Rapport de projet Algorithmique
Le compte est bon

MAKHLOUF Mohamed Mehdi
EL AFRIT Mohamed Amine
BUISSON Rémi
KAIDI Sanaa

LaTeX

Table des matières

Chapitre 1

Introduction

1.1 Problème

Le compte est bon est un jeu se déroulant de la façon suivante : un entier x et plusieurs entiers y_1,...,y_l sont choisis. Il s'agit alors d'obtenir x comme résultat d'une expression arithmétique utilisant les entiers y_1,...,y_l. On souhaite écrire un algorithme qui permet d'obtenir une telle expression si celà est possible, ou bien une meilleure approximation dans le cas contraire.

1.2 Exemple

Voici un exemple d'expérience :
x = 26
y_1=1
y_2=4
y_3=5
y_4=3

Le programme donne alors : 3.(5+4)-1 = 26.
En fait l'algorithme va effectuer les étapes suivantes :

- au départ l'algorithme analyse : 1, 4, 5, 3 (comme entrée)
- après l'algorithme analyse : 9, 1, 3 (où 9=4+5)
- l'algorithme analyse ensuite : 27, 1 (où 27=9.3)
- enfin l'algorithme donne : 26 (où 26=27-1)

1.3 Pourquoi le langage C ?

Le langage C nous a été imposé mais nous voulions simplement ajouté qu'un "compte est bon" a intérêt à être implanté dans ce langage.
En effet, un algorithme qui demande autant de calculs doit être implanté dans un langage rapide : c'est la particularité du C.

Chapitre 2

L'idée principale de la solution

Pour visualiser les différentes possibilités de solutions, il est plus facile de les représenter sous forme d'arbre de calcul (cf. 2.1). Cette idée, quoique courante et "populaire", est difficilement implantable. Pour celà on ne va pas utiliser la structure arbre, on va plutôt coder chaque résultat de l'arbre des calculs de manière ordonnée. C'est justement le rôle du compteur tableau (cf. le rapport d'algorithmique).
Mais cette idée nous met face à un dilemme : soit parcourir l'arbre verticalement et gagner en complexité en temps, soit le parcourir en largeur .
Faute de temps (pour combiner les deux versions) il nous a semblé judicieux de vous présenter les deux versions.

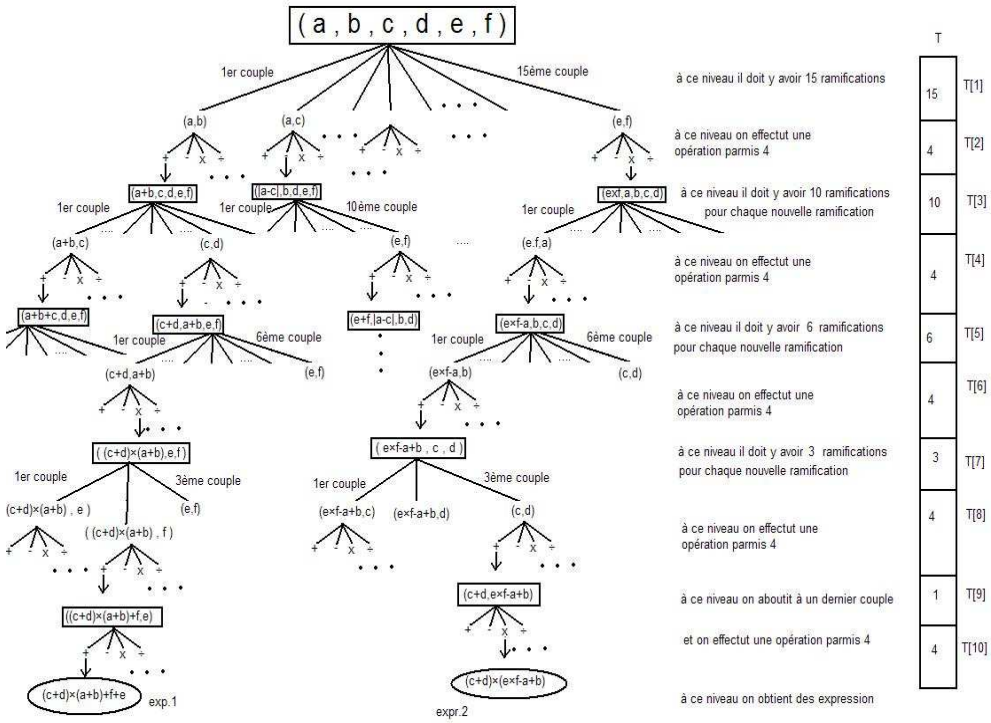

FIG. 2.1 – Arbre des calculs

Chapitre 3

L'implantation du programme en C

3.1 La problématique

Une fois l'algorithme fait, il faut nécessairement passer à la phrase d'implantation.
En effet, programmer l'algorithme permet de vérifier son fonctionnement dans le contexte d'une exécution par une machine, qui fait "bêtement" ce qu'on lui dit et ainsi nous permet de nous focaliser sur ce que fait l'algorithme et non sur ce qu'il doit faire.
Ainsi, on peut trouver des erreurs dans l'algorithme et voire, l'optimiser.

3.2 L'utilisation de structures pour mieux "coller" à l'algorithme : ce qui est en commun pour les deux solutions

Les types abstraits, utilisés dans l'algorithme, ne sont pas implantés par le langage C dans sa version normalisée par l'ANCSI (*American National Standards Institute*).

Dans l'algorithme (cf. le rapport d'algorithmiquerithmique) apparaît la notion de séquence.
Pour mieux "coller" à l'algorithme, nous avons donc implanté une strcuture `sequence_` qui contient un entier et un pointeur vers l'élément suivant.
En effet, une séquence n'est pas implantable via un tableau car elle n'est pas indicée, c'est une liste.
Néanmoins, nous avons quand même utilisé un tableau de structures `sequence_` afin de pallier aux problèmes posés par l'interdiction d'utiliser la fonction `malloc()`. C'est un autre problème car il faut fixer au préalable la taille du tableau. On a choisi une taille de 20 ce qui nous a semblé résonnable, donc nos programmes n'acceptent pas plus de 20 entiers pour un jeu. Mais on pourra quand même modifier cette valeur en modifiant simplement la valeur passée à la directive préprocesseur (`#define MAX_TAB`).
En plus, on a utilisé un tableau de deux entiers pour representer `un couple`.
Aussi pour représenter les différents opérateurs arithmétiques, on a construit un tableau de 4 caractères, contenant évidemment les 4 opérateurs arithmétiques.

3.3 Les fonctions utilisées (en commun pour les deux versions)

Les fonctions principales contenues dans ce fichier sont :
- `nbCombinaison()` : renvoie le nombre de combinaison de 2 parmis n,
- `recherche()` :permet de chercher la solution s'il en existe une,
- `calcul()` : permet d'effectuer une opération sur un couple d'entier (cf. 3.2),
- `couple()` : cette fonction remplie un tableau avec les deux entiers du i^{e}couple de la séquence, (cf. le rapport d'algorithmique),

– `remplacer()` : cette fonction ajoute au début d'une séquence un entier résultant de l'opération effectuée sur deux entiers qu'on supprime par la suite de la séquence.

3.4 La premiere version : "parcours en profondeur" (récursive)

3.4.1 Le fonctionnement

Le fonctionnement consiste, comme on l'a déjà dit (cf.2), à parcourir l'arbre de calcul en profondeur, mais lors de l'implantation du compteur tableau on a remarqué que l'incrémentation de celui ci peut se faire sans utiliser un tableau. On peut plutôt utiliser des boucles "for" imbriquées de manière récursive (cf. 6.1).

3.4.2 Les structures utilisées : en particulier pour cette version

Nous avons choisi d'implanter une structure `operations_` qui n'est ni plus ni moins qu'une liste chaînée d'opérations.

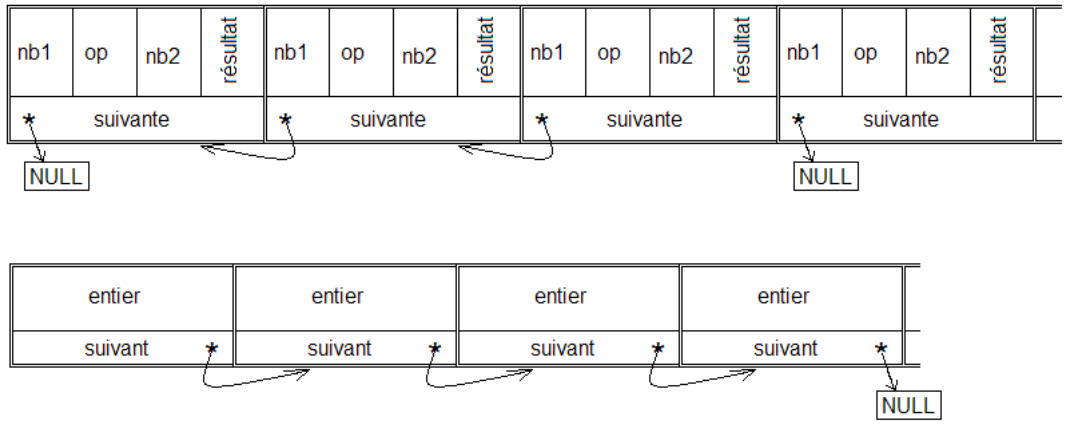

FIG. 3.1 – Structure utilisée dans le parcours en profondeur

3.4.3 Les problèmes

Bien que la compléxité en temps de cet algorithme soit bonne et qu'il s'exécute trés rapidement, il ne donne pas le resultat souhaité avec le minimum d'opérations (cf. 4). En effet le parcours vertical de "l'arbre", ne permet pas de trouver la meilleure solution (moins d'opérations) s'il en existe deux : une a la gauche de l'arbre mais tout au fond et l'autre a droite mais tout au début (comme le montre le schéma 4.1), l'algorithme va s'arrêter en trouvant la prèmiere solution alors qu'il en existe une autre plus simple et qui nécessite moins d'operations.
De plus, on doit appeler la fonction recherche une seconde fois pour chercher la solution la plus proche s'il n' y a pas de solution exacte, ce qui double la complexité en temps (cf. 4.0.6 second exemple).
C'est notre point de départ pour améliorer cette version et donner naissance à une version de parcours en largeur.

3.5 La deuxieme version : “parcours en largeur” (itérative)

3.5.1 Le fonctionnement

Le fonctionnement de notre algorithme consiste à parcourir “l’arbre” en largeur c’est à dire horizontalement. Pour celà on a utilise un compteur tableau (cf. le rapport d’algorithmique) sur le même principe de calculs inventé par Al-Khawarasmi. En effet le tableau est initialisé à des 0 et des 1 comme suit :

1	0	1	0	1	0	1	1	0	0

Ensuite on commence à la case d’indice $2 \times (nb.element) - 3$ (ici nb.element $= 5$) (cf. rapport algorithme pour la démonstration (par récurence)) et on incrémente à chaque fois jusqu’à ce qu’on trouve un résultat et tant qu’on a des opérations à effectuer.
Dans cette version la fonction recherche (cf. 3.3) est itérative.

3.5.2 Les structures utilisées : pour cette version

La taille du compteur tableau (cf. le rapport d’algorithmique) dépend du nombre d’entiers du jeu. Il faut définir à l’avance la taille d’un tableau en C. Pour celà on utilise des directives préprocesseur. Cette taille peut être calculée ; elle est à peut près égale au double du nombre d’éléments du jeu (cf. rapport algo pour la démonstration).

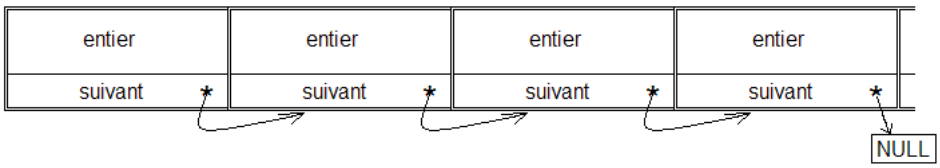

FIG. 3.2 – Structure utilisée dans le parcours en largeur

3.5.3 Les fonctions spécifiques à cette version

- `suivant()` : permet d’incrémenter le compteur tableau (cf. le rapport d’algorithmique),
- `valeur()` : permet de calculer la valeur de l’expression codée par le compteur tableau,
- `afficher()` : permet d’afficher les étapes de calcul de l’expression codée par le compteur tableau.

3.5.4 Le problème

cette version est plus complexe en temps ce qui rend son exécution lente pour certains exemples :
- la solution est en bas de “l’arbre” (c’est à dire que le compteur tableau atteint les plus grandes valeurs).
- Il n’y a pas de solution exacte (le compte n’est pas bon) ; dans ce cas aussi le compteur doit s’incrementer jusqu’à la fin.

3.5.5 Avantages par rapport à la première version

- Cette version, bien que de complexité grande en temps, trouve la solution exacte ou la plus proche avec le minimum d’opérations.
- Avec cette version on n’affiche pas des opérations inutiles.
- On n’utilise plus une structure complexe (cf. 3.4.2) pour stocker les opérations faites (complexité en espace moindre).

– On n'a plus besoin d'exécuter la fonction recherche deux fois quand on cherche la solution la plus proche : tout se fait en une seule fois (cf. le rapport d'algorithmique).

Mais la pertinence du résultat obtenu, sa simplicité et sa structure rendent cette solution plus attractive.

3.6 L'organisation des fichiers

3.6.1 Le fichier d'entête structures.h

Ce fichier contient les définitions de toutes les structures ainsi que les directives pré-processeur (`#define CONSTANTE VALEUR`).

3.6.2 Le fichier fonction.h

Ce fichier contient les prototypes des fonctions utilisées dans le programme (cf. 3.3 & 3.5.3).

3.6.3 Le fichier fonction.c

Ce fichier contient les implémentations des fonctions dont le prototype est definie dans le fichier fonctions.h (cf. 3.6.2).

3.6.4 Le fichier main.c

Ce fichier contient la fonction principale ; il fait appel aux différentes fonctions implantées dans fonction.c.

3.6.5 Le fichier makefile

Ce fichier contient les règles de construction du projet (fichiers sources et dépendances). Il utilise la commande `$>make`. Ainsi on utilise :

- `$>make` ou `$>make prog` pour compiler le projet,
- `$>make clean` pour supprimer les fichiers intermédiaires (`$>*.o *` ...),
- `$>make mrproper` pour supprimer les fichiers intermédiaires, tout comme `make clean` ainsi que le fichier compilé ; cette commande peut être utile lorsqu'on envoit le projet dans une archive, pour inciter l'utilisateur à compiler et ne pas exécuter le fichier existant.

Chapitre 4

Jeux d'essais

Nous avons effectué nos tests sur une machine de processeur intel centrino fréquencé à 1,73 GHz.

Il est important de réaliser des jeux d'essais lorsqu'on a terminé d'implanter un programme . C'est celà qui va nous permettre d'observer le comportement pratique du programme et juger de sa réussite.
En effet les différents tests effectués sur les deux versions nous ont permis d'affirmer que la deuxième version, c'est à dire, le parcours en largeur de "l'arbre" fonctionne bien et répond à nos attentes (cf. 3.4.3). Il met parfois un peu de temps pour trouver le résultat surtout lorsqu'il s'agit du résultat le plus proche (là où on doit parcourir tout "l'arbre").

4.0.6 Des exemples pour "le compte est bon"

Un premier exemple

Le résultat à chercher est : x = 394.
Les entiers du jeu sont :

4	18	11	27	56	15	22

La première version (parcours en profondeur) affiche :
Le compte est bon :
18 + 4 = 22
22 + 11 = 33
56 - 33 = 23
23 * 15 = 345
345 + 27 = 372
372 + 22 = 394

La deuxième version (parcours en largeur) affiche :
Le compte est bon :
27 * 15 = 405
405 - 11 = 394

Il est bien clair que la seconde version trouve le résultat avec moins d'opérations.
Les solutions sont à peut près localisées de la manière suivante dans "l'arbre" (4.1).

Un autre exemple (complexiter en temps pour la première version)

Le résultat à chercher est : x = 12
Les entiers du jeu sont

123	547	989	324	56	45	122	2314	5637	11

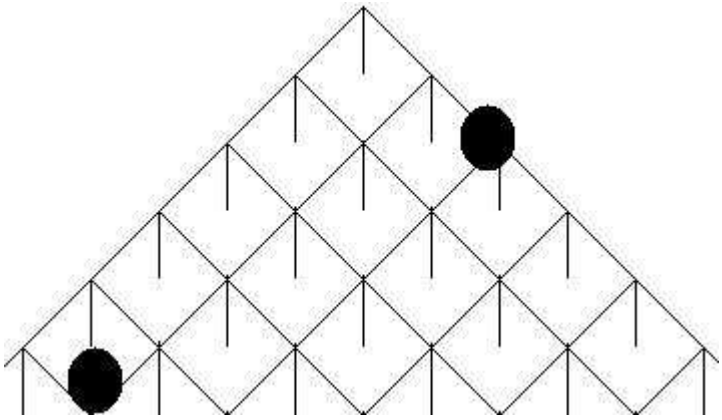

FIG. 4.1 – Exemple 1

La première version (parcours en profondeur) affiche :
Le compte est bon :
547 + 123 = 670
989 + 670 = 1659
1659 + 324 = 1983
56 + 45 = 101
2314 + 11 = 2325
2325 * 101 = 234825
234825 + 5637 = 240462
240462 / 122 = 1971
1983 - 1971 = 12

La deuxième version (parcours en largeur) affiche :
Le compte est bon :
122 - 11 = 111
123 - 111 = 12

Dans cet exemple le parcours en profondeur ne donne pas seulement plus d'opérations mais prend aussi beaucoup plus de temps pour trouver un résultat. En effet : le parcours en largeur trouve le résultat en quelques fractions de secondes alors que le parcours en profondeur le trouve en un vingtaine de secondes!!
En effet, les solutions sont à peut près localisées de la manière suivante dans "l'arbre" (4.2) :

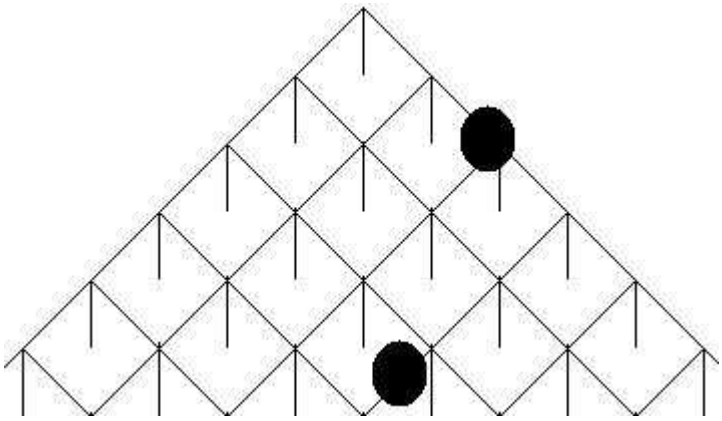

FIG. 4.2 – Exemple 2

Mais si on redonne le même exemple à la version parcours en profondeur mais en inversant l'ordre des entiers, c'est à dire dans cet ordre :

Le résultat à chercher est : x = 12
Les entiers du jeu sont :

11	5637	2314	122	45	56	324	989	547	123

Le parcours en profondeur affiche :
Le compte est bon :
5637 + 11 = 5648
5648 + 2314 = 7962
7962 + 122 = 8084
8084 + 45 = 8129
8129 + 989 = 9118
123 * 56 = 6888
6888 - 324 = 6564
6564 / 547 = 12

Et dans ce cas le résultat est atteind rapidement car la solution est localisée de la manière suivante dans l'arbre : (4.3)

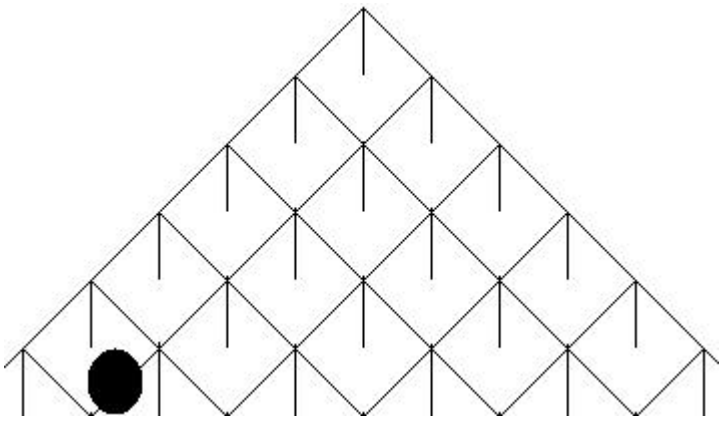

FIG. 4.3 – Exemple 3

Un autre exemple (complexiter en temps pour la deuxième version)

Le résultat à chercher est : x = 960
Les entiers du jeu sont :

1	2	3	4	5	6

La première version (parcours en profondeur) affiche :
Le compte est bon :
3 + 1 = 4
4 * 2 = 8
8 * 4 = 32
32 * 5 = 160
160 * 6 = 960

La deuxième version (parcours en largeur) affiche :
Le compte est bon :
3 + 1 = 4
4 * 2 = 8

8 * 4 = 32
32 * 5 = 160
160 * 6 = 960

L'affichage est le même mais le parcours en profondeur donne dans ce cas le résultat plus rapidement (en quelques fractions de secondes) que le parcours en largeur qui met à peut près 5 secondes pour l'atteindre.

Dans ce cas la solution est localisée de la manière suivante dans l'arbre : (4.4)

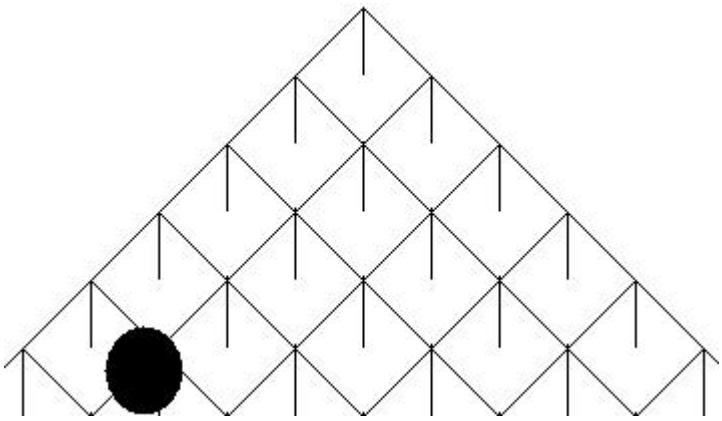

FIG. 4.4 – exemple4

4.0.7 Des exemples pour "le résultat le plus proche"

Un premier exemple

Le résultat à chercher est : x = 983
Les entiers du jeu sont :

1	2	3	4	5	6

La première version (parcours en profondeur) affiche :
La solution la plus proche est :
3 + 1 = 4
4 * 2 = 8
8 * 4 = 32
32 * 5 = 160
160 * 6 = 960

La première version (parcours en profondeur) affiche :
La solution la plus proche est :
3 + 1 = 4
4 * 2 = 8
8 * 4 = 32
32 * 5 = 160
160 * 6 = 960

Un deuxième exemple (cas particulier)

Le résultat à chercher est : x = 660
Les entiers du jeu sont

2	2	7	50	75	100

La première version (parcours en profondeur) affiche :

La solution la plus proche est :
7 + 2 = 9
75 - 2 = 73
73 * 9 = 657
100 / 50 = 2
657 + 2 = 659

La deuxième version (parcours en largeur) affiche :
La solution la plus proche est :
100 + 50 = 150
75 - 2 = 73
73 * 7 = 511
511 + 150 = 661

Les positions des solutions sont à peut près localisées de la manière suivante dans "l'arbre" (4.5) :

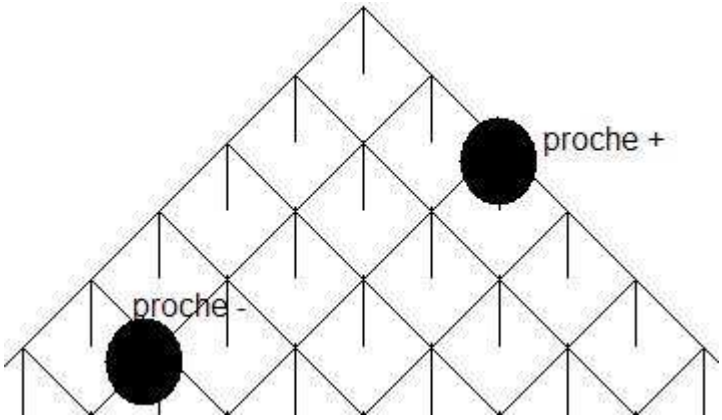

FIG. 4.5 – exemple5

On remarque que le résultat le plus proche n'est pas le même dans ce cas. En fait il s'agit d'un cas particulier ici car il existe "deux" résultats plus proches un par valeur superieure (notée "proche $\oplus$") l'autre par valeur inferieure (notée "proche $\ominus$"). Ce n'est pas une surprise car la recherche, dans la version parcours en profondeur se fait en mémorisant la valeur dont la différence au résultat souhaité soit la plus petite et on ne change cette valeur (dite plus proche) que si la différence est plus petite (cf. le rapport d'algorithmique).
Ici la recherche en profondeur a trouvé la valeur "proche $\ominus$" avant "proche $\oplus$" et donc la première valeur mémorisée ("proche $\ominus$") n'a pas était modifiée car la différence (en valeur absolue) n'est pas plus petite.
D'autre part la recherche en profondeur a trouvé la valeur "proche $\oplus$" avant "proche $\ominus$" et donc la première valeur mémorisée ("proche $\oplus$")n'a pas était modifiée car la différence (en valeur absolue) n'est pas plus petite.

Chapitre 5

Conclusion

Le projet qui nous a été donné nous a permis de découvrir le travail en équipe et les difficultés qui en découlent.
En effet une des premières difficultés est la communication : faire comprendre ses idées aux autres et les convaincre que c'est la meilleure.
Cette difficulté n'est pas observable lorsqu'on mène un projet tout seul, car il est difficile d'être en désaccord avec soi-même.
Un second obstacle est l'échange des projets.
En effet, même si le rapport intermédiaire est bien fait, se plonger dans le code du projet et se mettre dans son ambiance, est un exercice qui n'est pas évident.
Un autre obstacle est le respect du facteur temps.
En effet, respecter le délait de préparation ou achever une mission pendant une durée plus ou moins courte est une tâche à ne pas négliger par une entreprise pour ne pas perdre ses clients.
Ainsi, pour conclure, nous pouvons dire que ce qui a été avant tout mis en avant dans le projet est la démarche de l'ingénieur dans une entreprise : convaincre, communiquer et motiver le personnel pour avancer.

Chapitre 6

Annexes

6.1 Code source

```c
/*main.c*/

#include"structures.h"
#include"fonctions.h"

#define MAX_TAB 20
#define MAX_TABOP 50

int main (int argc, char * argv[])
{
  struct sequence_ tab_entier[MAX_TAB];
  struct operations_ resultat[MAX_TABOP];
  operations o;
  sequence s;
  char tab_op[] = {'+', '-', '*', '/'};
  int nombre;
  int nb_entiers; /*nombre d'entiers du jeux*/
  int a_trouver; /*le resultat Ã  trouver*/
  int i;
  int c;
  int proche;
  int indice_o;

  /* Initialisations */
  i = 0;
  nombre = 0;
  a_trouver = 0;
  nb_entiers = 0;
  s = & tab_entier[0];
  proche = 0;
  indice_o = 0;
  o = &resultat[0];

  /*****obtenir les entiers du jeux*****/

  /* Repeter */
  do
    {
      printf("Saisir un entier (-1 pour arreter) : ");

      /* on demande un entier tant qu'il n'est pas negatif ou egal a -1 */
      while(scanf("%d", &nombre) == 0 || (nombre < 0 && nombre != -1))
        {
          printf("Veuillez saisir un entier positif !! : ");
          while ((c = getchar()) != '\n' && c != EOF); /* mange le tampon */
        }

      /* si le nombre est different de -1, on l'ajoute a la sequence */
      if(nombre != -1)
        {
          /* si c'est le premier de la sequence */
          /*on ne cree pas un element pour rien */
          if(i != 0)
            {
              s->suivant = & tab_entier[i];
              s = s->suivant;
              s->suivant = NULL;
            }
          s->entier = nombre;
          nb_entiers ++;
        }
      i++;
    }
/*tant que le nombre est different de -1 */
/*et qu'il y a de la place dans le tableau*/
  while(nombre != -1 && i < MAX_TAB);

  /******obtenir le resultat Ã  atteindre*****/

  /* si l'utilisateur a saisi des entiers on lui demande le nombre a trouver */
  if(nb_entiers > 0)
    {
      printf("\nSaisir l'entier a trouver : ");
```



```c
      /* on demande le nombre tant qu'il n'est pas positif */
      while(scanf("%d", &a_trouver) == 0 || a_trouver < 0)
        {
          printf("Veuillez saisir un entier positif !! : ");
          while ((c = getchar()) != '\n' && c != EOF); /* mange le tampon */
        }
    }

  proche = a_trouver;

  printf("Recherche d'une combinaison lineaire pour obtenir %d ...\n",
         a_trouver);

  /*****recherche*****/

  /* si on n'a pas atteint le resultat dans cette recherche,*/
  /* on fait la recherche sur le nombre le plus proche */
  if(!recherche(nb_entiers, tab_entier, a_trouver, &proche,
                resultat, MAX_TABOP, &indice_o, &o))
    {
      printf("La solution la plus proche est :\n");
      recherche(nb_entiers, tab_entier, a_trouver + proche, &proche,
                resultat, MAX_TABOP, &indice_o, &o);
    }
  else
    {
      printf("Le compte est bon :\n");
    }

  /*****affichage*****/

  /* on affiche les calculs effectues */
  while(o != NULL)
    {
      printf("%d %c %d = %d\n", max(o->nb1,o->nb2), tab_op[o->op - 1],
             min(o->nb2,o->nb1), o->resultat);
      o = o->suivante;
    }

  return 0;
}
```

```
#ifndef STRUCTURES_H
#define STRUCTURES_H

struct sequence_
{
  int entier;
  struct sequence_ * suivant;
};

struct operations_
{
  int nb1;
  int op;
  int nb2;
  int resultat;
  struct operations_ * suivante;
};

typedef struct sequence_ * sequence;
typedef struct operations_ * operations;

#endif
```

```c
/*fonction.c*/

#include"fonctions.h"

int /* on retourne un entier, le nombre de combinaisons de 2 parmis n */
nbCombinaison( int n /* le nombre n de C n 2 */)
{
  return n*(n-1)/2;
}

/*********************************/

/*fonftion qui donne le maximum de deux entiers*/

int max(int nb1, int nb2)
{
  int max;
  max = nb1;
  if(nb2>nb1)
    max = nb2;
  return max;
}

/*********************************/

/*fonftion qui donne le minimum de deux entiers*/

int min(int nb1, int nb2)
{
  int min;
  min = nb1;
  if(nb2<nb1)
    min = nb2;
  return min;
}

/*********************************/

/*on n'a pas utiliser une structure d'arbre mais les differentes possibilités*/
/*de calcul peuvent etre schematisés par un arbre*/
/*cette version de recherche fait comme si */
/*on parcourt les solutions en profondeur*/

int recherche(int nb_elements, sequence s, int a_trouver, int *  proche,
              struct operations_ o[], int taille_o, int * indice_o,
              operations * op)
{
  int i, j;
  int resultat;
  int difference;
  int atteint; /* booleen a vrai si le resultat a ete atteint */
  int icouple[2]; /* un couple d'elements */
  sequence s2; /* prend les sequences renvoyees par remplacer()  */
  /* le tableau de sequence_ a remplir a chaque remplacer() */
  struct sequence_ t[MAX_TAB];

  i = 1;
  j = 1;
  atteint = 0;
  difference = 0;

  /* on parcourt tous les couples possibles*/
  /* tant qu'on n'a pas trouve la solution */
  while(i <= nbCombinaison(nb_elements) && !atteint)
    {
      /*on parcourt tous les operateurs*/
      /* tant qu'on n'a pas trouve la solution */
      while(j <= 4 && !atteint)
        {
          couple(nb_elements, icouple, i, 0, s); /* calcul du ieme couple */
          resultat = calcul(icouple[0], icouple[1], j); /* on fait le calcul*/

          /*on calcul la difference entre le nombre resultat */
```



```c
          /*et le nombre a trouver*/
          difference = resultat - a_trouver;

          /* on compare le resultat trouve avec le nombre demande */
          if(difference == 0)
            {
              *proche = 0; /* proche=0 car on a atteint le bon resultat */
              atteint = 1; /* atteint vaut vrai car on a atteint la solution*/

              /* on enregistre les operandes, l'operation et le resultat  */
              o[*indice_o].op = j;
              o[*indice_o].nb1 = icouple[0];
              o[*indice_o].nb2 = icouple[1];
              o[*indice_o].resultat = resultat;
              o[*indice_o].suivante = NULL;
              /*ici, on ne peut pas descendre plus dans l'abre de recursivite*/
              /*c'est le dernier calcul; l'operation suivante n'existe pas  */
            }
          else
            {
              /* on remplace la sequence courante par une sequence qui ne*/
              /*contient plus le ieme couple */
              /*et a laquelle on ajoute le resultat*/
              s2 = remplacer(s, t, icouple[0], icouple[1], resultat);

              /* si la difference est plus proche de nonbre */
              /*a trouver que proche, on affecte a proche difference */
              if(fabs((double)difference) < fabs((double)*proche))
                {
                  *proche = difference;
                }

              /* recursivite sur la nouvelle sequence */
              atteint = recherche(nb_elements - 1, s2, a_trouver,
                                  proche, o, taille_o, indice_o, op);

              /* si on a atteint le resultat dans la recursivite,*/
              /* on enregistre le calcul courant */
              if(atteint)
                {
                  /* on enregistre le resultat courant*/
                  /* si on a assez de place dans le tableau */
                  if(*indice_o < taille_o)
                    {
                      *indice_o += 1; /* case suivante du tableau */
                      o[*indice_o].op = j;
                      o[*indice_o].nb1 = icouple[0];
                      o[*indice_o].nb2 = icouple[1];
                      o[*indice_o].resultat = resultat;
                      o[*indice_o].suivante = &o[*indice_o - 1];
                      /* le calcul suivant est celui effectue dans */
                      /*la recursivite donc c'est la case precedente */
                      /*du tableau */
                      *op = &o[*indice_o]; /* le pointeur sur la liste de*/
                      /* calculs pointe dÃ©sormais sur le calcul courant */
                    }
                }
            }     /*fin else*/
          j ++;
        }   /*fin while*/
      j = 1;
      i ++;
    }  /*fin while*/

  return atteint;
}

/*********************************/

int  /* le resultat du calcul est un entier */
calcul(int nb1, /* le nombre 1 */
       int nb2, /* le nombre 2 */
       int operateur) /* selon l'operateur: 1, 2, 3 ou 4 */
```

```
    {
      int minim;
      int maxim;
150   int retour;

      maxim = max(nb1,nb2);
      minim = min(nb1,nb2);

155   /* selon l'operateur on effectue une operation differente */
      switch (operateur)
        {
        case 1 : retour = nb1 + nb2;
          break;
160     case 2 : retour = maxim - minim ;
          break;
        case 3 : retour = nb1 * nb2;
          break;
        case 4 :
165       if(nb1 == 0 || nb2 == 0) /*si un des deux nombres est 0 pas de division*/
            {
              retour = 0;
            }
          else
170         {
              if(maxim % minim == 0)  /*on n'effectue une division que*/
                                      /* si le resultat est entier*/
                {
                  retour = maxim / minim;
175             }
              else
                {
                  retour = 0;
                }
180         }
          break;
        default: retour = 0;
        }
      return retour;
185 }

    /*******************************/

    void /* on ne renvoit rien on modifie un tableau */
190 couple(int nb_elements, /* le nombre d'elements dans la sequence */
           int * tab, /* le tableau d'elements a modifier  */
           int i, /* le ieme couple a recuperer */
           int position, /* position courante (position eme couple) */
           sequence s) /* la sequence sur laquelle travailler */
195 {
      sequence s2;

      /* pointeur temporaire pour parcourir la sequence */
      s2 = s;
200
      /* si on demande un couple qui n'existe pas on retourne -1 */
      if(i < 1 || i > nbCombinaison(nb_elements))
        {
          tab[0] = -1;
205       tab[1] = -1;
        }
      else
        {
          /* tant qu'on n'a pas atteint la position voulue*/
210       /* ou la fin de la sequence on incremente les compteurs */
          while(s2 != NULL && i > position)
            {
              s2 = s2->suivant;
              position ++;
215         }

          /* si on est toujours dans la sequence */
          /*et que les positions correspondent c'est bon */
          if(s2 != NULL && i == position)
```



```
220         {
              tab[0] = s->entier;
              tab[1] = s2->entier;
            }
          else /* sinon appel recursif */
225         {
              couple(nb_elements, tab, i, position - 1, s->suivant);
            }
        }
    }
230
    /*******************************/

    sequence /* la sequence renvoyee */
    remplacer(sequence s, /* la sequence d'origine */
235           struct sequence_ tab[], /* le tableau de sequences a remplir */
              int nb1, /* le premier entier du couple a remplacer */
              int nb2, /* le second entier du couple a remplacer */
              int resultat /* le resultat du calcul nb1 op nb2 */)
    {
240
      int nb_suppr1; /* le nombre de non-copies de nb1 */
      int nb_suppr2; /* le nombre de non-copies de nb2 */
      int index; /* l'index du tableau de sequence_ */
      int copie;
245   sequence s2; /* la sequence a retourner */

      copie = 0;
      index = 0;
      nb_suppr1 = 0;
250   nb_suppr2 = 0;

      /* si les deux nombres sont egaux,*/
      /* on ne copie pas une fois de plus (cas particulier) */
      if(nb1 == nb2)
255     {
          nb_suppr1 = -1;
          nb_suppr2 = -1;
        }

260   /* on copie le resultat en tete de sequence */
      tab[index].entier = resultat;
      tab[index].suivant = NULL;

      while(s != NULL)
265     {
          /* si on a deja supprime nb1 */
          /*et que on trouve nb1 dans la sequence, on le copie */
          if(nb_suppr1 >= 1 && s->entier == nb1)
            {
270           tab[index].suivant = & tab[index + 1];
              index ++;
              tab[index].entier = s->entier;
              tab[index].suivant = NULL;
              copie = 1;
275         }

          /* si on a deja supprime nb2 et que on trouve nb2 dans la sequence,*/
          /* on le copie */
          if(nb_suppr2 >= 1 && s->entier == nb2 && !copie)
280         {
              tab[index].suivant = & tab[index + 1];
              index ++;
              tab[index].entier = s->entier;
              tab[index].suivant = NULL;
285         }
          copie = 0;

          /* si l'entier n'est ni nb1 ni nb2 on le recopie */
          if(s->entier != nb1 && s->entier != nb2)
290         {
              tab[index].suivant = & tab[index + 1];
              index ++;
```

```
              tab[index].entier = s->entier;
              tab[index].suivant = NULL;
            }
          else
            {
              /* sinon on supprime en incrementant le compteur correspondant */
              if(s->entier == nb1)
                {
                  nb_suppr1 ++;
                }
              else
                {
                  nb_suppr2 ++;
                }
            }

          /* element suivant */
          s = s->suivant;
        }

      s2 = tab;

      return s2;
    }
```

```
#ifndef FONCTIONS_H
#define FONCTIONS_H

#define MAX_TAB 20

#include<stdio.h>
#include<stdlib.h>
#include<math.h>
#include<string.h>
#include"structures.h"

/* Définition des prototypes de fonctions */

 /* fonction pour calculer le nombre de combinaisons de 2 parmis n */
int nbCombinaison(int n);

/* permet de rechercher la solution */
int recherche(int nb_elements, sequence s, int a_trouver, int *  proche,
              struct operations_ o[], int taille_o, int * indice_o,
              operations * op);

/* permet de calculer nb1 operateur nb2 */
int calcul(int nb1, int nb2, int operateur);

 /* renvoie le ieme couple de la sequence */
void /* on ne renvoit rien on modifie un tableau */
couple(int nb_elements, int * tab, int i, int position, sequence s);

 /*fonction qui permet de remplacer un couple par le resultat de nb1 op nb2 */
sequence remplacer(sequence s, struct sequence_ tab[],int nb1,int nb2,
                   int resultat);

int max(int nb1, int nb2);  /*fonction qui donne le maximum de nb1 et nb2*/

int min(int nb1, int nb2);  /*fonction qui donne le minimum de nb1 et nb2*/

#endif
```