Devoir d'Informatique n$^{\text{O}}$ 1

1$^{\text{ère}}$ année

Adeline Darmon
Mathieu Goutelle
Groupe de TD 11

10 mars 2002

Texte mis en forme avec LaTeX 2$_\varepsilon$

1 Présentation du jeu

Il s'agit d'un jeu à deux joueurs identifiés dans la suite joueur A et joueur B. Le joueur A forme des tas d'allumettes. Les joueurs enlèvent alors alternativement, en commençant par le joueur B, autant d'allumettes qu'ils le veulent (au moins une) mais seulement dans un tas. Le gagnant est celui qui enlève la dernière allumette.

2 Analyse du problème

Pour jouer, on convertit chaque nombre d'allumettes en binaire et on ajoute *bit* à *bit* les nombres ainsi convertis. On obtient ainsi quatre nombres (nombre d'allumettes inférieurs à 10, donc codés sur quatre bits) dont on ne conserve que le reste de la division entière par 2 pour tester leur parité.

La stratégie définie dans le sujet indique que le joueur B doit toujours s'efforcer de rendre toutes les sommes paires, *i.e.* tous les nombres définis précédemment égaux à zéro. S'il n'y a pas d'imparité, on sera obligé de choisir un tas et un nombre d'allumettes au hasard car l'on est obligé de créer une imparité en jouant. Au contraire, s'il existe une somme impaire, on devra choisir un tas et enlever des allumettes de telle sorte que des 1 et/ou des 0 apparaissent ou disparaissent selon les besoins. Ainsi, si l'on définit les sommes comme :

$$S = [s_3; s_2; s_1; s_0]$$

il faudra agir sur le *bit* i du nombre d'allumettes du tas choisi si $s_i = 1$ pour $0 \leq i \leq 3$.

Le choix du tas est réalisé ainsi : il faut que la somme impaire de « poids le plus fort » corresponde à un 1 dans la décomposition binaire du nombre d'allumettes pour que l'on puisse supprimer cette première imparité en enlevant des allumettes. On choisit donc le premier tas satisfaisant à cette condition.

Alors, si s_i est le rang non nul de poids le plus fort, on peut donc commencer à soustraire 2^i à n et la nouvelle valeur de n est $n - 2^i$. On supprime ainsi la première imparité. Ensuite, si l'on veut agir sur le *bit* j ($j \leq i$), on peut soit ajouter soit soustraire 2^j. On peut en effet ajouter des allumettes compte tenu du fait que l'on a déjà soustrait un nombre plus grand d'allumettes (2^i) et car :

$$2^i > \sum_{j=0}^{i-1} 2^j = 2^i - 1$$

On peut donc ajouter toutes les puissances de 2 de rang inférieur à i si l'on a soustrait 2^i : cela revient à enlever une seule allumette.

Alors, supprimer l'imparité s_j revient à prendre le complément du *bit* j de n : en effet, si le *bit* j de n vaut 1, on soustrait 2^j et il dévient égal à 0. S'il vaut 0, alors en ajoutant 2^j, on fait apparaître un 1. En répétant cette méthode pour tous les *bits* non nuls de S, on supprime toutes les imparités puisque chacune des actions précédentes est indépendante car elles agissent sur des *bits* différents du nombre d'allumettes n du tas choisi.

On possède alors une méthode qui permet au joueur B de jouer face au joueur A en rendant chaque fois que cela est possible les sommes impaires et ainsi de développer une stratégie — presque toujours — gagnante.

3 Algorithme

L'algorithme va mettre en œuvre la méthode décrite plus haut (section 2). On ne donnera pas, dans cette section, les détails de certaines fonctions (initialisation, affichage de l'état du jeu, jeu du joueur A...) qui sont relativement dépendantes du langage de

programmation — et aussi du programmeur. Nous nous restreindrons au détail du jeu de B et à la fonction principale de jeu qui gère l'enchaînement des différents procédures.

Dans les algorithmes suivants, l'entier `nbre_tas` *contient le nombre de tas au départ du jeu et le tableau* `jeu` *contient l'état des tas,* i.e. *le nombre d'allumettes qu'ils contiennent. Ces deux variables seront initialisées au début du jeu. Les commentaires apparaissent ici entre accolades ({...}). On utilisera aussi une fonction* `aleat(n)` *qui renvoie un entier naturel aléatoire entre* 0 *et* $n-1$.

La procédure `traite` (cf. ALGORITHME 1) va, les parités des sommes étant données en paramètre (ligne 1), soit jouer au hasard s'il n'y a pas d'imparité, soit modifier l'état du jeu afin de rendre toutes les sommes paires. On cherche d'abord le nombre d'imparités (lignes 5-8) en partant du *bit* de poids le plus fort et on le mémorise (ligne 9). Ensuite, le premier cas (lignes 11-15) est relativement simple. Il suffit de tirer deux nombres aléatoirement : le premier est le numéro du tas (lignes 12-14), en s'assurant que celui-ci n'est pas vide, et le deuxième le nombre d'allumettes à enlever (ligne 15), en s'assurant que l'on enlève au moins une allumette et au maximum le nombre d'allumettes du tas.

Dans le second cas (lignes 16-30), on commence d'abord par choisir un tas comme décrit dans la section 2. Pour cela, on parcourt le tableau `jeu` : si la valeur du tableau correspond à un nombre d'allumettes possédant le *bit* de rang *sauv* à 1 (ligne 19). Le test de la ligne 19 s'explique ainsi : n s'écrit en binaire $n = \sum_i a_i 2^i$. Pour connaître la valeur de a_j, on divise par 2^i :

$$n = 2^j(\sum_{i \geq j} a_i 2^i) + r \quad \text{où} \quad r = \sum_{i<j} a_i 2^i$$

Donc en calculant le quotient de la division entière et en prenant le reste *modulo* 2, on obtient la valeur de a_j. Enfin, le numéro du tas choisi est enregistré dans la variable `rang` et on sort de la boucle.

Le tas étant choisi (ligne 20) on sort de la boucle et on peut appliquer exactement la méthode de modification décrite dans la section 2 : Pour toutes les sommes, si le *bit* courant du nombre d'allumettes vaut 1, on le met à 0 en soustrayant la puissance de 2 correspondante et sinon on ajoute cette puissance de 2 pour le mettre à 0.

Le joueur B (cf. ALGORITHME 2) commence avant tout par calculer les sommes colonne par colonne des écritures binaires des valeurs des tas (cf. section 2). Pour cela, il n'est pas nécessaire de stocker toutes les décompositions binaires des nombres d'allumettes. On ajoute dans un tableau les valeurs directement au tableau `somme` (ligne 8) au fur et à mesure que l'on calcule les *bits* du nombre d'allumettes (lignes 8-9). Ensuite, on appelle la procédure `traite` précédemment définie en passant comme paramètre les sommes.

La fonction `est_fini`(cf. ALGORITHME 3) parcourt l'ensemble des tas et calcule le total du nombre d'allumettes. Si celui-ci est nul, *i.e.* que tous les tas sont vides, la fonction renvoie 1 (le jeu est fini) et sinon, elle renvoie 0.

Pour jouer (cf. ALGORITHME 4), on initialise le jeu (procédure non présentée dans cette section, voir les détails de programmation à la section 4) et on l'affiche avant de débuter. Ensuite, on enchaîne dans une boucle infinie les différentes fonctions définies auparavant (sauf pour l'affichage du jeu et le jeu du joueur A, voir les détails de programmation à la section 4) : Le joueur B joue et on présente le jeu après (lignes 6-7). Si le jeu est fini, on arrête (lignes 8-11). Sinon (lignes 13-17), on fait jouer le joueur A, on met à jour l'écran et on teste si le jeu est fini : si c'est le cas, on sort de la boucle (lignes 15-16) et sinon on recommence au début.

Algorithme 1 Procédure `traite`

```
    Procédure traite(ES jeu : tableau[0..9] d'entiers, E nbre_tas : entier,
                     E somme : tableau[0..3] d'entiers)
 2: i, rang : entiers;

 4: i ← 3
    Tantque (somme[i] = 0 et i ≥ 0) Faire
 6:   i ← i + 1
    Fin Tantque
 8: sauv ← i

10: Si (sauv=-1) Alors {toutes les sommes sont paires}
      Répète
12:     rang ← aleat(nbre_tas)
      Jusqu'à (jeu[rang] ≠ 0)
14:   i ← aleat(jeu[rang]) + 1
    Sinon {il existe une somme impaire}
16:   i ← 0; rang ← −1
      Tantque (i < nbre_tas et rang < 0) Faire
18:     Si (jeu[i]/2^sauv est pair) Alors
          rang ← i
20:     Fin Si
        i ← i + 1
22:   Fin Tantque

24:   Pour i = 3 à 0 Faire
        Si (jeu[rang]/2^i est pair) Alors
26:       jeu[rang] ← jeu[i] − somme[i] ∗ 2^i
        Sinon
28:       jeu[rang] ← jeu[i] + somme[i] ∗ 2^i
        Fin Si
30:   Fin Pour
    Fin Si
```

Algorithme 2 Procédure `joueurB`

```
    Procédure joueurB(ES jeu : tableau[0..9] d'entiers, E nbre_tas : entier)
2:  i, j, n : entiers;
    somme : tableau[0..3] d'entiers = [0,0,0,0] {On initialise le tableau à zéro}
4:
    Pour i = 0 à nbre_tas Faire
6:      n ← jeu[i]
        Pour j = 0 à 3 Faire
8:          somme[j] ← (somme[j] + n mod 2) mod 2
            n ← n/2 {division entière}
10:     Fin Pour
    Fin Pour
12:
    traite(jeu, nbre_tas, somme)
```

Algorithme 3 Fonction `est_fini`

```
    Fonction est_fini(E jeu : tableau[0..9] d'entiers, E nbre_tas : entier) : entier
2:  total, i : entiers

4:  total ← 0
    Pour i = 0 à nbre_tas Faire
6:      total ← total + jeu[i]
    Fin Pour
8:
    Si (total = 0) Alors
10:     Retourner(1)
    Sinon {il existe un tas non vide}
12:     Retourner(0)
    Fin Si
```

Algorithme 4 Procédure `jouer`

```
    Procédure jouer(ES jeu : tableau[0..9] d'entiers, ES nbre_tas : entier)
 2: init(jeu, nbre_tas) {Initialisation}
    affiche(jeu, nbre_tas) {Affichage de l'état du jeu}
 4:
    Boucle {Boucle infinie}
 6:   joueurB(jeu, nbre_tas)
      affiche(jeu, nbre_tas) {Affichage de l'état du jeu}
 8:   Si est_fini = 0 Alors
        Afficher("J'ai gagné !")
10:     Sortir {Sortie de la boucle}
      Fin Si
12:
      joueurA(jeu, nbre_tas) {jeu du joueur A}
14:   affiche(jeu, nbre_tas) {Affichage de l'état du jeu}
      Si est_fini = 0 Alors
16:     Afficher("Bravo ! Vous avez gagné !")
        Sortir {Sortie de la boucle}
18:   Fin Si

20:   Faire une pause
    Fin Boucle
```

4 Source du programme C

Nous avons pris ici le parti de ne pas commenter les sources des programmes afin de ne pas les surcharger. Les remarques que nous pouvons avoir à faire apparaissent dans les paragraphes suivant le programme. Ce programme a été compilé avec le compilateur C DJGPP *(norme* ISO / ANSI*) : il peut donc apparaître des différences par rapport au compilateur* Microsoft Visual C++ *présent sur les machines mises à disposition, notamment au niveau des bibliothèques.*

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

const puiss2 [4] = {1, 2, 4, 8};

int nbre_tas = 10;
int jeu [10];
```

Ces quelques lignes composent l'en-tête du programme et permettent de charger les bibliothèques (lignes 1-3) pour accéder à des fonctions non prédéfinies :

- la bibliothèque `stdio.h` pour les fonctions d'entrées-sorties de base (`printf`, `scanf`...) ;
- la bibliothèque `stdlib.h` pour les fonctions de tirages de nombres aléatoires (`srand` et `rand`) ;
- la bibliothèque `conio.h` pour les fonctions `kbhit` et `clrscr` ;
- la bibliothèque `time.h` pour l'initialisation du générateur de nombres aléatoires avec l'heure.

On trouve aussi une déclaration de constante contenant les trois premières puissances de 2 (ligne 5) et de deux variables globales (lignes 7-8) : le nombre de tas et un tableau pour stocker l'état du jeu.

```
int  init () {
    int  i;

    srand(time(0));

     clrscr ();
     printf("\n                                    Le jeu de NIM\n");
     printf("                                    ------------\n\n\n");
     printf("Entrer le nombre de tas (entre 1 et 10) : ");
     scanf("%d", &nbre_tas);
     if  (nbre_tas > 10     nbre_tas < 1) return(1);
     printf("Remplir les tas avec (nombre entre 1 et 10) : ");
     for  (i = 0; i < nbre_tas; i++) {
         scanf("%d", jeu + i);
         if  (jeu[i] > 10     jeu[i] < 0) return(1);
     };
     return(0);
}
```

La fonction `init` renvoie 0 si l'initialisation s'est déroulée correctement et 1 sinon. La ligne 4 permet d'initier la séquence de nombres aléatoires par une « graine » (*seed* en anglais) avec l'heure courante du système. Après quelques instructions consacrées à la présentation (lignes 6-8), on demande à l'utilisateur le nombre de tas : si celui-ci est supérieur à 10 ou négatif, la fonction se termine prématurément et renvoie 1. Ensuite, le

nombre d'allumettes pour chacun des tas est demandé et de la même façon, si ce nombre n'est pas compris entre 1 et 10, la fonction renvoie 1. Si tout s'est bien passé, 0 est retourné.

```
void joueurB() {
2   void traite(int *somme);

4   int i, j, n;
    int somm[4] = {0, 0, 0, 0};
6
    for (i = 0; i < nbre_tas; i++) {
8       n = jeu[i];
        for (j = 0; j <= 3; j++) {
10          somm[j] ^= n%2;
            n = n/2;
12      }
    }
14  for (i = 0; i <= 3; i++) somm[i] *= puiss2[i];

16  traite(somm);
}
```

La fonction `traite` est définie ici (ligne 2) pour plus de clarté : elle fait jouer le joueur B comme décrit à la section 2 en fonction des valeurs de la somme et implémente l'algorithme défini à la section 3.

Dans cette procédure (`joueurB`), on calcule les sommes des différentes valeurs colonne par colonne des nombres d'allumettes dans chaque tas. Pour cela, on définit un tableau de quatre éléments qui contiendra le résultat. Ensuite, on parcourt l'ensemble des tas (lignes 7-13) et on convertit la valeur du tas (lignes 9-11) en binaire : on ajoute chaque *bit* à la valeur de la somme déjà calculée grâce à l'opérateur ^ (ou exclusif) car seule la parité nous importe. En réalité, la ligne 10 est équivalente à : somm[j] = (somm[j] + n%2)%2;. Après avoir calculé la somme, on multiplie chaque *bit* par la puissance de 2 correspondante pour des questions de facilité (cf. procédure `traite` ci-après). On appelle ensuite la procédure `traite`.

```
void traite(int *somme) {
2   int rang, i, sauv;

4   i = 3;
    while (somme[i] == 0 && i >= 0) i--;
6   sauv = i;

8   if (sauv == -1) {
      do
10      rang = rand()%nbre_tas;
      while (jeu[rang] == 0);
12    i = rand()%jeu[rang] + 1;
      jeu[rang] -= i; }
14  else {
      i = 0; rang = -1;
16    while (i < nbre_tas && rang < 0) {
          if ((jeu[i]/puiss2[sauv])%2) rang = j;
18        i++;
      }
20
      for (i = 3; i >= 0; i--)
22      if ((jeu[rang]/puiss2[i])%2)
```

```
        jeu[rang] −= somme[i];
24      else
        jeu[rang] += somme[i];
26    };
}
```

Cette procédure (`traite`) est le « cœur » du programme : c'est elle qui détermine le jeu du joueur B. Elle prend en argument un tableau contenant les sommes calculées par la procédure précédente.

On détermine en premier lieu s'il y a des sommes paires (ligne 6) et on sauve le rang de la somme impaire de poids le plus fort. Si elles sont toutes paires, on choisit un tas et un nombre d'allumettes au hasard et on joue. Sinon, on met en œuvre l'algorithme de choix du tas décrit à la section 3 : on détermine le premier tas (lignes 15-19) dont le nombre d'allumettes possède un *bit* égal à 1 au rang `sauv`. Enfin, on calcule le nombre d'allumettes à enlever (lignes 21-26).

```
void joueurA() {
2   int tas, allu;

4   do {
      printf("Entrer␣le␣numéro␣du␣tas␣et␣le␣nombre␣d'allumettes␣(Ex␣:␣4␣2)␣:␣");
6     scanf("%d%d", &tas, &allu);
      tas−−; }
8   while (jeu[tas] < allu    jeu[tas] == 0    allu <= 0    tas > (nbre_tas − 1));
    jeu[tas] −= allu;
10 }
```

La procédure qui permet au joueur A de jouer est relativement simple : après avoir afficher un message (ligne 5), on attend deux entiers correspondant au tas et au nombre d'allumettes à enlever sur ce tas. Si le tas est vide, n'existe pas ou que le nombre d'allumettes est trop grand ou négatif, on demande à nouveau la saisie. Si les nombres fournis sont corrects, on enlève les allumettes au tas spécifié.

```
void affiche () {
2   int i, j;

4   clrscr ();
    printf("\n");
6   printf("␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣Le␣jeu␣de␣NIM\n");
    printf("␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣−−−−−−−−−−−−−−\n\n");
8   for (i = 10; i >= 1; i−−) {
      printf("␣%2d␣", i);
10    for (j = 0; j < nbre_tas; j++) {
    if (jeu[j] >= i)
12      printf("␣o−−−−−␣");
        else
14      printf("␣␣␣␣␣␣␣␣");
      };
16    printf("␣\n");
    };
18  printf("␣␣␣␣␣␣␣−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−");
    printf("−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−\n");
20  printf("␣␣␣␣␣␣␣␣␣1␣␣␣␣␣␣␣2␣␣␣␣␣␣␣␣3␣␣␣␣␣␣4␣␣␣␣␣␣␣5");
    printf("␣␣␣␣␣␣␣␣6␣␣␣␣␣␣7␣␣␣␣␣␣␣8␣␣␣␣␣␣␣␣9␣␣␣␣␣␣10\n");
22  printf("␣\n");
}
```

La procédure d'affichage permet au joueur de connaître l'état du jeu : on symbolise les tas par des piquets et les allumettes par des « disques » que l'on affiche de haut en bas selon le nombre d'allumettes de chaque tas.

```
int  est_fini () {
2    int  total = 0;
     int  i;
4
     for (i=0; i < nbre_tas; i++)
6        total += jeu[i];

8    return(!total);
}
```

La fonction `est_fini`permet de tester si le jeu est fini : elle parcourt l'ensemble des tas et calcule le nombre total d'allumettes. S'il est nul, elle renvoie 1 et sinon 0.

```
void jouer() {
2    while (1) {
       joueurB();
4       affiche ();
       if ( est_fini ()) {
6          printf("J'ai␣gagne␣!\n");
          break;
8       };
       printf("Je␣viens␣de␣jouer.␣C'est␣a␣vous...\n");
10
       joueurA();
12      affiche ();
       if ( est_fini ()) {
14         printf("Bravo␣!␣Vous␣avez␣gagne␣!\n");
          break;
16      };
       printf("Appuyer␣sur␣une␣touche␣pour␣continuer.\n");
18     while (!kbhit());
        }
20 }
```

Cette procédure (`jouer`) gère l'enchaînement des différentes actions définies précédemment : dans une boucle infinie (ligne 2), on fait jouer le joueur B et on affiche. Si le jeu est fini, on sort de la boucle (ligne 7) en affichant un message (ligne 8). Sinon, le joueur A joue et on affiche l'état du jeu. De même que précédemment, si le jeu est fini, on sort de la boucle avec un message. S'il faut continuer, on ménage une pause (lignes 17-18) pour permettre au joueur de voir le jeu.

```
void main () {
2    char c;

4    while (1) {
       while (init());
6       affiche ();
       printf("Appuyer␣sur␣une␣touche␣pour␣commencer.\n");
8      while (!kbhit());

10     jouer();
```

```
12      printf("Voulez-vous recommencer ? (O/N)");
        do
14        c = getchar();
        while (c != 'N' && c != 'n' && c != 'O' && c != 'o');
16      if (c == 'N'    c == 'n') break;
    }
18  printf("\nA bientot !\n");
}
```

Finalement, la procédure principale encapsule tout ce qui a été vu précédemment dans une boucle infinie : on demande l'initialisation tant que celle-ci échoue, on affiche le jeu et on ménage un pause (lignes 6-8). Ensuite on lance le jeu. Lorsque celui-ci est terminé, on demande au joueur s'il veut continuer. S'il veut quitter le jeu, on sort de la boucle et donc du programme. Sinon, on reboucle sur l'initialisation.

5 Trace d'exécution

La trace d'exécution va montrer que le joueur B joue bien comme on l'a prévu dans la section 2. Pour cela, nous allons étudier deux cas que nous espérons suffisamment représentatifs et exhaustifs :

- cas où toutes les sommes sont paires ;
- cas où il existe des sommes impaires

Nous avons éliminer de la trace d'exécution les messages qui étaient inutiles ici tel le message de pause ou encore celui demandant au joueur A de jouer. Il ne reste donc que l'état du jeu avant et après que le joueur B a joué.

5.1 Premier cas

C'est le cas le plus simple à tester : en effet, il suffit de vérifier si l'ordinateur joue au hasard — ce qui est difficile à voir sur un seul exemple. Prenons pas exemple le jeu avec 2 tas comportant chacun 2 allumettes.

```
Le jeu de NIM
-------------

 10
  9
  8
  7
  6
  5
  4
  3
  2  o----  o----
  1  o----  o----
  ------------------------------------------------------------------------
 1      2      3      4      5      6      7      8      9      10
```

Ce qui donne après que le joueur B a joué :

```
Le jeu de NIM
-------------

 10
  9
  8
  7
  6
  5
  4
  3
  2  o----
  1  o----  o----
  ------------------------------------------------------------------
 1      2      3      4      5      6      7      8      9      10
```

5.2 Deuxième cas

Reprenons pour ce cas la configuration donnée dans le sujet, soit 3 tas contenant respectivement 3, 4 et 5 allumettes.

```
Le jeu de NIM
-------------

 10
  9
  8
  7
  6
  5                o----
  4         o----  o----
  3  o----  o----  o----
  2  o----  o----  o----
  1  o----  o----  o----
  ------------------------------------------------------------------
 1      2      3      4      5      6      7      8      9      10
```

Compte tenu du calcul des sommes ($S = [0; 0; 1; 0]$), il faut agir sur le second *bit* du nombre d'allumettes maximum (ici 5) auquel on enlèvera deux allumettes, ce qui est bien réalisé par le programme.

```
Le jeu de NIM
-------------

 10
  9
  8
  7
  6
  5                o----
  4         o----  o----
  3         o----  o----
  2         o----  o----
  1  o----  o----  o----
  ------------------------------------------------------------------
 1      2      3      4      5      6      7      8      9      10
```

Si l'on se trouve dans une configuration où il y a quatre imparités, observons si l'ordinateur rend toutes les sommes paires.

```
Le jeu de NIM
-------------

 10
  9                o----
  8                o----
  7                o----  o----
  6                o----  o----
  5  o----         o----  o----
  4  o----  o----  o----  o----
  3  o----  o----  o----  o----
  2  o----  o----  o----  o----
  1  o----  o----  o----  o----
  ------------------------------------------------------------------
 1      2      3      4      5      6      7      8      9      10
```

Comme le montre la trace suivante, le programme a bien fait disparaître les quatre imparités.

```
Le jeu de NIM
-------------

 10
  9
  8
  7                       o----
  6                o----  o----
  5  o----         o----  o----
  4  o----  o----  o----  o----
  3  o----  o----  o----  o----
  2  o----  o----  o----  o----
  1  o----  o----  o----  o----
  ------------------------------------------------------------------
 1      2      3      4      5      6      7      8      9      10
```