PROJET MALG

Groupe 8

Préparé par

CRAPANZANO Claire
FERRÉ Nicolas
MOLLARD Romaric
RUCHOT Guillaume

Mai 2017

Sommaire

1 Introduction

Ce document présente différentes méthodes de preuves partielles et complètes des trois programmes du sujet numéro 8.

2 Programme premier

2.1 Description de l'algorithme

Ce programme demande à l'utilisateur un nombre n quelconque (entier), et le programme affiche alors les n premiers nombres premiers en partant de zero. Si n est négatif ou nul, le programme ne doit rien afficher.

2.2 Traduction PlusCal

```
EXTENDS Integers, TLC
CONSTANTS n

(*
--algorithm algo1 {

    variables i=3,count,c=2,nbprintf=0,lastprintedvalue=2,previouslastprintedvalue=1;
    {
        if (n >= 1) {
            print <<"First ", n, " prime numbers are :">>;
            print <<2>>;
            nbprintf := nbprintf + 1;
        };

        count := 2;
        while (count <= n) {
            c := 2;
            while (c <= (i - 1) /\ (i % c) # 0) {
                c := c + 1;
            };

            if (c = i) {
                print <<i>>;
                nbprintf := nbprintf + 1;
                previouslastprintedvalue := lastprintedvalue;
                lastprintedvalue := i;
                count := count + 1;
            };

            i := i + 1;
        }
    }
}

*)
\* BEGIN TRANSLATION
CONSTANT defaultInitValue
VARIABLES i, count, c, nbprintf, lastprintedvalue, previouslastprintedvalue,
          pc

vars == << i, count, c, nbprintf, lastprintedvalue, previouslastprintedvalue,
           pc >>
```

```
Init == (* Global variables *)
        /\ i = 3
        /\ count = defaultInitValue
        /\ c = 2
        /\ nbprintf = 0
        /\ lastprintedvalue = 2
        /\ previouslastprintedvalue = 1
        /\ pc = "Lbl_1"

Lbl_1 == /\ pc = "Lbl_1"
         /\ IF n >= 1
               THEN /\ PrintT(<<"First ", n, " prime numbers are :">>)
                    /\ PrintT(<<2>>)
                    /\ nbprintf' = nbprintf + 1
               ELSE /\ TRUE
                    /\ UNCHANGED nbprintf
         /\ count' = 2
         /\ pc' = "Lbl_2"
         /\ UNCHANGED << i, c, lastprintedvalue, previouslastprintedvalue >>

Lbl_2 == /\ pc = "Lbl_2"
         /\ IF count <= n
               THEN /\ c' = 2
                    /\ pc' = "Lbl_3"
               ELSE /\ pc' = "Done"
                    /\ c' = c
         /\ UNCHANGED << i, count, nbprintf, lastprintedvalue,
                         previouslastprintedvalue >>

Lbl_3 == /\ pc = "Lbl_3"
         /\ IF c <= (i - 1) /\ (i % c) # 0
               THEN /\ c' = c + 1
                    /\ pc' = "Lbl_3"
                    /\ UNCHANGED << i, count, nbprintf, lastprintedvalue,
                                    previouslastprintedvalue >>
               ELSE /\ IF c = i
                          THEN /\ PrintT(<<i>>)
                               /\ nbprintf' = nbprintf + 1
                               /\ previouslastprintedvalue' = lastprintedvalue
                               /\ lastprintedvalue' = i
                               /\ count' = count + 1
                          ELSE /\ TRUE
                               /\ UNCHANGED << count, nbprintf,
                                               lastprintedvalue,
                                               previouslastprintedvalue >>
                    /\ i' = i + 1
                    /\ pc' = "Lbl_2"
                    /\ c' = c

Next == Lbl_1 \/ Lbl_2 \/ Lbl_3
          \/ (* Disjunct to prevent deadlock on termination *)
             (pc = "Done" /\ UNCHANGED vars)

Spec == Init /\ [][Next]_vars

Termination == <>(pc = "Done")
```

```
\* END TRANSLATION
```

Lors de la traduction en PlusCal, nous avons fait le choix d'introduire deux nouvelles variables : *nbprintf*, *previouslastprintedvalue* et *lastprintedvalue*.
Ces variables, qui correspondent respectivement au nombre de valeurs affichés, à l'avant dernière valeur affichée, et à la dernière valeur affichée, permettront de mettre en place correctement la preuve de ce programme.

2.3 Preuve TLA+

```
max(a, b) == IF a > b THEN a ELSE b
prime(a) == \A x \in 2..a : (x = a \/ a % x # 0)

Pre ==
    /\ pc = "Lbl_1" => n \in Int (* l'entree n est conforme *)

Post ==
    /\ pc = "Done" /\ n \geq 0 => nbprintf = n (* le nombre de valeur affichees au total
        est correct *)
    /\ pc = "Done" /\ n < 0 => nbprintf = 0 (* le nombre de valeur affichees au total est
        correct *)

Safe ==
    /\ prime(lastprintedvalue) (* les nombres affiches sont premiers *)
    /\ n \geq 0 => (nbprintf \leq n /\ nbprintf \geq 0) (* test du nombre de printf *)
    /\ n < 0 => nbprintf = 0
    /\ lastprintedvalue > previouslastprintedvalue (* les valeurs affichees sont dans
        l'ordre croissant *)
    /\ lastprintedvalue - previouslastprintedvalue > 1 => (\A x \in
        previouslastprintedvalue + 1 .. lastprintedvalue - 1 : prime(x) = FALSE) (* aucune
        valeur entre deux affichÃ©es n'est premier *)

Exec ==
    /\ n \geq 1 => (count \leq n + 1 /\ count \geq 0) (* limites de count entre 0 et n + 1
        *)
    /\ n \leq 0 => (count \geq 0 /\ count \leq 2)
    /\ c \leq i /\ c \geq 1 (* limites de c entre 1 et i *)
    /\ i \geq 3 /\ i \in Nat (* i ne dÃ©passe pas la limite des entiers *)
```

Nous avons défini plusieurs règles à la suite de la traduction du PlusCal. La première est *Pre*, qui indique si la précondition du programme est bien respectée. Ici la precondition est simplement sur l'entrée n, cette constante doit être entière.
La deuxième est *Post*, et indique la postcondition à obtenir à la fin du programme. Elle vérifie simplement que le nombre de valeurs affichées soit égal à n.
La troisième règle est *Safe*, et désigne ce qui doit être respecté tout au long du programme. D'abord, les nombres affichées *lastprintedvalue* doivent être premiers. De plus, le nombre de valeur affichées doit être toujours compris entre 0 et n (et toujours nul si n est négatif), il s'agit des deuxième et troisième lignes de *Safe*. La quatrième indique que les nombres premiers affichés doivent être affichés dans l'ordre croissant, et la dernière ligne que entre deux valeurs affichées, il n'y a aucun nombre premier.
Ces trois règles permettent de vérifier la correction partielle du programme. En exécutant les tests, on voit que ces conditions sont bien respectées, le programme affiche la liste des n premiers nombres

premiers.

Pour prouver l'absence d'erreur à l'exécution, on doit montrer qu'il n'y a pas de division par zéro et que les variables restent dans les limites des nombres entiers d'une machine. *Exec* permet de tester cette absence d'erreur à l'exécution. Les deux premières lignes montrent que count est compris entre 0 et $n + 1$. n étant un nombre entier, *count* ne dépassera donc pas les bornes des entiers. La troisième ligne montre que c est compris entre 0 et i. Il est important que c ne soit pas égal à zéro, car la seule division du programme (un modulo en réalité) à comme diviseur c. La dernière ligne indique que le minimum de i est 3, et qu'il doit rester dans la limite des naturels.
Bien qu'avec un n raisonnable (¡ 200) la règle *Exec* est respectée par le programme, on peut aisément imaginer que pour un n assez grand (mais restant dans la limite des nombres entiers), i peut dépasser la limite des nombres entiers. En effet, i désigne le nombre à tester (pour savoir s'il est premier, et dans ce cas l'afficher). i va croitre bien plus vite que n, donc le programme ne sera pas sans erreur à l'exécution pour n assez grand.

2.4 Preuve FramaC

Nous avons prouvé deux éléments grace à l'outil FramaC :
1. Le programme ne retourne que des nombres premiers
2. Le programme retourne le nombre de résultats demandé
Pour celà nous avons rajouté quelques données dans l'algorithme afin par exemple de comptabiliser le nombre de printf, qui est matérialisé par la variable 'nbprintf' initialisée à 0.

```
int main(){
  int n, i = 3, count, c;

  /*@ assert i==3; */

  printf("Enter the number of prime numbers required\n");
  scanf("%d",&n);

  //On protege n des valeurs negatives pour framac
  if(n<0){
     n=0;
  }

  //On ajoute une variable pour compter le nombre d'affichages total
  int nbprintf = 0;

  /*@ assert n>=0; */
  /*@ assert nbprintf==0; */

  if (n>=1){
    printf("First %d prime numbers are :\n", n);
    printf("2\n"); //Preuve 1. Les resultats sont tous des nombes premiers
   //On incrÃ©mente nbprintf
   nbprintf++;
  }

  //On s'assure qu'on a affiche 2 pour commencer
  /*@ assert n>=1 ==> nbprintf==1; */ //Preuve 2. On a n==nombre de printf
  /*@ assert n<1 ==> nbprintf==0; */ //Preuve 2. On a n==nombre de printf
```

```
//Il faut prouver que la boucle termine

/*@ loop invariant 2 <= count;
    loop invariant i>count;
    loop invariant nbprintf<=n;
    loop invariant n>=1 ==> count<=n+1;
    loop invariant n>=1 ==> nbprintf==count-1;
    loop invariant n<1 ==> nbprintf==0;
*/
for (count = 2; count <= n; ){

  /*@ assert count<=n;*/

  /*@ loop invariant 2 <= c <= i;
      loop invariant 2 <= count;
      loop invariant i>count;
      loop invariant nbprintf==count-1;
      loop invariant count<=n;
      loop invariant nbprintf<=n;
      loop invariant c>=3 ==> i%(c-1)!=0;
      loop invariant \forall int w; 2<=w<c ==> i%w!=0;
  */
  for (c = 2; c<= i-1;c++){
    if (i%c == 0) {
      /*@ assert i%c==0; */
      break;
    }
    /*@ assert i%c!=0;*/
  }
  /*@ assert i%c==0 || i==c; */
  /*@ assert \forall int w; 2<=w<c ==> i%w!=0;*/
  /*@ assert count<=n;*/
  /*@ assert nbprintf==count-1;*/

  if (c == i){
    /*@ assert i==c; */

    //On prouve que i est premier
    /*@ assert \forall int w; 2<=w<i ==> i%w!=0;*/ //Preuve 1. Les resultats sont tous
        des nombes premiers
    printf("%d\n",i);

   //On incremente nbprintf
   nbprintf++;
    /*@ assert nbprintf==count;*/

    count++;
    /*@ assert count<=n+1;*/
    /*@ assert nbprintf==count-1;*/
    /*@ assert nbprintf<=n;*/
  }

  i++;
  /*@assert count<=n+1;*/
}
/*@ assert count>=n+1; */
```

```
  /*@ assert n>=1 ==> count==n+1; */

  //On prouve qu'il y a le bon nombre de rÃ©sultats
  /*@ assert n>=1 ==> nbprintf==n; */
  /*@ assert n<1 ==> nbprintf==0; */
  /*@ assert n>=0; */


  /*@ assert nbprintf==n; */ //Preuve 2. On a n==nombre de printf

  return 0;
}
```

Nous avons donc bien validé $/*@assert nbprintf == n;*/$ et $/*@assert \forall int w; 2 <= w < i ==> i\%w! = 0;*/$ avant chaque sortie de valeur.

3 Tri 1 : Sélection

3.1 Description de l'algorithme

Ce programme propose à l'utilisateur de créer une liste de n éléments, puis lui demande de rentrer les valeurs de cette liste une par une. Le programme affiche ensuite la liste triée par la méthode du tri sélection.

3.2 Traduction PlusCal

```
---------------------------- MODULE algo2 ----------------------------
EXTENDS Naturals, TLC
CONSTANTS nhat
(*
--algorithm algo2{
    variables c, d, position, swap, array;
    {
        c := 0;
        while(c < (n-1)){
            position := c;
            d := c+1;
            while(d < n){
                if(array[position] > array[d]){
                    position := d;
                };
                d := d+1;
            };
            if(position != c){
                swap := array[c];
                array[c] := array[position];
                array[position] := swap;
            };
            c := c+1;
        };
        print<<"Sorted list in ascending order">>;
        c := 0;
        while(c < n){
            print<<array[c]>>;
            c := c+1;
        };
    }
}
*)


\* BEGIN TRANSLATION
CONSTANT defaultInitValue
VARIABLES c, d, position, swap, array, pc

vars == << c, d, position, swap, array, pc >>
```

```
Init == (* Global variables *)
       /\ c = defaultInitValue
       /\ d = defaultInitValue
       /\ position = defaultInitValue
       /\ swap = defaultInitValue
       /\ array = defaultInitValue
       /\ pc = "Lbl_1"

Lbl_1 == /\ pc = "Lbl_1"
        /\ c' = 0
        /\ pc' = "Lbl_2"
        /\ UNCHANGED << d, position, swap, array >>

Lbl_2 == /\ pc = "Lbl_2"
        /\ IF c < (n-1)
             THEN /\ position' = c
                  /\ d' = c+1
                  /\ pc' = "Lbl_3"
                  /\ c' = c
             ELSE /\ PrintT(<<"Sorted list in ascending order">>)
                  /\ c' = 0
                  /\ pc' = "Lbl_6"
                  /\ UNCHANGED << d, position >>
        /\ UNCHANGED << swap, array >>

Lbl_3 == /\ pc = "Lbl_3"
        /\ IF d < n
             THEN /\ IF array[position] > array[d]
                       THEN /\ position' = d
                       ELSE /\ TRUE
                            /\ UNCHANGED position
                  /\ d' = d+1
                  /\ pc' = "Lbl_3"
                  /\ UNCHANGED << swap, array >>
             ELSE /\ IF position != c
                       THEN /\ swap' = array[c]
                            /\ array' = [array EXCEPT ![c] = array[position]]
                            /\ pc' = "Lbl_4"
                       ELSE /\ pc' = "Lbl_5"
                            /\ UNCHANGED << swap, array >>
                  /\ UNCHANGED << d, position >>
        /\ c' = c

Lbl_4 == /\ pc = "Lbl_4"
        /\ array' = [array EXCEPT ![position] = swap]
        /\ pc' = "Lbl_5"
        /\ UNCHANGED << c, d, position, swap >>

Lbl_5 == /\ pc = "Lbl_5"
        /\ c' = c+1
        /\ pc' = "Lbl_2"
        /\ UNCHANGED << d, position, swap, array >>

Lbl_6 == /\ pc = "Lbl_6"
        /\ IF c < n
             THEN /\ PrintT(<<array[c]>>)
                  /\ c' = c+1
```

```
                    /\ pc' = "Lbl_6"
                ELSE /\ pc' = "Done"
                    /\ c' = c
          /\ UNCHANGED << d, position, swap, array >>

Next == Lbl_1 \/ Lbl_2 \/ Lbl_3 \/ Lbl_4 \/ Lbl_5 \/ Lbl_6
           \/ (* Disjunct to prevent deadlock on termination *)
              (pc = "Done" /\ UNCHANGED vars)

Spec == Init /\ [][Next]_vars

Termination == <>(pc = "Done")

\* END TRANSLATION
```

3.3 Preuve TLA+

On vérifie qu'on ne dépasse pas les bornes au début, que l'indice est toujours dans le tableau et que le tableau est trié à la fin.

```
PreCond == n <= 100
Invariant == c <= n
PostCond == pc = "Done" /\ \A i \in 1..n-1 : array[i] <= array[i+1]
```

3.4 Preuve FramaC

Avec l'outil FramaC, nous avons pu prouver qu'à la fin de la boucle principale, le tableau d'entier renseigné au début de l'algorithme est trié par ordre croissant.
La méthode de tri étant le tri sélection, à chaque itération de la boucle un élément de plus était trié, ainsi c'est ce que l'on a prouvé, pour en déduire le résultat final.
De plus, comme nous déclarons un array de 100 éléments, nous avons ajouté une vérification dans le programme évitant que l'utilisateur rentre un nombre négatif ou supérieur à cette limite lorsque l'on lui demande la taille du tableau.
Enfin, puisque l'on manipule des tableaux, il est souvent nécessaire de rappeler les assertions dans le programme, notamment lorsque l'on swap deux éléments.

```
#include <stdio.h>

int main(){
  int array[100], n = -1, c, d, position, swap;
  while (n<1 || n>100){
  printf("Enter number of elements < 100\n");
  scanf("%d",&n);
  }
  printf("Enter %d integers\n",n);

  // On s'assure que l'on a que 100 elements
  /*@ loop invariant \valid(array+ (0..n-1));
      loop invariant 0<n<=100; */
  for (c=0; c<n; c++){
    scanf("%d", &array[c]);
```

```
  }

  //A chaque iteration, un element de plus est trie, c'est ce que l'on montre avec le
      premier invariant
  /*@ loop invariant \forall int i,j; (0 <= i < j < n && i < c) ==> array[i] <= array[j];
      loop invariant 0 <= c < n;
  */
  for (c = 0; c < (n-1); c++){
    position = c;

    /*@ loop invariant \forall int i; c <= i < d ==> array[position] <= array[i];
        loop invariant \forall int i,j; (0 <= i < j < n && i < c) ==> array[i] <= array[j];
        loop invariant c+1 <= d <= n;
        loop invariant c<=position<n;
        loop assigns d,position; */
    for (d = c+1; d<n; d++){
      if (array[position]>array[d]) position = d;
    }

    //@ assert \forall int i; c<=i<n ==>array[position] <= array[i];
    //@ assert \forall int i,j; (0 <= i < j < n && i < c) ==> array[i] <= array[j];

    if (position != c){
      swap = array[c];
      array[c] = array[position];
      //@ assert \forall int i,j; (0 <= i < j < n && i < c) ==> array[i] <= array[j];
      array[position] = swap;
      //On rappelle les assertions aprÃ¨s la modification de l'array
      //@ assert array[c] <= array[position];
      //@ assert 0 <= c < n;
      //@ assert \forall int i; c<=i<n ==>array[c] <= array[i];
      //@ assert \forall int i,j; (0 <= i < j < n && i < c) ==> array[i] <= array[j];

    }
  }
  //L'assertion suivante est celle qui prouve le fonctionnement du programme,
  //puisque l'on ne modifie plus le tableau par la suite

  //@ assert \forall int i,j; (0 <= i < j <= n-1) ==> array[i] <= array[j] ;
  printf("Sorted list in ascending order :\n");

  /*@ loop invariant \forall int i,j; (0 <= i < j <= n-1) ==> array[i] <= array[j] ;
      loop invariant 0<= c <= n;*/
  for (c=0; c<n; c++){
    //@ assert c<n-1 ==> array[c] <= array[c+1];
    printf("%d\n",array[c]);
  }

  return 0;
}
```

4 Autre algorithme

4.1 Description de l'algorithme

Ce programme mystère calcule les éléments de la formule de récurrence

$$r(i) = 2 * r(i-1) + 2 * r(i-2), r(0) = 1, r(1) = 1$$

Après résolution, ceci correspond à la formule suivante :

$$r(i) = 1/2 * ((1 - \sqrt{(3)})^i + (1 + \sqrt{(3)})^i)$$

4.2 Traduction PlusCal

```
EXTENDS TLC,Integers, Reals

(*
--algorithm Algo3 {
variables a,b,c,i,r;
{
     a:=1;
     b:=1;
     i:=2;
     if(x==0){
        r:=1;
     }else{
        if(x==1){
            r:=1;
        }else{
            while(i-1 < x)
            {
                i:=i+1;
                c:=a;
                a:=b;
                b:=2*c+2*b;
            };
            r:=b;
        }
    }
}

}

*)

\* BEGIN TRANSLATION
CONSTANT x
VARIABLES a, b, c, i, r, pc
```

```
vars == << a, b, c, i, r, pc >>

Init == (* Global variables *)
       /\ a = 1
       /\ b = 1
       /\ c = 0
       /\ r = 0
       /\ i = 2
       /\ pc = "Lbl_1"

Lbl_1 == /\ pc = "Lbl_1"
         /\ a' = 1
         /\ b' = 1
         /\ i' = 2
         /\ IF x=0
              THEN /\ r' = 1
                   /\ pc' = "Done"
              ELSE /\ IF x=1
                        THEN /\ r' = 1
                             /\ pc' = "Done"
                        ELSE /\ pc' = "Lbl_2"
                             /\ r' = r
         /\ c' = c

Lbl_2 == /\ pc = "Lbl_2"
         /\ IF i-1 < x
              THEN /\ i' = i+1
                   /\ c' = a
                   /\ a' = b
                   /\ b' = 2*c'+2*b
                   /\ pc' = "Lbl_2"
                   /\ r' = r
              ELSE /\ r' = b
                   /\ pc' = "Done"
                   /\ UNCHANGED << a, b, c, i >>

Next == Lbl_1 \/ Lbl_2
           \/ (* Disjunct to prevent deadlock on termination *)
              (pc = "Done" /\ UNCHANGED vars)

Spec == Init /\ [][Next]_vars

Termination == <>(pc = "Done")

\* END TRANSLATION
```

4.3 Preuve TLA+

La preuve TLA+ consiste en une preuve partielle du programme.
La traduction automatique produit trois états :
Init : qui représente le programme avant le premier if
Lbl_1 : qui représente les tests et conditions if / else
Lbl_2 : qui représente le while et donc le calcul de la formule décrite ci-dessus.
La variable pc nous permet de connaitre l'état actuel du programme.

```
  PreCond ==
      /\ pc = "Lbl_1" => (n \leq 24 /\ n >= 0)

  TestPreLoop == i>2 \/ b = 1 \* On a pas commencÃ© la boucle donc b=1
  TestPostLoop == i\leq2 \/ b = 2*c+2*a \* On a commencÃ© la boucle donc b= 2*c + 2*a

  PostCond ==
      /\ pc = "Done" => ( (x\leq1 /\ r=1) \/ (x>1 /\ r=2*c+2*a) ) \*TerminÃ© donc soit
          c'est pas passÃ© dans la boucle et c'est 1 soit c'est la formule rÃ©cursive
```

Ceci nous permet de vérifier la formule de récurrence (en admettant c==b[i-1] et a==b[i-2]) et que les résultats pour i=0 et i=1 sont bien 1, soit les valeurs de base de la formule récurrente.
Comme indiqué ci-dessous, n doit être compris entre 0 et 24 pour ne pas avoir de dépassement de capacités.

4.4 Preuve FramaC

A l'aide de l'outil FramaC, nous avons prouvé que l'algorithme calcule bien la formule que nous avons déduit. Nous avions le choix entre tenter de prouver la formule sous forme récursive ou sous forme finale.
Ayant eu peur que la précision flottante de la racine carré présente dans la deuxième solution nuise à la preuve, nous sommes donc partis sur le premier choix.
Ainsi nous avons redéfini la formule récursive sur FramaC à l'aide des axiomes, et nous avons prouvé qu'à chaque itération de la boucle nous avançons en suivant cette formule.
Enfin, nous avons ajouté une vérification sur le nombre entré par l'utilisateur, car il doit être positif et au inférieur strictement à 25. Tout nombre supérieur à 25 entraîne une dépassement des Integer et invalide donc toutes les preuves et le fonctionnement du programme.

```
#include <stdio.h>
#include <limits.h>
int inmax = INT_MAX;
/*@
axiomatic Solve {
  logic integer solve(integer n);

  axiom solve_0: solve(0) == 1;

  axiom solve_1: solve(1) == 1;

  axiom solve_n: \forall integer n; (n>1 ==> solve(n) == 2*solve(n-2)+2*solve(n-1)) && (n<0
      ==> solve(n) == 1);
}*/


/*@
requires 0 <= x <= 24;
requires \forall int k; 0 <= k <= 1 ==> solve(k) == 1 && 2 <= k < 25 ==> solve(k) ==
    2*solve(k-1) + 2*solve(k-2);
assigns \nothing;
ensures \result > 0;
ensures \result == solve(x);
```

```
*/
int p1(int x){
  int a,b,c,i,r,fin;
  a=1;
  b=1;
  i=2;
  if (x==0) {
    r=1;
  } else if (x==1) {
    r=1;
  } else {
    //Ces invariants sont necessaires pour prouver la formule
      /*@ loop invariant positive : a>0 && b>0;
            loop invariant solvb : b == solve(i-1);
            loop invariant solva : a == solve(i-2);
            loop invariant solvc : i>=3 ==> c == solve(i-3);
            loop invariant i : 2 <= i <= x+1;
            loop assigns a,b,c,i;*/
    while (i-1 < x){
      i=i+1;
       c=a;
       a=b;
       b=2*c+2*b;

     }
   r=b;
  }
  //Cette assertion prouve que l'on renvoit le bon resultat
   //@assert solve(x) == r;
  return(r);
}


/*@
     ensures \result == 0;
*/
int main(){
  int v = -1;
  //On verifie que l'utilisateur rentre un nombre valide (0 <= v < 25)
   while (v < 0 || v >= 25){
   printf("Entrez la valeur pour v\n");
   scanf("%d", &v);
   }
   //@ assert v >= 0 && v< 25;
   int p1v = p1(v);
   //@ assert p1v == solve(v);
  printf(" voici la reponse de votre solution p2(%d)=%d\n",v,p1(v));
  return 0;
}
```

5 Conclusion

5.1 Temps et méthodes de travail

Nous avons mis en place un dépôt git pour travailler en équipe, et nous avons divisés les 6 algorithmes entre nous 4 afin de travailler plus efficacement.

	C. CRAPANZANO	**N. FERRÉ**	**R. MOLLARD**	**G. RUCHOT**
Algorithme 1	0h	6h	3h	0h
Algorithme 2	4h	0h	0h	4h
Algorithme 3	0h	0h	3h	6h
Rapport	1h	1h	2h	1h
Total	**5h**	**7h**	**8h**	**11h**