IN 101 - TD 03

Solution

Attention : désormais l'implémentation des exercices sera à faire en fin de TD, après que nous ayons examiné toutes les questions de réflexion et de structuration des algorithmes. Laissez le clavier...

1 Point fixe d'un tableau

≪Écrivez une fonction qui vérifie s'il existe un point fixe dans un tableau ≫

On appelle ≪point fixe d'un tableau ≫ un indice `i` tel que `t[i] == i`.

Soit un tableau `t` contenant, **par hypothèse**, uniquement des des entiers naturels tous différents et triés en ordre croissant strict.

Q 1.1. Proposez un algorithme pour une fonction `fixpt` qui prend en argument un tel tableau et détermine s'il existe un point fixe dedans.

Solution

Si à un endroit dans le tableau, on a t[i] == i, par exemple, t[5] == 5, alors vu que tous les naturels sont strictement croissants, il ne peut y avoir dans les **5 cases précédentes**, que **5 autres** valeurs **inférieures strictement à 5**, **positives** et **différentes entre elles**.

Donc 0, 1, 2, 3 et 4. Comme elles doivent être en ordre **strictement croissant**, elles ne peuvent être que dans l'ordre 0, 1, 2, 3 et 4.

Donc forcément, s'il existe un point fixe quelque part, toutes les cases précédentes ont aussi un point fixe, en particulier la case 0. Il n'y a donc qu'à tester si la case 0 est un point fixe.

Attention, pour accéder à la case 0, il faut que le tableau contienne au moins une valeur. Donc si le tableau est vide, il faut gérer le cas particulier. Un tableau vide ne contient pas de point fixe manifestement. L'algorithme est alors simplement :

```
fixpt (t, len) =
  Si len = 0 retourner faux
  Retourner t[0] == 0
```

ou encore plus court :

```
fixpt (t, len) =
  Retourner (len <> 0) et (t[0] == 0)
```

2 Fou aux échecs

≪Écrivez une fonction qui vérifie si le déplacement d'un fou d'une position initiale à une position finale en un coup est conforme aux règles du jeu d'échecs. ≫

On rappelle que les échecs se jouent sur un damier de taille 8×8 et qu'un fou ne peut se déplacer qu'en **diagonale**.

On modélise le damier par une matrice de coordonnées comprises entre 0 et 7 inclus.

On **ne cherche pas** à vérifier si la position initiale est atteignable par les règles du jeu d'échec (par exemple, de par la disposition initiale des pièces, il est impossible de trouver un des deux fous noirs sur une case blanche au cours d'une partie respectant les règles).

L'échiquier ne contient que le fou (pas d'autres pièces qui pourraient le bloquer.

Q 2.1. Quel sont les domaines des entrées et des sorties ?

Solution

Le jeu d'échec se jouant sur un damier, chaque position est un point du plan, donc un **couple** d'entiers. Chaque coordonnée doit appartenir à $[0; 7]$. La fonction doit prendre en argument la position initiale et la position finale, donc elle doit prendre 2 **couples** d'entiers. Plutôt que fabriquer une `struct` à cet effet, nous nous simplifierons la tâche en passant 4 entiers à notre fonction.

Nous **ne tenterons pas** de restreindre ces entiers à être positifs en utilisant des `unsigned int` pour une raison que nous verrons après avoir répondu à la question suivante.

La fonction devant dire si « oui » ou « non » le déplacement est légal, elle retourne une valeur de vérité donc un **booléen**.

Q 2.2. Quelle sont les relations entre les positions initiales et finales d'un déplacement légal ?

Solution

Notons (x, y) les coordonnées de la position initiale et (x', y') celles de la position finale. Si le déplacement est diagonal, il faut que pour un certain k l'on ait :

$$\begin{aligned} x' &= x \pm k \\ y' &= y \pm k \end{aligned}$$

Donc :

$$\begin{aligned} x' &= x + k \quad \Rightarrow \quad k = x' - x \\ &\text{ou} \\ x' &= x - k \quad \Rightarrow \quad k = x - x' \end{aligned}$$

et

$$\begin{aligned} y' &= y + k \quad \Rightarrow \quad k = y' - y \\ &\text{ou} \\ y' &= y - k \quad \Rightarrow \quad k = y - y' \end{aligned}$$

Plutôt que de réfléchir en terme des combinaisons de cas, il suffit de remarquer que $x - x'$ et $x' - x$ se regroupent en $\mid x - x' \mid$. Donc le déplacement est légal si l'on a :

$$\begin{aligned} k &= \mid x - x' \mid \\ k &= \mid y - y' \mid \end{aligned}$$

et finalement si $\mid x - x' \mid = \mid y - y' \mid$.

Nous voyons ici pourquoi nous n'avons pas décidé d'utiliser des `unsigned int`. En effet, la soustraction de 2 `unsigned` donnerait un `unsigned` (avec quelle valeur si $x < y$?) et rendrait donc la fonction de valeur absolue totalement inutilisable !

Tout cela ne contraint pas la position d'arrivée à rester dans le damier. Il faudra donc en plus vérifier que x' et $y' \in [0; 7]$.

Et un ≪déplacement nul ≫ ? Avoir les positions initiale et finale égales, est-ce un déplacement conforme aux règles du jeu ? Le choix d'interprétation est libre mais **il faut se poser la question**. Pour la correction de cet exercice le choix est de considérer que c'est illégal puisqu'il n'y a pas à proprement parler de ≪déplacement ≫.

3 État des finances

≪On souhaite écrire un programme permettant de connaître le solde de chacune des personnes d'un groupe où les membres se doivent de l'argent. ≫

Exemple pratique

— Charlie doit à Alice 10.
— Alice doit à Bob 15.
— Charlie doit à Bob 7.
— Bob doit à Charlie 5.

La question à laquelle répondre est ≪quel est le solde de chacun après règlement de ces dettes ? ≫ La réponse (affichage souhaité) :

```
$ ./money.x bill2.txt
Alice : -5
Bob : 17
Charlie : -12
```

Q 3.1. Proposez une esquisse d'algorithme pour résoudre le problème de calcul des soldes (on s'occupera de l'acquisition des données plus loin).

Solution

Lorsqu'une personne A rembourse sa dette d à une personne B, on retire d du solde de A et on ajoute d à celui de B. Il suffit donc de mémoriser le solde de chaque personne et d'itérer sur les dettes en effectuant l'addition et la soustraction entre soldes concernés.

Pour mémoriser les soldes, on utilisera un **tableau** `balances` où chaque case correspond au solde d'une personne. Il faudra être capable de savoir quelle case (quel indice de tableau) correspond à quelle personne. On peut donc utiliser un second tableau `names` dans lequel on stockera les noms. Ainsi, à un indice i, `balances[i]` correspondra au solde de la personne de nom `names[i]`.

names		**balances**
Alice	0	
Bob	1	
Charlie	2	

Notons que l'on pourrait également utiliser un tableau à 2 dimensions, ou un tableau mémorisant des couples (par des `struct`) $(nom, solde)$.

```
Pour chaque ligne de transaction
  Le nom devant de l'argent est le premier
  Celui recevant et le montant est le second
  index_debit <- trouver l'index du nom devant de l'argent
  index_credit <- trouver l'index du nom recevant de l'argent
  balances[index_debit] <- balances[index_debit] - montant
  balances[index_credit] <- balances[index_credit] + montant
```

Nous n'avons pas envie de saisir au terminal toutes les informations, nous allons utiliser un fichier **texte** comme entrée. Pour manipuler un fichier **texte** il faut :

1. L'ouvrir : fonction `fopen`.
2. Y lire (ou y écrire) ce que l'on veut : fonction `fscanf` (ou `fprintf`).
3. Le fermer une fois que l'on n'en a plus besoin : fonction `fclose`.
4. Pour tester si l'on est arrivé à la **fin du fichier**, il faut avoir fait **au moins une lecture** avant d'utiliser la fonction dédiée : `feof`. C'est important car ça joue sur la forme de l'algorithme.

On ne s'intéresse pas à toutes les fonctionnalités des fichiers, on ne regarde que ce dont nous avons besoin. Nous détaillerons ces fonctions dans la partie implémentation. Il faut actuellement juste savoir que `fscanf` fonctionne comme `scanf` (que vous connaissez déjà), sauf qu'au lieu de récupérer les données au clavier, elle les récupère dans un fichier.

La structure d'un fichier de données est fixe. La **première** ligne contient, le nombre n de personnes impliquées dans le problème. Ensuite, séparés par des espaces ou des retours à la ligne (ce qui ne change rien pour `scanf`/`fscanf`), les n noms des personnes.

Suivent un nombre **quelconque** de lignes comportant à chaque fois : le nom de la personne **devant** de l'argent suivi du nom de celle **recevant** l'argent suivi du **montant** (un entier), le tout séparé par des **espaces**.

```
$ more bill2.txt
3
Alice Bob Charlie
Charlie Alice 10
Alice Bob 15
Charlie Bob 7
Bob Charlie 5
MyMachine:~
```

Q 3.2. Complétez votre esquisse d'algorithme de la question Q3.1 pour acquérir les données d'entrée.

Solution

```
Ouvrir le fichier
Lire la première valeur qui est le nombre de participants (nb_persons)
Créer le tableau de taille nb_person pour mémoriser les noms
Lire nb_person noms et les stocker dans le tableau
```

```
Créer le tableau de taille nb_person pour stocker les soldes, en mettant
                    0 dans chaque case
Lire 2 noms et un montant
Tant que la fin de fichier n'est pas atteinte
  Le nom devant de l'argent est le premier lu
  Celui recevant le montant est le second lu
  Rechercher pour chaque nom à quel indice du tableau de noms il se trouve
  Faire l'addition et la soustraction qui vont bien à ces indices dans
               le tableau des soldes
  Lire 2 noms et un montant
Fermer le fichier
Parcourir les 2 tableaux en paralèlle
  Afficher le nom et le solde final de la personne concernée
Libérer les tableaux créés
```

Q 3.3. Comment allez-vous retrouver l'indice du solde d'une personne dans le tableau de soldes à partir de son nom ? Quels sont les domaines d'entrée(s) et de sortie(s) de la fonction de reherche d'indice ?

Solution

Pour trouver à quel indice se trouve le solde d'une personne il faut trouver à quel indice se trouve son nom dans le tableau de noms. Il faut donc balayer le tableau jusqu'à trouver le nom recherché et retourner l'indice courant. La fonction prend donc en entrée le tableau de noms (donc de chaînes de caractères), la taille du tableau et le nom à rechercher (donc une chaîne de caractères) et retourne une valeur entière positive.

Attention, il faut gérer le cas où l'on ne trouve pas le nom recherché. C'est une erreur et l'on pourra choisir de retourner un indice impossible, par exemple -1. Cela signifie qu'il faudra gérer cette erreur dans la fonction qui calcule les soldes.

```
find_index_by_name (names, len, name)
  Parcourir names avec i de 0 à len - 1
    Si names[i] est égal à name alors on a trouvé, retourner i
  Retourner -1
```

Q 3.4. À quoi devrait-on faire attention en lisant une ≪ligne de transaction ≫ afin que le programme soit robuste ?

Solution

Pour être plus robuste, il serait intéressant de vérifier que chaque ligne de ≪transaction ≫ est bien formée (avec 2 chaînes et un entier). Dans le cas contraire, ce serait une erreur et il faudrait penser à libérer les ressources allouées (fichier et mémoire).

4 Implémentation

Q 4.1. Écrivez en `C` l'algorithme esquissé dans l'exercice 2 pour vérifier la légalité d'un déplacement. Votre programme prendra ses arguments sur la ligne de commande. Vous pourrez tester votre programme comparant ses résultats avec ceux qui suivent :

```
move_bishop (1, 1, 1, 1)    → false
move_bishop (1, 5, 4, 5)    → false
move_bishop (1, 5, 2, 4)    → true
move_bishop (2, 4, 1, 5)    → true
move_bishop (7, 6, 1, 0)    → true
move_bishop (8, 7, 1, 0)    → false
move_bishop (2, 2, -1, -1)  → false
```

Solution

```c
#include <stdio.h>
#include <stdbool.h>
#include <stdlib.h>

int abs (int i) {
  if (i < 0) return (-i) ;
  return i ;
}


bool move_bishop (int x, int y, int x_prim, int y_prim) {
  if ((x < 0) || (y < 0) || (x > 7) || (y > 7) || (x_prim < 0) ||
      (y_prim < 0) || (x_prim > 7) || (y_prim > 7) ||
      (x == x_prim && y == y_prim))
    return false ;
  return (abs (x - x_prim) == abs (y - y_prim)) ;
}


int main (int argc, char *argv[]) {
  if (argc != 5) {
    printf ("Error. Wrong number of arguments.\n") ;
    return 1 ;
  }

  if (move_bishop
      (atoi (argv[1]), atoi (argv[2]), atoi (argv[3]), atoi (argv[4])))
    printf ("Legal\n") ;
  else printf ("Illegal\n") ;
  return 1 ;
}
```

Manipulation de fichiers en `C`

ATTENTION : Savoir lire dans un fichier est à maîtriser **absolument** car cela resservira dans d'autres TDs et possiblement en **examen**.

— L'ouverture d'un fichier se fait grâce à la fonction

FILE *fopen (**char** *filename, **char** *mode)

qui prend en argument le nom du fichier et une chaîne de caractères représentant le mode d'accès (`"rb"` pour lecture, `"wb"` pour écriture) et renvoie un pointeur sur un « *descripteur* » du fichier (de type `FILE`). Si l'ouverture échoue, `fopen` retourne le pointeur `NULL`.

— La lecture dans un fichier se fait par la fonction

int fscanf (FILE *stream, **char** *format, ...)

qui opère comme `scanf`, mais en lisant dans le fichier désigné par `stream`. Par rapport

à `scanf`, il faut juste passer en plus en premier argument le descripteur du fichier dans lequel lire. Chaque lecture consécutive «avance» automatiquement dans le fichier. Si la lecture échoue (au moins pour un des % du format) `fscanf` retourne la valeur `EOF` (égale à -1). Sinon, elle retourne le nombre d'éléments lus (nombre de % dans le format).

— Pour savoir si l'on a atteint la fin du fichier, il faut interroger la fonction

int feof (FILE *stream)

après lecture. Elle renvoie «vrai» si la fin du fichier a été atteinte, «faux» sinon.

— Finalement, lorsque l'on n'a plus besoin de travailler sur un fichier ouvert, il faut le fermer en utilisant la fonction

int fclose (FILE *stream)

Q 4.2. Écrivez la fonction `find_index_by_name` qui implémente l'algorithme de la question Q3.3.

Solution

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>


int find_index_by_name (char *in_array[], int tlen, char *name) {
  for (int i = 0; i < tlen; i++) {
    if (strcmp (in_array[i], name) == 0) return i ;
  }
  return -1 ;
}
```

Q 4.3. Écrivez la fonction `compute_amount` qui implémente le reste de l'algorithme de l'exercice 3. Vous rajouterez un `main` prenant le nom du fichier de transactions en argument via la ligne de commande.

Solution

```c
void compute_amounts (char *fname) {
  unsigned int nb_persons ;
  char **names ; /* Array of names of persons. */
  /* Temporary strings to fetch names. We could save one but let's keep 3
     instead of 2 for sake of readability. */
  char tmp_name[256], name_debit[256], name_credit[256] ;
  int *balances ;       /* Array of balances. */
  int amount ;     /* Read amount at each iteration in the file. */

  /* Open the file. */
  FILE *file = fopen (fname, "rb") ;
  if (file == NULL) {
    printf ("Error. Unable to open file.\n") ;
    return ;
  }

  /* Get the number of persones. */
  fscanf (file, "%u", &nb_persons) ;
  /* Create the array of persons. */
  names = malloc (nb_persons * sizeof (char*)) ;
  if (names == NULL) {
    printf ("Error. Unable to allocate the array of persons.\n") ;
    fclose (file) ;
```

```
    return ;
  }

  /* Fill the array of persons. */
  for (int i = 0; i < nb_persons; i++) {
    fscanf (file, "%s", tmp_name) ;
    names[i] = malloc ((strlen (tmp_name) + 1) * sizeof (char)) ;
    if (names[i] == NULL) {
      printf ("Error. Unable to allocate a name.\n") ;
      /* Free the names already allocated at this point. */
      for (i = i - 1; i >= 0; i--) free (names[i]) ;
      free (names) ;
      fclose (file) ;
      return ;
    }
    strcpy (names[i], tmp_name) ;
  }

  /* Create an array with a balance of 0 for each person. */
  balances = malloc (nb_persons * sizeof (int)) ;
  if (balances == NULL) {
    printf ("Error. Unable to allocate the array of balances.\n") ;
    for (int i = 0; i < nb_persons; i++) free (names[i]) ;
    free (names) ;
    fclose (file) ;
    return ;
  }
  for (int i = 0; i < nb_persons; i++) balances[i] = 0 ;

  /* Loop on the file until its end... */
  fscanf (file, "%s %s %d", name_debit, name_credit, &amount) ;
  while (!feof (file)) {
    /* Get the index of the person to debit. */
    int index_debit = find_index_by_name (names, nb_persons, name_debit) ;
    /* Get the index of the person to credit. */
    int index_credit = find_index_by_name (names, nb_persons, name_credit) ;
    if ((index_credit == -1) || (index_debit == -1)) {
      printf ("Error. Unable to find a person.\n") ;
      /* TODO : must free arrays and close the file. See how it is done
         at the end of the function. I'm lazy in this solution. */
      return ;
    }
    /* Subtract / add the amount. */
    balances[index_debit] = balances[index_debit] - amount ;
    balances[index_credit] = balances[index_credit] + amount ;
    /* Read next line if any. */
    fscanf (file, "%s %s %d", name_debit, name_credit, &amount) ;
  }
  /* Close the opened file. */
  fclose (file) ;
  /* Summarize the debts/credits. */
  for (int i = 0; i < nb_persons; i++)
    printf ("%s : %d\n", names[i], balances[i]) ;

  /* Free the allocated memory. */
  for (int i = 0; i < nb_persons; i++) free (names[i]) ;
  free (names) ;
  free (balances) ;
}


int main (int argc, char *argv[]) {
  if (argc != 2) {
    printf ("Error. Wrong number of arguments.\n") ;
    return 1 ;
  }

  compute_amounts (argv[1]) ;
  return 0 ;
```

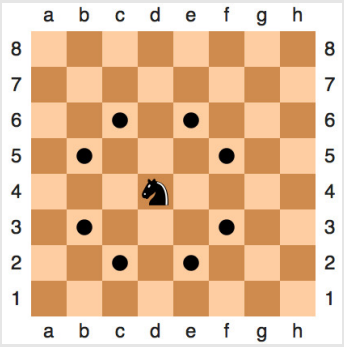
```
}
```

S'il vous reste du temps ou pour continuer après la séance.

5 Cavalier aux échecs

≪Écrivez une fonction qui vérifie si le déplacement d'un cavalier d'une position initiale à une position finale en un coup est conforme aux règles du jeu d'échecs. ≫

Les déplacements possibles d'un cavalier sur un damier d'échec sont rappelés sur l'image ci-dessous (provenance Wikipédia) :



Q 5.1. Quelle sont les relations entre les positions initiales et finales d'un déplacement légal ?

Solution

L'image ci-dessus nous montre qu'il y a 8 déplacements possibles :

$$\begin{array}{ll} x' = x+1 & x' = x+2 \\ y' = y+2 & y' = y+1 \\ y' = y-2 & y' = y-1 \end{array}$$

$$\begin{array}{ll} x' = x-1 & x' = x-2 \\ y' = y+2 & y' = y+1 \\ y' = y-2 & y' = y-1 \end{array}$$

Encore une fois, $x' = x+1$ et $x' = x-1$ se réduisent à $\mid x - x' \mid = 1$ (et similairement pour ± 2 et pour y et y'. Ainsi un déplacement est légal si :

$$\begin{array}{lll} \mid x - x' \mid = 1 & \text{et} & \mid y - y' \mid = 2 \\ & \text{ou} & \\ \mid x - x' \mid = 2 & \text{et} & \mid y - y' \mid = 1 \end{array}$$

Tout comme dans l'exercice précédent, Il faut contraindre la position d'arrivée à rester dans le damier et rejeter le ≪déplacement nul ≫.

Q 5.2. Écrivez en `C` la fonction `move_knight` qui vérifie la légalité d'un déplacement. Vous pourrez tester votre programme comparant ses résultats avec ceux qui suivent :

`move_knight (1, 1, 1, 1)` $\rightarrow$ $false$
`move_knight (4, 5, 3, 7)` $\rightarrow$ $true$
`move_knight (3, 5, 2, 7)` $\rightarrow$ $true$
`move_knight (0, 3, -1, 2)` $\rightarrow$ $false$
`move_knight (4, 6, 5, 6)` $\rightarrow$ $false$
`move_knight (5, 5, 6, 7)` $\rightarrow$ $true$

Solution

```c
#include <stdio.h>
#include <stdbool.h>
#include <stdlib.h>

int abs (int i) {
  if (i < 0) return (-i) ;
  return i ;
}


bool move_knight (int x, int y, int x_prim, int y_prim) {
  if ((x < 0) || (y < 0) || (x > 7) || (y > 7) || (x_prim < 0) ||
      (y_prim < 0) || (x_prim > 7) || (y_prim > 7) ||
      ((x == x_prim) && (y == y_prim)))
    return false ;
  return (((abs (x - x_prim) == 1) && (abs (y - y_prim) == 2))
          ||
          ((abs (x - x_prim) == 2) && (abs (y - y_prim) == 1))) ;
}


int main (int argc, char *argv[]) {
  if (argc != 5) {
    printf ("Error. Wrong number of arguments.\n") ;
    return 1 ;
  }

  if (move_knight
      (atoi (argv[1]), atoi (argv[2]), atoi (argv[3]), atoi (argv[4])))
    printf ("Legal\n") ;
  else printf ("Illegal\n") ;
  return 1 ;
}
```