IN 101 - TD 04 — Solution

1 Magie or not magie

≪Déterminez s'il y a un truc caché (ou une connivence avec la tierce personne) dans le tour de carte que vous venez de voir en l'automatisant. ≫

Dans le tour qui vient de vous être présenté, après le mélange des cartes effectué par votre camarade, après le choix des mouvements de cartes effectué sur chaque pile, celui-ci a réussi à faire ressortir les deux même cartes à la fin.

Pour espérer comprendre comment fonctionne ce tour et au moins savoir s'il y a eu une manipulation qui vous est passée inaperçue, on souhaite le traduire en un programme qui effectue les opérations que vous avez vues : on remplace le magicien par un programme qui sera forcément rationnel et déterministe.

En faisant tourner ce programme avec différentes séquences d'actions, on espère comprendre s'il y a des conditions initiales particulières expliquant quand et comment ce tour de magie fonctionne. Si l'on ne trouve rien de concluant, c'est vraisemblablement que votre chargé de TD a fait une passe cachée qui vous a échappée ou que votre camarade était en fait complice.

Q 1.1. Quelle structure de données choisir pour représenter un paquet de cartes ?

Solution

Un paquet de cartes étant une séquence de cartes, un tableau sera tout à fait adapté. Lorsque le participant désigne une carte dans le paquet, il désigne en fait la valeur contenue à un indice donné dans ce tableau.

On représentera une carte par une valeur entière (par exemple, dans les faits ça n'a aucune espèce d'importance).

Q 1.2. Identifiez les grandes étapes . . .du tour.

Solution

Le tour se découpe en 4 étapes distinctes :

1. La préparation du paquet.
2. Les coupes successives permettant de mélanger le paquet de cartes.
3. La séparation du paquet en 2 sous-paquets (gauche et droite).
4. L'exécution des mouvements dans chaque sous-paquet jusqu'à l'obtention des deux dernières cartes pour vérification d'égalité.

Q 1.3. Quelles sont les traitements de la dernière étape ? Combien de fois sont-ils effectués ?

Solution

À chaque tour il faut :

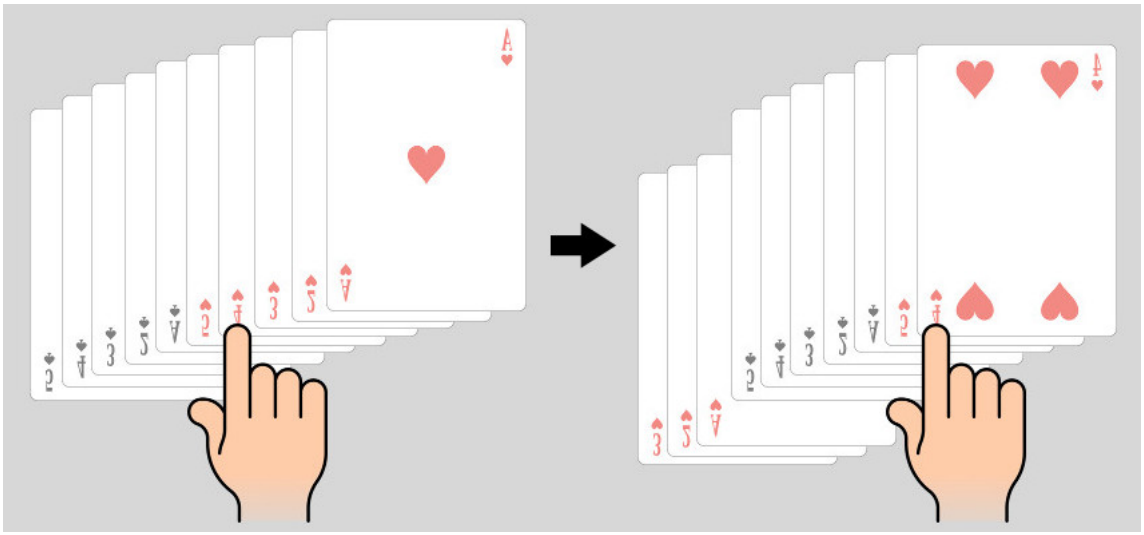
1. Demander le nombre de déplacements à faire pour chaque paquet et vérifier que ce nombre est égal à la taille d'un paquet - 1.
2. Effectuer les déplacements.
3. Supprimer la première carte de chaque paquet.

On itère ce processus « (nombre initial de cartes dans le paquet / 2) - 1 » fois.

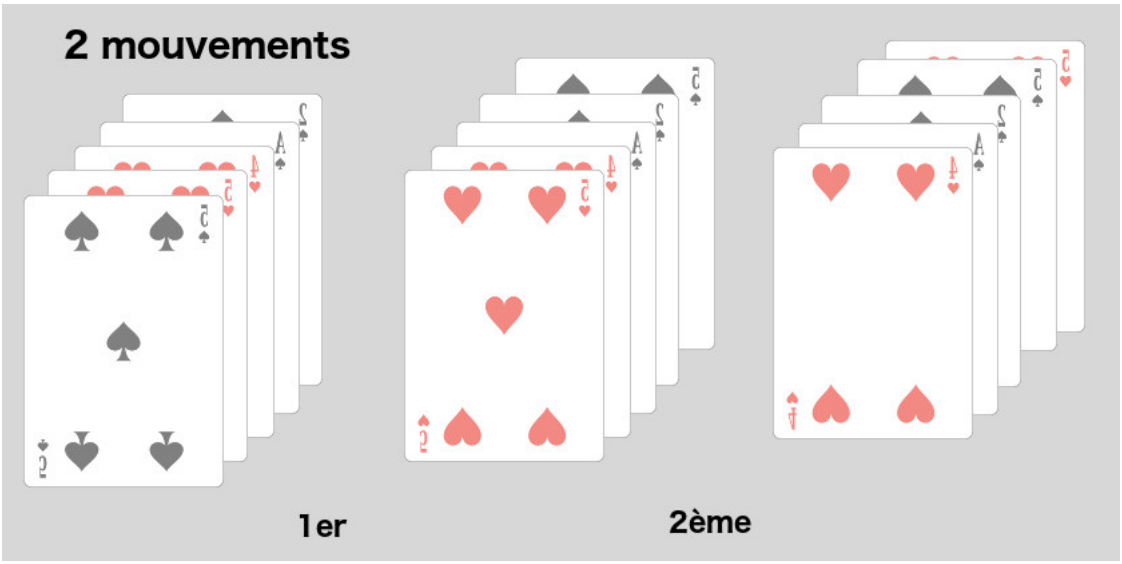
Q 1.4. Comparez les opérations Q1.2-2 et Q1.3-2, que remarquez-vous ?

Solution

En Q1.2-2 on prend toutes les cartes au-dessus de celle montrée pour les mettre en fin de paquet.



En Q1.3-2 on met en fin de paquet un nombre donné de cartes en les prenant sur le dessus.



Ce sont les **mêmes** opérations. En Q1.2-2 on désigne une carte par un index quelconque dans le tableau. En Q1.3-2, c'est le nombre de mouvements qui est directement l'index. Dans les deux cas on coupe le jeu et on remet la partie supérieure du paquet en dessous.

On n'a donc besoin que **d'une seule** fonction pour faire ces 2 traitements.

Q 1.5. De l'algorithme esquissé en Q1.2, de quelles fonctions avez-vous besoin, que font-elles, quels sont leurs domaines ?

Solution

De Q1.2, on déduit que l'on a besoin de :

— 1 fonction `cut` de coupe d'un paquet (passage de cartes du dessus au dessous).
Domaine d'entrée : un tableau d'entiers, un indice dans le tableau et la taille du tableau.
Domaine de sortie : un tableau d'entiers (de même taille que le tableau d'entrée).

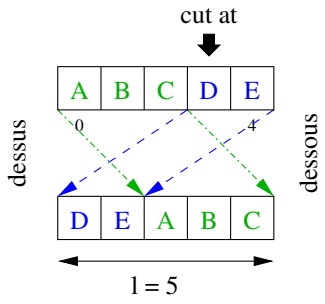
— 1 fonction `split_deck_in_2` pour séparer le paquet initial en 2 paquets.
Domaine d'entrée : un tableau (de taille paire) d'entiers.
Domaine de sortie : 2 tableaux d'entiers. Il va donc falloir soit en retourner un par passage d'argument par adresse, soit se créer une structure de couple. Nous choisissons cette dernière solution.

— 1 fonction `play` pour effectuer les mouvements sur les paquets jusqu'à obtention des 2 cartes finales.
Domaine d'entrée : 2 tableaux d'entiers, le nombre de mouvements autorisés initialement.
Domaine de sortie : néant : affichera si le tour a réussi ou non.

Q 1.6. Quel est l'algorithme de la fonction `cut` issue de la question Q1.5 ?

Solution

Un petit schéma montre qu'il faut ≪tasser≫ la fin du tableau initial, depuis l'endroit où l'on coupe, au début du tableau final. Ensuite, il faut terminer le tableau final en mettant le contenu du tableau initial depuis 0 jusqu'à l'endroit où l'on coupe - 1.

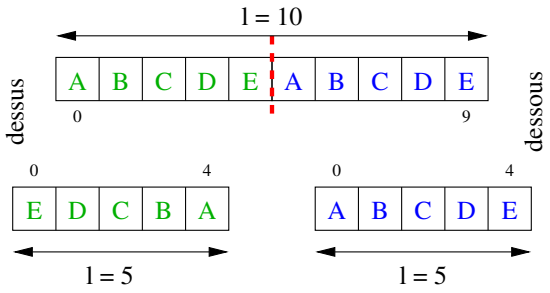



```
cut (deck, at, nb_cards) =
  Créer le tableau résultat (new_deck) de taille nb_cards (vérifier le succès)
  Pour dest de 0 à (nb_cards - at) exclus,
    new_deck[dest] <- deck[at + dest]
  Pour src de 0 à at exclus,
     new_deck[dest] <- deck[src]
     dest <- dest + 1
  Retourner le tableau résultat new_deck
```

Q 1.7. Quel est l'algorithme de la fonction `split_deck_in_2` issue de la question Q1.5 ?

Solution

Si l'on analyse le mouvement des cartes, on remarque que la première moitié du paquet se retrouve inversée dans le sous-paquet de gauche et l'autre moitié se retrouve telle quel dans le sous-paquet de droite.




```
split_deck_in_2 (deck, nb_cards) =
  Créer le tableau left_deck de taille (nb_cards / 2)
  Créer le tableau right_deck de taille (nb_cards / 2)
  Pour i de 0 à (nb_cards / 2) exclus,
    left_deck[i] <- deck[(nb_cards / 2) - 1 - i]  /* Ordre inverse. */
    right_deck[i] <- deck[(nb_cards / 2) + i]     /* Ordre inchangé. */
  Faire un couple des tableaux left_deck et right_deck
  Retourner le couple.
```

Q 1.8. Quelle est la forme de la fonction `remove_first` identifiée en Q1.3, qui permet de supprimer la carte se trouvant au-dessus d'un paquet ? Quels sont ses domaines d'entrée et de sortie ?

Solution

Cette fonction prend en argument un tableau d'entiers de taille non nulle, la taille de ce tableau (puisque la taille d'un tableau en `C` n'est pas contenue dans sa structure) et retourne un tableau d'entier de taille «diminuée de 1 ».

Selon comment l'on écrit la fonction chargée de «jouer » le tour, on peut même contraindre la taille initiale à être supérieure ou égale à 2 car l'on n'appellera jamais cette fonction sur un paquet ne contenant plus qu'une carte (dernière étape du tour où l'on regarde la carte qui reste dans chaque paquet).

```
remove_first (deck, nb_cards) =
  Créer un tableau new_deck de taille (nb_cards - 1)
  Recopier les (nb_cards - 1) dernières valeurs de deck dans new_deck
  Retourner le tableau résultat new_deck
```

Q 1.9. Donnez la forme de la fonction `play` qui prend en argument les 2 paquets de cartes et exécute les mouvements dans chaque sous-paquet jusqu'à l'obtention des deux dernières cartes pour vérification d'égalité.

Solution

Nous pouvons écrire cette fonction de manière récursive ou itérative. Dans cette solution la récursion est choisie.

Cette fonction prend en argument 2 tableaux, la taille de ces tableaux (c'est la même pour les 2) et le nombre total de mouvements restant à effectuer.

Pour se simplifier la tâche et éviter de vérifier que la somme des mouvements sur les 2 paquets est bien égale au nombre autorisé, on ne demande à l'utilisateur que le nombre de mouvements pour le paquet de gauche et l'on en déduit celui pour le paquet de droite.

```
play (deck_left, deck_right, decks_size, nb_moves) =
  Si nb_moves == 0, on a fini et on vérifie si deck_left[0] == deck_right[0]
  Sinon
    Demander combien de mouvements faire sur l'un des paquets --> nb_moves_left
    On peut s'assurer de la validé de ce nombre et recommencer la demande si
        le nombre reçu est incorrect
    Calculer nombre de mouvements de l'autre paquet
         --> nb_moves_right = nb_moves - nb_moves_left
    deck_left2 <- cut (deck_left, nb_moves_left, decks_size)
    deck_right2 <- cut (deck_right, nb_moves_right, decks_size)
    deck_left3 <- remove_first (deck_left2, decks_size)
    deck_right3 <- remove_first (deck_right2, decks_size)
    Récurser avec deck_left3, deck_right3, decks_size - 1 et nb_moves - 1
```

Lors de l'implémentation en `C`, il faudra bien sûr penser à libérer les tableaux intermédiaires.

Q 1.10. Il ne vous reste plus décrire la forme de la fonction ≪principale≫ qui orchestre le tour selon l'analyse que vous avez faite en Q1.2.

Solution

Comme énoncé en Q1.2, il faut préparer le paquet, permettre au spectateur de couper le paquet (autant de fois qu'il souhaite), séparer le paquet initial en 2 sous-paquets, puis enfin exécuter les mouvements sur les paquets.

Par choix, lorsque l'on souhaite arrêter de mélanger le jeu, on entrera un indice de carte invalide dans le paquet (< 0 ou $\geq$ nombre de cartes).

Le paquet initial est le tableau : [1, 2, 3, 4, 5, 1, 2, 3, 4, 5] qui contient 10 cartes. Plutôt que de mettre ≪en dur≫ ce nombre 10 de cartes, on préférera utiliser une variable au cas où l'on voudrait changer le nombre de cartes initial.

```
main () =
  Créer le tableau full_deck de taille DECK_SIZE
  Remplir deck avec les valeurs de 1 à (DECK_SIZE / 2) répétées 2 fois
  Demander où couper --> cut_at
  Tant que cut_at est valide
    Couper le tableau représentant le paquet à cut_at
    Demander où couper --> cut_at
  Couper le paquet en 2 sous-paquets
  Appeler play avec ces 2 sous-paquets de taille (DECK_SIZE / 2) et avec
       (DECK_SIZE / 2) - 1 mouvements restant à faire
```

Bien entendu, lors de l'implémenation en `C`, il faudra penser à vérifier le succès des allocations et libérer la mémoire devenue inutile.

2 Implémentation

Q 2.1. Implémentez en `C` toutes les fonctions identifiées et dont vous avez esquissé les algorithmes.

Solution

Il faut bien faire attention de vérifier que les allocations mémoire réussissent. En cas d'échec, on émet un message d'erreur et l'on interrompt directement le programme avec la fonction `exit`. Notons que l'on ne prend pas la peine de chercher à libérer la mémoire dans ce cas. En fait, ce n'est pas très grave car la fin du programme la libérera de fait. Néanmoins, ce n'est pas une habitude très saine que de ne pas proprement restituer les ressources avant la fin d'un programme. On prend cette liberté ici pour ne pas rallonger cette correction.

`cut`

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

/* Cut the deck. Put everything above 'at' below the deck. In a deck, the upper
   card is at index 0, deeper cards are a increasing indices.
   [1, 2, 3, 4, 5, 1, 2, 3, 4, 5]
          ^
   Cut a index 2 take everything before the cell of index 2 end put it at the
   end of the deck (array).
   [3, 4, 5, 1, 2, 3, 4, 5, 1, 2] */
int* cut (int *deck, int at, unsigned int nb_cards) {
  if ((at >= 0) && (at < nb_cards)) {
    int *new_deck = malloc (nb_cards * sizeof (int)) ;
    if (new_deck == NULL) {
      printf ("Error. cut. No memory.\n") ;
      exit (1) ;
    }

    int dest = 0 ;
    for (; dest < nb_cards - at; dest++) new_deck[dest] = deck[at + dest] ;
    for (int src = 0; src < at; src++) {
      new_deck[dest] = deck[src] ;
      dest++ ;
    }
    return new_deck ;
  }
  printf ("Error. cut. Invalid position.\n") ;
  exit (1) ;
}
```

Pour retourner les 2 sous-paquets, nous définissons une `struct deck_pair_t`. Nous avons fait le choix que la fonction `split_deck_in_2` alloue dynamiquement une telle structure pour ensuite la retourner. Elle aurait pu le faire sans allocation dynamique pour la `struct` en elle-même. Néanmoins, quelle que soit la solution choisie, elle doit ensuite allouer **dynamiquement** les deux tableaux à mémoriser dans cette structure pour qu'ils puissent ≪survivre≫ au retour de la fonction.

`split_deck_in_2`

```
/* Pair of sub-decks obtained after splitting the initial deck in two parts. */
struct deck_pair_t {
  int *left_deck ;
  int *right_deck ;
};


/* Split a deck (assumed to contain an even number of cards) into 2 subdecks.
   The left deck contains the the first half of cards in reverse order.
   The right deck contains the last half of cards without changing their order.
   Deck [1, 2, 3, 4, 5, 1, 2, 3, 4, 5] =>
      left deck = [5, 4, 2, 2, 1]
      right deck = [1, 2, 3, 4, 5] */
```

```c
struct deck_pair_t* split_deck_in_2 (int *deck, unsigned int nb_cards) {
  if (nb_cards % 2 != 0) return NULL ;
  int sub_deck_len = nb_cards / 2 ;
  struct deck_pair_t *decks = malloc (sizeof (struct deck_pair_t)) ;
  if (decks == NULL) {
    printf ("Error. split_deck_in_2. No memory.\n") ;
    exit (1) ;
  }
  decks->left_deck = malloc (sub_deck_len * sizeof (int)) ;
  if (decks->left_deck == NULL) {
    printf ("Error. split_deck_in_2. No memory.\n") ;
    free (decks) ;
    exit (1) ;
  }
  decks->right_deck = malloc (sub_deck_len * sizeof (int)) ;
  if (decks->right_deck == NULL) {
    printf ("Error. split_deck_in_2. No memory.\n") ;
    free (decks->left_deck) ;
    free (decks) ;
    exit (1) ;
  }
  /* Left deck contains the first half of cards in reverse order ! */
  /* Right deck contains the last half of cards without changing their order. */
  for (int i = 0; i < sub_deck_len; i++) {
    decks->left_deck[i] = deck[sub_deck_len - 1 - i] ;
    decks->right_deck[i] = deck[sub_deck_len + i] ;
  }
  return (decks) ;
}
```

`remove_first`

```c
/* Remove the first card of a deck assumed to contain at least 2 cards. Return
   the new deck whose size is initial deck's size - 1. */
int* remove_first (int *deck, unsigned int nb_cards) {
  int new_size = nb_cards - 1 ;
  if (new_size == 0) {
    printf ("Error. remove_first. Null new array size.\n") ;
    exit (1) ;
  }
  int *new_deck = malloc (new_size * sizeof (int)) ;
  if (new_deck == NULL) {
    printf ("Error. remove_first. No memory.\n") ;
    exit (1) ;
  }
  for (int i = 0; i < new_size; i++) new_deck[i] = deck[i + 1] ;
  return new_deck ;
}
```

`play` Nous avons vu que cette fonction enchaîne 2 étapes : `cut` puis `remove_first` (pour chaque sous-paquet). Or chacune de ces fonctions retourne un nouveau tableau alloué dynamiquement. Les 2 tableaux intermédiaires issus de `cut` deviennent inutiles après avoir obtenu ceux retournés par `remove_first`. Il faut donc penser à les libérer.

```c
/* Play the scenario of asking how many moves on each deck.
   One remarks that the "move" operation on each deck is exactly the same
   than the "shuffle" of the beginning.
   Indeed, the argument [decks_size] is useless since it is always equal to
   [nb_moves]. */
void play (int *deck_left, int *deck_right, unsigned int decks_size,
           unsigned int nb_moves) {
  if (nb_moves == 0) {
    printf ("%d -- %d\n", deck_left[0], deck_right[0]) ;
```

```
    if (deck_left[0] == deck_right[0]) printf ("Correct\n") ;
    else printf ("Lost\n") ;
   }
  else {
    int nb_moves_left ;
    printf ("%d moves allowed.\n#moves ont left deck? ", nb_moves) ;
    scanf ("%d", &nb_moves_left) ;
    while ((nb_moves_left < 0) || (nb_moves_left > nb_moves)) {
      printf ("Bad number of moves.\n#moves on left deck? ") ;
      scanf ("%d", &nb_moves_left) ;
    }

    unsigned int nb_moves_right = nb_moves - nb_moves_left ;
    /* Make the moves. */
    int *deck_left2 = cut (deck_left, nb_moves_left, decks_size) ;
    int *deck_right2 = cut (deck_right, nb_moves_right, decks_size) ;
    /* Suppress the top card. */
    int *deck_left3 = remove_first (deck_left2, decks_size) ;
    int *deck_right3 = remove_first (deck_right2, decks_size) ;
    /* Free no more used decks. */
    free (deck_left2) ;
    free (deck_right2) ;
    play (deck_left3, deck_right3, decks_size - 1, nb_moves - 1) ;
  }
```

Nous pourrions être tenté d'allouer le paquet initial statiquement. Cela n'est pas une bonne idée car lorsque nous sommes dans la partie des ≪coupes≫, la fonction `cut` nous renvoie un nouveau tableau alloué dynamiquement. Nous devons alors libérer celui de la précédente coupe. Or, après la première coupe, nous nous retrouverions à tenter de libérer un tableau alloué statiquement (plantage garanti). Il faudrait alors faire un cas particulier ce qui compliquerait inutilement l'algorithme.

`main`

```
/* Size = 10 --> deck = { 1, 2, 3, 4, 5, 1, 2, 3, 4, 5 }. */
#define DECK_SIZE (10)


/* Finally create the initial deck, ask the user to cut where he wants, how
   long he wants, then ask and execute the moves and check that the 2 ending
   cards are the same. */
int main () {
  int cut_at ;
  int *full_deck = malloc (DECK_SIZE * sizeof (int)) ;
  if (full_deck == NULL) {
    printf ("Error. main. No memory.\n") ;
    return 1 ;
  }
  for (unsigned int i = 0; i < (DECK_SIZE / 2); i++) {
    full_deck[i] = i + 1 ;
    full_deck[i + (DECK_SIZE / 2)] = i + 1 ;
  }

  printf ("Where to cut? ") ;
  scanf ("%d", &cut_at) ;
  while ((cut_at >= 0) && (cut_at < DECK_SIZE)) {
    int *new_deck = cut (full_deck, cut_at, DECK_SIZE) ;
    /* TODO: check success of the allocation. */
    free (full_deck) ;
    full_deck = new_deck ;
    printf ("Where to cut? ") ;
    scanf ("%d", &cut_at) ;
  }
  /* Some debug the spectator should never see ^^ */
  printf ("Deck is now:") ;
```

```
for (unsigned int i = 0; i < DECK_SIZE; i++) printf (" %d", full_deck[i]) ;
printf ("\n") ;

struct deck_pair_t *decks = split_deck_in_2 (full_deck, DECK_SIZE) ;
free (full_deck) ;
play (decks->left_deck, decks->right_deck,
      (DECK_SIZE / 2), (DECK_SIZE / 2) - 1) ;
free (decks->left_deck) ;
free (decks->right_deck) ;
free (decks) ;

return 0 ;
```

3 Conclusion

Q 3.1. Alors, conclusion ?

Solution

Ce tour fonctionne toujours : il est le résultat d'un invariant mathématique qui se retrouve évidemment dans toutes les étapes de l'algorithme du programme.

Au début du tour, les cartes sont ordonnées : 1, 2, 3, 4, 5, 1, 2, 3, 4, 5. Le fait de couper le jeu autant de fois que l'on veut ne change pas cet **ordre cyclique modulo le nombre de cartes**. Les coupes ne font que des **permutations circulaires** : l'ordre relatif entre les cartes (toujours modulo) reste le même.

La coupe du jeu en 2 sous-paquets égaux rend les deux piles ≪palindromiques≫ (ie. miroir l'une de l'autre). Cela signifie que si l'on regarde les deux piles l et r, toutes deux de longueur n, on a :

$$\forall i \in [0; n[,\ l[i] = r[n-1-i]$$

À chaque ≪tour de mouvements≫, l'utilisateur a $n-1$ mouvements à disposition. Soit m_l et m_r les nombres de mouvements décidés par l'utilisateur, pour les paquets l et r. On a :

$$\forall m_l\ m_r,\ m_l + m_r = n-1$$

donc $m_r = n - 1 - m_l$.

On se pose la question, est-ce que $l[m_l] = r[m_r]$?
Donc, est-ce que $l[m_l] = r[n-1-m_l]$?
La réponse est **oui, toujours** puisque c'est exactement la propriété de piles palindromiques énoncée précédemment.

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

4 Stop aux allocations dynamiques

Dans l'exercice précédent, chaque manipulation sur les piles de cartes donnait naissance à un ou des nouveaux tableaux. Nous avons donc fait de nombreuses allocations dynamiques. Pour autant, on peut se convaincre aisément que toutes ces manipulations auraient pu être faites au

sein du tableau initial.

Le but de cet exercice est de reprendre le programme précédent et de ne plus créer de nouveaux (sous)-tableaux.

Q 4.1. Comment peut-on réécrire la fonction `cut` afin qu'elle effectue ses modifications directement dans le tableau reçu en argument ? Que deviennent ses domaines d'entrée et de sortie ?

Solution

On peut peut-être déjà répondre à la seconde question. Puisqu'elle effectue les mêmes opérations, elle va prendre les mêmes arguments.

On remarque que cette fonction ne change pas la taille du tableau : le tableau résultat commencera et terminera donc aux mêmes endroits que le tableau reçu en argument. Donc, elle n'a rien de nouveau à retourner, l'adresse du tableau résultat étant déjà connue par la fonction appelante (eh oui, c'est l'argument avec lequel elle fait l'appel). Donc cette fonction ne retournera plus rien au sens C (`void`).

Nous devons faire les mêmes déplacements que ceux identifiés en question 1.6. Malheureusement, les zones source et destination des copies **se chevauchent**. Et une simple variable temporaire ne suffit pas pour mémoriser la valeur d'une case avant de l'écraser (comme c'est le cas lorsque l'on inverse l'ordre des éléments d'un tableau).

Nous allons donc devoir utiliser un tableau intermédiaire (comme dans la version initiale) que l'on recopiera dans le tableau initial à la fin, puis que l'on libérera avant de quitter la fonction.

Nous mettons en fond grisé les parties de l'algorithme qui restent inchangées par rapport à la version initiale.

```
(void) cut (deck, at, nb_cards) =
  Créer le tableau intermédiaire (tmp_deck) de taille nb_cards (vérifier le succès)
  Pour dest de 0 à (nb_cards - at) exclus,
     tmp_deck[dest] <- deck[at + dest]
  Pour src de 0 à at exclus,
     tmp_deck[dest] <- deck[src]
     dest <- dest + 1
  Pour i de 0 à nb_cards exclus,
     deck[i] <- tmp_deck[i]
  Libérer tmp_deck
```

Q 4.2. C'est bien entendu maintenant au tour de `split_deck_in_2` de subir le même sort. Quels sont ses domaines d'entrée et de sortie désormais ? Puis quel est son nouvel algorithme ?

Solution

Cette fonction va prendre les mêmes arguments puisqu'elle effectue le même travail. Elle doit toujours nous retourner les deux sous-tableaux. Donc ses domaines d'entrée et de sortie n'ont pas de raison de changer.

Néanmoins, ce qui va changer c'est que les deux pointeurs représentant les sous-tableaux gauche et droite ne vont plus désigner des tableaux « nouveaux », mais des endroits dans le tableau reçu en argument. Invariablement, le sous-tableau de gauche correspondra à l'adresse du tableau initial. Et le sous-tableau de droite correspondra à l'adresse de la case au milieu du tableau initial.

Comme nous l'avons vu en question 1.7, la partie droite doit contenir les éléments initialement à droite, mais en ordre inverse. Pour la partie gauche, c'est gratuit puisque l'ordre reste inchangé.

```
split_deck_in_2 (deck, nb_cards) =
  Pour i de 0 à ((nb_cards / 2) / 2) exclus,      /* Ordre inverse. */
    tmp <-  deck[i]
    deck[i] <- deck[(nb_cards / 2) - 1 - i]
    deck[(nb_cards / 2) - 1 - i] = tmp
  left_deck <- deck                               /* Ordre inchangé. */
  right_deck <- adresse de deck[nb_cards / 2]
  Faire un couple des pointeurs left_deck et right_deck
  Retourner le couple
```

Notons au passage, pour la future implantation en C, que l'on pourrait tout à fait de pas allouer dynamiquement la structure de couple puisqu'il est possible de retourner une `struct` en C.

Q 4.3. Que devient la fonction `remove_first` de la question 1.8 ?

Solution

Cette fonction se contentait de retourner le tableau argument privé de sa première case. Il lui suffit donc de juste retourner l'adresse de sa seconde case.

```
remove_first (deck, nb_cards) =
  Retourner l'adresse de deck[1]
```

Q 4.4. Que devient la fonction `play` de la question 1.9 ?

Solution

La seule différence est qu'il n'y a plus besoin de tableaux intermédiaires, donc plus besoin d'en libérer. De plus, `cut` effectuant désormais ses modifications en place, elle ne retourne plus rien, donc pas besoin de mémoriser son résultat.

```
play (deck_left, deck_right, decks_size, nb_moves) =
  Si nb_moves == 0, on a fini et on vérifie si deck_left[0] == deck_right[0]
  Sinon
    Demander combien de mouvements faire sur l'un des paquets --> nb_moves_left
    On peut s'assurer de la validé de ce nombre et recommencer la demande si
        le nombre reçu est incorrect
    Calculer nombre de mouvements de l'autre paquet
         --> nb_moves_right = nb_moves - nb_moves_left
    cut (deck_left, nb_moves_left, decks_size)
    cut (deck_right, nb_moves_right, decks_size)
    deck_left2 <- remove_first (deck_left, decks_size)
    deck_right2 <- remove_first (deck_right, decks_size)
    Récurser avec deck_left2, deck_right2, decks_size - 1 et nb_moves - 1
```

Q 4.5. Que devient la fonction `play` de la question 1.9 ?

Solution

Dans le principe, elle ne change quasiment pas. La forme de son algorithme reste identique, néanmoins l'implantation en C n'a plus besoin de libérer les tableaux intermédiaires anciennement retournés par `cut` et `split_deck_in_2`.

```
...
while ((cut_at >= 0) && (cut_at < DECK_SIZE)) {
  int *new_deck = cut (full_deck, cut_at, DECK_SIZE) ;
  free (full_deck) ;                    /* (*) */
  full_deck = new_deck ;
  printf ("Where to cut? ") ;
  scanf ("%d", &cut_at) ;
}

struct deck_pair_t *decks = split_deck_in_2 (full_deck, DECK_SIZE) ;
free (full_deck) ;
play (decks->left_deck, decks->right_deck,
      (DECK_SIZE / 2), (DECK_SIZE / 2) - 1) ;
free (decks->left_deck) ;
free (decks->right_deck) ;
free (full_deck) ;                      /* <--- (*) Déplacé ici. */
free (decks) ;
return 0 ;
```

Il est même possible de supprimer l'allocation dynamique du tableau initial `full_deck`. En effet, vu que dans cet exercice il est de taille fixe, on peut l'allouer statiquement. On remplace le `malloc` par :

int full_deck [DECK_SIZE] ;

et l'on supprime le

free (full_deck) ;

final.

Le code complet de la solution n'est pas inclu dans ce PDF mais il peut être consulté dans l'archive de correction qui vous est fournie (fichier `magic-opt.c`).