier
    La copier dans le damier à la ligne y
    y <- y + 1
    Lire une chaîne du fichier
  Fermer fichier
  Vérifier que l'on a bien lu toutes les lignes
  Retourner le damier et la taille
```

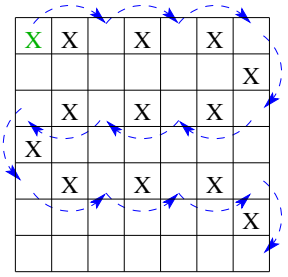
1.3 Q1.3

Quel est le nombre maximal de sauts en fonction de la largeur du damier ? Si vous n'arrivez pas à répondre à cette question, ne perdez pas tout votre temps, vous utiliserez alors une valeur arbitraire (assez grande). Réfléchissez à quoi ce nombre va vous servir dans l'algorithme d'exploration.

Solution

Considérons uniquement le cas d'une grille de taille impaire. En effet si la grille est de taille paire, alors en partant du haut on n'attendra jamais le bas. Donc le chamin le plus long sera celui de la grille de taille impaire immédiatement précédente.

Le chemin le plus long consiste à parcourir le damier de haut en bas en alternant droite / gauche à chaque ligne, en partant de la position $(0, 0)$.



Sur chaque ligne, on fait $\frac{size-1}{2}$ sauts. On parcourt $\frac{size-1}{2}$ lignes. Donc, ça fait déjà $(\frac{size-1}{2})^2$. Ensuite, il faut compter les sauts verticaux. On en fait aussi $\frac{size-1}{2}$. Donc, la longueur maximale d'un chemin pour un damier de largeur $size$ est donc $(\frac{size-1}{2})^2 + \frac{size-1}{2}$.

Il va servir à retourner la valeur « pas de chemin possible ». Il va aussi servir à initialiser le tableau de distances qui nous permettra d'éviter de re-parcourir un chemin déjà exploré (mémoïsation, question Q1.5).

1.4 Q1.4

Identifiez de quelle manière vous allez explorer le damier. On vous rappelle que l'on n'a pas le droit de remonter. Attention, pensez au cas gênant qui consisterait à sauter d'un côté puis immédiatement après de l'autre coté... à l'infini.

Solution

Pour éviter de boucler, il faut interdire de se déplacer à l'opposé du coup précédent, donc interdire d'aller dans la direction d'où l'on vient. On va donc mémoriser dans un argument de la fonction (`pmov` pour « previous move ») quel déplacement nous a fait arriver sur la case courante. On remarque que, puisque l'on n'a pas le droit de remonter, notre exploration va récursivement explorer à gauche, en bas et à droite, mais jamais en haut. Donc, on n'a pas de risque de « boucler verticalement ».

```
enum prev_mov_t { Left, Right, Bottom }

int explore (int x, int y, char **grid, int size, enum prev_mov_t pmov) =
 int g = max_nb_jumps, b = max_nb_jumps, d = max_nb_jumps
  Si arrivé sur la dernière ligne en bas
    Retourner 0

  /* Gauche. */
  Si (x - 2 >= 0 && grid[y][x - 1] == 'X' && pmov != Right)
    g <- 1 + explore (x - 2, y, grid, size, Left)
  /* Bas. */
  Si (y + 2 < size && grid[y + 1][x] == 'X')
    b <- 1 + explore (x, y + 2, grid, size, Bottom)
  /* Droite. */
  Si (x + 2 < size && grid[y][x + 1] == 'X' && pmov != Left)
    d = 1 + explore (x + 2, y, grid, size, Right)
  best <- min (g, min (b, d))
  Retourner best
```

1.5 Q1.5

Si ce n'est déjà fait, réfléchissez à comment vous pourriez gagner du temps durant l'exploration. Par exemple, considérez le damier ci-contre du fichier `game6.txt`. En partant de la position $(x, y) = (5, 0)$, quel(s) chemin(s) pouvez-vous emprunter ? Que remarquez-vous à propos de la fin du/des chemin(s) ? Qu'avez-vous envie de faire pour gagner du temps ?

Remarque : si vous ne trouvez pas la réponse à cette question et que votre algorithme semble fonctionner, ne perdez pas tout votre temps, vous y reviendrez éventuellement plus tard s'il vous reste du temps.

```
11
..X.XXX.X..
.X.......X.
...........
.X.......X.
..X.X.X.X..
.......X...
...........
.......X...
..X.X.X...X
.X.......X.
....X......
```

Solution

Il y a 2 chemins qui se rejoignent. Leur suffixe est commun et lorsque l'on l'a déjà parcouru une fois, on ne veut pas le refaire. On va donc mémoïser. On prend en argument supplémentaire un tableau dans lequel on va mémoriser la meilleure distance pour faire une dame à partir d'une position (x, y). Ce tableau devra être alloué dans le `main` et initialement rempli avec la distance maximale + 1 (valeur calculée en question Q1.3).

Dans l'algorithme d'exploration, avant de parcourir récursivement le damier, on regarde si dans le tableau se trouve déjà une distance $< long\_max + 1$. Si oui on retourne directement cette valeur. Si non, on parcourt récursivement le damier, on en déduit la longueur du chemin le plus court à partir de la position courante et avant de retourner cette longueur, on la mémorise dans le tableau pour la position courante.

```
int explore (int x, int y, char **grid, int size, int **dists,
             enum prev_mov_t pmov) =
  int g = max_nb_jumps, b = max_nb_jumps, d = max_nb_jumps
  Si arrivé sur la dernière ligne en bas
    dists[y][x] <- 0
    Retourner 0

  Si (dists[y][x] < max_nb_jumps) Retourner dists[y][x]

  /* Gauche. */
  Si (x - 2 >= 0 && grid[y][x - 1] == 'X' && pmov != Right)
    g <- 1 + explore (x - 2, y, grid, size, dists, Left)
  /* Bas. */
  Si (y + 2 < size && grid[y + 1][x] == 'X')
    b <- 1 + explore (x, y + 2, grid, size, dists, Bottom)
  /* Droite. */
  Si (x + 2 < size && grid[y][x + 1] == 'X' && pmov != Left)
    d <- 1 + explore (x + 2, y, grid, size, dists, Right)
  best = min (g, min (b, d))
  dists[y][x] <- best
  Retourner best
```

2 Chiffres décroissants ($\sim 30\%$)

Solution

La fonction `dec` effectuant la vérification prendra en argument un entier et retournera une valeur de vérité, donc un booléen.

La fonction principale sera en charge de l'appeler. Elle commencera par vérifier la disponibilité de l'argument de ligne de commande, puis appellera la fonction `dec` et affichera la chaîne « Ok » ou « Ko » en fonction de si `dec` a retourné vrai ou faux.

Il faut vérifier si l'écriture décimale de l'entier, lue de gauche à droite est composée de chiffres décroissants de 1 à chaque fois. Donc, il va falloir « tronçonner » le nombre, à chaque fois se souvenir du précédent chiffre que l'on avait vu et vérifier que le chiffre courant est inférieur de 1 au précédent. On a donc besoin d'une mémoire *c_pred* pour se souvenir du chiffre précédent.

Pour « tronçonner » le nombre n, il faut que l'on traite les chiffres en commençant le plus à gauche. Comment les obtient-on ? Pour extraire les chiffres d'un nombre, rien de tel que des coups de division (entière) par 10 et de modulo par 10. Si l'on calcule $n \,\%\, 10$ on obtient le chiffre le plus à droite et en calculant $n \,/\, 10$ on obtient le nombre représentant la partie gauche de n. Zut, on voulait avoir les chiffres dans l'autre sens...

Ah oui, mais si l'on inverse le problème, on n'a qu'à vérifier que l'écriture de **droite à gauche** est composée de chiffres **croissants** de 1 à chaque fois. Donc on a une simple boucle qui va extraire le chiffre courant c en faisant $c \leftarrow n \,\%\, 10$ et le « reste » du nombre en faisant $n \leftarrow n \,/\, 10$. À chaque tour, on vérifie si c est égal à $c\_pred + 1$ qui sera le chiffre vu à l'itération précédente. Si ce n'est pas le cas, alors le test échoue, sinon on continue. Et il faudra dire que $c\_pred \leftarrow c$ pour l'itération suivante.

Il nous reste 2 questions. On arrête la boucle quand ? Manifestement lorsque n vaut 0. Quelle est la valeur initiale du « chiffre précédent » ? Zut, on ne peut pas la connaître à l'avance. On n'a qu'à « dérouler » notre boucle une fois, en faisant un coup de modulo et division, comme cela on aura la valeur pertinente de *c_pred*.

Mais que se passe-t-il si initialement n est négatif ? Oui, car on nous a dit que n était un entier. Le modulo sur des négatifs est une opération pas forcément très claire...En plus, « si je me souviens bien, le prof nous a dit que l'opérateur de modulo se comporte différemment sur les négatifs selon les langages ». Autant aller au plus simple : se ramener au cas positif puisque de toute façon le signe n'est pas un chiffre et il ne change pas la vérification. Donc, avant toute chose, si n est négatif, on prend son opposé. Et ensuite on commence à travailler.

```
bool dec (n) :
  int pred_c, c

  Si n < 0 alors n <- -n
  pred_c <- n % 10
  n <- n / 10
  Tant que n <> 0
    c <- n % 10
    Si c <> pred_c + 1 alors retourner faux
    pred_c <- c
    n <- n / 10

  retourner vrai
```

Remarque : il est possible d'écrire cet algorithme de manière récursive. Il y a même 2 solutions. La première consiste à vérifier les chiffres de droite à gauche, comme fait ici, en inversant la condition donnée dans l'énoncé. La seconde consiste à vérifier les chiffres de gauche à droite sans inverser la condition. La seule différence sera que dans le premier cas on effectue la vérification avant l'appel récursif et dans le second on l'effectue après.