Les bases : exercices corrigés en C
Corrigé

Consignes : Les exercices 2, 4, 6 et 7 sont facultatifs. Dans le cas de l'exercice 5, on pourra se limiter au cas des puissances positives (x^n avec $n \geq 0$).

Objectifs

- Raffiner des problèmes simples ;
- Écrire quelques algorithmes simples ;
- Savoir utiliser les types de base ;
- Savoir utiliser les instructions « élémentaires » : d'entrée/sortie, affichage, bloc...
- Manipuler les conditionnelles ;
- Manipuler les répétitions.

Exercice 1 : Résolution d'une équation du second degré

Soit l'équation du second degré $ax^2 + bx + c = 0$ où a, b et c sont des coefficients réels. Écrire un programme qui saisit les coefficients et affiche les solutions de l'équation.
Indication : Les solutions sont cherchées dans les réels. Ainsi, dans le cas général, en considérant le discriminant $\Delta = b^2 - 4ac$, l'équation admet comme solutions analytiques :

$$\begin{cases} \Delta < 0 & \text{pas de solutions réelles.} \\ \Delta = 0 & \text{une solution double : } \frac{-b}{2a} \\ \Delta > 0 & \text{deux solutions : } x_1 = \frac{-b-\sqrt{\Delta}}{2a} \text{ et } x_2 = \frac{-b+\sqrt{\Delta}}{2a} \end{cases}$$

Quelles sont les solutions de l'équation si le coefficient a est nul ?
Solution : Voici un raffinage possible :

```
1  R0 : Résoudre l'équation du second degré
2
3  R1 : Raffinage De « Résoudre l'équation du second degré »
4       |  Saisir les 3 coefficients              a, b, c: out RÉEL
5       |                          -- les coefficients de l' équation
```

```
6       |  Calculer et afficher les solutions
7
8   R2 : Raffinage De « Saisir les 3 coefficients »
9       |  Écrire("Entrer les valeurs de a, b et c : ")
10      |  Lire(a, b, c)
11
12  R2 : Raffinage De « Calculer et afficher les solutions »
13      |  Si équation du premier degré Alors
14      |      Résoudre l'équation du premier degré bx + c = 0
15      |  Sinon
16      |      Calculer le discriminant
17      |          --> delta: RÉEL          -- le discriminant de ax^2 + bx + c = 0
18      |      Si discriminant nul Alors            -- une solution double
19      |          Écrire("Une solution double : ", - B / (2 * A))
20      |      SinonSi discriminant positif Alors  -- deux solutions distinctes
21      |          Écrire("Deux solutions : ")
22      |          Écrire((-B + √Delta) / (2*A))
23      |          Écrire((-B - √Delta) / (2*A))
24      |      Sinon       { discriminant négatif }        -- pas de solution dans IR
25      |          Écrire("Pas de solution réelle")
26      |      FinSi
27      |  FinSi
28
29  R3 : Raffinage De « Résoudre l'équation du premier degré bx + c = 0 »
30      |  Si B = 0 Alors
31      |      Si C = 0 Alors
32      |          Écrire("Tout IR est solution")
33      |      Sinon
34      |          Écrire("Pas de solution")
35      |      FinSi
36      |  Sinon
37      |      Écrire("Une solution : ", - C / B)
38      |  FinSi
```

Remarque : Notons que nous n'avons pas un premier niveau de raffinage en trois étapes, *saisir*, *calculer*, *afficher* car il est difficile de dissocier les deux dernières étapes car la forme des racines de l'équation du second est très variable et ne peut pas être facilement capturée par un type (deux solution distinctes, une solution double, pas de solution, une infinité de solution). Aussi, nous avons regroupé calcul et affichage.

Et l'algorithme correspondant :

```
1   Algorithme second_degré
2
3       -- Résoudre l'équation du second degré
4
5   Variables
6       a, b, c: Réel       -- les coefficients de l'équation
7       delta: Réel         -- le discriminant
8
9   Début
```

```
10      -- Saisir les 3 coefficients
11      Écrire("Entrer les valeurs de a, b et c : ")
12      Lire(a, b, c)
13
14      -- Calculer et afficher les solutions
15      Si a = 0 Alors              -- équation du premier degré
16          -- Résoudre l'équation du premier degré bx + c = 0
17          Si B = 0 Alors          -- équation constante
18              Si C = 0 Alors
19                  Écrire("Tout IR est solution")
20              Sinon
21                  Écrire("Pas de solution")
22              FinSi
23          Sinon                   -- équation réellement du premier degré
24              Écrire("Une solution : ", - C / B)
25          FinSi
26
27      Sinon                       -- équation réellement du second degré
28          -- Calculer le discriminant delta
29          delta <- b*b - 4*a*c
30
31          -- Déterminer et afficher les solutions
32          Si delta = 0 Alors              -- une solution double
33              Écrire("Une solution double : ", - B / (2 * A))
34          SinonSi delta > 0 Alors -- deux solutions distinctes
35              Écrire("Deux solutions : ")
36              Écrire((-B + sqrt(delta)) / (2*A))
37              Écrire((-B - sqrt(delta)) / (2*A))
38          Sinon   { discriminant négatif }        -- pas de solution dans IR
39              Écrire("Pas de solution réelle")
40          FinSi
41      FinSi
42  Fin.
```

```
1   /*****************************************************************
2    *  Auteur   : Xavier Crégut <cregut@enseeiht.fr>
3    *  Version  : 1.4
4    *  Objectif : Résolution de l'équation du seond degré.
5    *****************************************************************/
6
7   #include <stdio.h>
8   #include <stdlib.h>
9   #include <math.h>
10
11  /* Demande à l'utilisateur les coefficients a, b et c de l'équation du
12   * second degré  ax^2 + bx + c* , calcule et affiche les solutions
13   * réelles de cette équation.
14   */
15  int main()
16  {
17      /* Déclaration des variables (C oblige) */
```

```c
    float a, b, c;      /* coefficients de l'équation   */

    /* Poser le problème */
    printf("Résolution de l'équation du second degré :\n");
    printf("\t    a.x2 + b.x +c = 0\n");

    /* Saisir la valeur des coefficients */
    printf("Donnez la valeur des coefficients a, b et c : ");
    scanf("%f%f%f", &a, &b, &c);

    /* Résoudre l'équation et afficher les résultats */
    if (a == 0) {       /*{ équation du premier degré }*/
        /* résoudre l'équation du premier degré */
        if (b == 0) {
            if (c == 0) {
                printf("Tout les réels sont solution");
            }
            else {
                printf("Pas de solution");
            }
        }
        else {
            printf("Une seule solution : %f", -c / b);
        }
    }
    else {
        /* cas général  */
        float delta = b*b - 4*a*c;              /* le discriminant */
        if (delta > 0) {
            /* deux solutions réelles   */
            printf("Deux solutions :\n");
            printf("\tsolution 1 = %f\n", (-b - sqrt(delta)) / 2 / a);
            printf("\tsolution 2 = %f\n", (-b + sqrt(delta)) / 2 / a);
        }
        else if (delta == 0) {
            /* une solution double      */
            printf("Une solution double : %f\n", (-b / 2 / a));
        }
        else {
            /* pas de solutions réelles */
            printf("Pas de solutions réelles");
        }
    }
    printf("\n");

    return EXIT_SUCCESS;
}
```

Exercice 2 : Résolution numériquement correcte d'une équation du second degré
Utiliser les formules analytiques de l'exercice 1 pour calculer les solutions réelles de l'équation du second degré n'est pas numériquement satisfaisant. En effet, si la valeur de b est positive et proche de celle de $\sqrt{\Delta}$, le calcul de x_2 comportera au numérateur la soustraction de deux nombres voisins ce qui augmente le risque d'erreur numérique.

Par exemple, considèrons l'équation $x^2 + 62,10x + 1 = 0$ dont les solutions sont approximativement :

$$x_1 = -62,08390$$
$$x_2 = -0,01610723$$

Dans cette équation, la valeur de b^2 est bien plus grande que le produit $4ac$ et le calcul de Δ conduit donc à un nombre très proche de b. Dans les calculs qui suivent, on ne tient compte que des 4 premiers chiffres significatifs.

$$\sqrt{\Delta} = \sqrt{(62,10)^2 - 4 \times 1 \times 1} = \sqrt{3856 - 4} = 62,06$$

On obtient alors :

$$x_2 = \frac{-62,10 + 62,06}{2} = -0,02$$

L'erreur relative sur x_2 induite est importante :

$$\frac{|-0,01611 + 0,02|}{|-0,01611|} = 0,24 \quad \text{(soit 24\% d'erreur).}$$

En revanche, pour le calcul de x_1 l'addition de deux nombres pratiquement égaux ne pose pas de problème et les calculs conduisent à un erreur relative faible $(3,210^{-4})$.

Pour diminuer l'erreur numérique sur x_2, il suffirait de réorganiser les calculs :

$$x_2 = \left(\frac{-b + \sqrt{\Delta}}{2a}\right)\left(\frac{-b - \sqrt{\Delta}}{-b - \sqrt{\Delta}}\right) = \frac{b^2 - \Delta}{2a(-b - \sqrt{\Delta})} = \frac{-2c}{b + \sqrt{\Delta}}$$

Mais il existe une solution plus simple qui consiste à calculer d'abord x_1 par la formule classique puis en déduire x_2 en considérant que le produit des racines est égal à c/a ($x_1 x_2 = c/a$).
Conclusion : En pratique, si b est négatif, on calcule d'abord la racine $\frac{-b+\sqrt{\Delta}}{2a}$, si b est positif, on calcule la racine $\frac{-b-\sqrt{\Delta}}{2a}$. L'autre racine est calculée à l'aide du produit c/a.

Écrire le programme qui calcule les racines réelles de l'équation du second degré en s'appuyant sur cette formule.

Solution :

```c
/*****************************************************************
 *  Auteur   : Xavier Crégut <cregut@enseeiht.fr>
 *  Version  : 1.4
 *  Objectif : Résolution de l'équation du seond degré.
 *****************************************************************/

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

```c
#include <math.h>

/* Demande à l'utilisateur les coefficients a, b et c de l'équation du
 * second degré  ax^2 + bx + c* , calcule et affiche les solutions
 * réelles de cette équation.
 */
int main()
{
    /* Déclaration des variables (C oblige) */
    float a, b, c;      /* coefficients de l'équation   */

    /* Poser le problème */
    printf("Résolution de l'équation du second degré :\n");
    printf("\t    a.x2 + b.x +c = 0\n");

    /* Saisir la valeur des coefficients */
    printf("Donnez la valeur des coefficients a, b et c : ");
    scanf("%f%f%f", &a, &b, &c);

    /* Résoudre l'équation et afficher les résultats */
    if (a == 0) {       /*{ équation du premier degré }*/
        /* résoudre l'équation du premier degré */
        if (b == 0) {
            if (c == 0) {
                printf("Tout les réels sont solution");
            }
            else {
                printf("Pas de solution");
            }
        }
        else {
            printf("Une seule solution : %f", -c / b);
        }
    }
    else {
        /* cas général  */
        float delta = b*b - 4*a*c;              /* le discriminant */
        if (delta > 0) {
            /* deux solutions réelles   */
            /* Principe : commencer par calculer la racine la plus
             * grande en valeur absolue puis en déduire la seconde
             * en appliquant x1 * x2 == c / a
             */
            float x1, x2;       /* les deux racines */
            if (b > 0) {
                x1 = (- b - sqrt(delta)) / 2 / a;
                x2 = c / a / x1;
            }
            else {      /*{ b <= 0 }*/
                x2 = (- b + sqrt(delta)) / 2 / a;
                x1 = c / a / x2;
            }
```

```
61              printf("Deux␣solutions␣:\n");
62              printf("\tsolution␣1␣=␣%f\n", x1);
63              printf("\tsolution␣2␣=␣%f\n", x2);
64          }
65          else if (delta == 0) {
66              /* une solution double      */
67              printf("Une␣solution␣double␣:␣%f\n", (-b / 2 / a));
68          }
69          else {
70              /* pas de solutions réelles */
71              printf("Pas␣de␣solutions␣réelles");
72          }
73      }
74      printf("\n");
75
76      return EXIT_SUCCESS;
77  }
```

Exercice 3 : Le nombre d'occurrences du maximun

Compléter le programme de l'exercice 2 (Exercices résolus en C, Semaine 1) qui calcule les statistiques sur une série de valeurs réelles pour qu'il affiche également le nombre d'occurrences de la plus grande valeur, c'est-à-dire le nombre de valeurs de la série qui correspondent à la valeur maxiamle.

Par exemple, pour la série 1 2 3 1 2 3 3 2 3 1 0, le max est 3 et il y a quatre occurrences de 3. Dans la série 1 2 3 3 3 3 1 2 4 1 2 3 0, il y a une seule occurrence du maximun qui est 4.

Solution : Le principe est d'ajouter une nouvelle variable d'accumulation, `nb_max`, qui comptabilise le nombre d'occurrences du max rencontrées. Elle est initialisée à 1 sur la première valeur. Elle est remise à 1 dès qu'un nouveau maximum est identifié. Elle est augmentée de 1 si la valeur lue est égale au maximum précédent.

```c
/*****************************************************************************
 *  Auteur  :  Xavier Crégut <cregut@enseeiht.fr>
 *  Version :  1.1
 *
 *  Objectif :
 *      Afficher la moyenne, la plus grande et la plus petite valeur et le
 *      nombre d'occurences de la plus grande valeur d'une série de valeurs
 *      réelles lues au clavier.
 *
 *****************************************************************************/

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

int main()
{
    double x;           /* un réel lu au clavier */
    int nb;             /* le nombre de valeurs lues de la série */
    double somme;       /* la somme des valeurs lues de la série */
    double min;         /* la plus petite des valeurs lues de la série */
    double max;         /* la plus grande des valeurs lues de la série */
    double nb_max;      /* le nombre d'occurrences du max */

    /* afficher la consigne */
    printf("Donnez une série d'entiers qui se termine par 0.\n");

    /* lire le premier réel */
    scanf("%lf", &x);

    if (x == 0) {       /* Pas de valeur dans la séries */
        printf("La série est vide.  ");
        printf("Les statistiques demandées n'ont pas de sens !\n");
    }
    else {
        /* initialiser les variables statistiques avec x */
        max = x;
        min = x;
        somme = x;
```

```c
        nb = 1;
        nb_max = 1;

        /* lire une nouvelle valeur x */
        scanf("%lf", &x);

        while (x != 0) {
            /* mettre à jour les variables statistiques */
            nb++;
            somme += + x;
            if (x > max) {
                max = x;
                nb_max = 1;
            }
            else if (x == max) {
                nb_max++;
            }
            else if (x < min) {
                min = x;
            }

            /* lire une nouvelle valeur x */
            scanf("%lf", &x);
        }

        /* afficher les statistiques */
        printf("Moyenne = %f\n", somme / nb);
        printf("Plus petite valeur = %f\n", min);
        printf("Plus grande valeur = %f\n", max);
        printf("Nb d'occurrences du max = %f\n", nb_max);
    }

    return EXIT_SUCCESS;
}

/*                      moyenne, min, max,      nb_max
    1 2 3 0     -->     2,      1,      3       1
    2 -2 0      -->     0,      -2,     2       1
    -4 -2 0     -->     -3,     -4,     -2      1
    13 0        -->     13,     13,     13      1
    0           -->     Non défini
    1 3 1 3 0   -->     2       1       3       2
    3 3 3 3 0   -->     3       3       3       4
*/
```

Exercice 4 : Nombres amis

Deux nombres N et M sont amis si la somme des diviseurs de M (en excluant M lui-même) est égale à N et la somme des diviseurs de N (en excluant N lui-même) est égale à M.

Écrire un programme qui affiche tous les couples (N, M) de nombres amis tels que $0 < N < M \leq MAX$, MAX étant lu au clavier.

Indication : Les nombres amis compris entre 1 et 100000 sont (220, 284), (1184, 1210), (2620, 2924), (5020, 5564), (6232, 6368), (10744, 10856), (12285, 14595), (17296, 18416), (66928, 66992), (67095, 71145), (63020, 76084), (69615, 87633) et (79750, 88730).

Solution : L'idée est de faire un parcours de tous les couples possibles et de se demander s'ils forment un couple de nombres amis ou non. Pour un n et un m donnés, il s'agit alors de calculer la somme de leurs diviseurs. C'est en fait deux fois le même problème. Il s'agit de calculer la somme des diviseurs d'un entier p. Naïvement, on peut regarder si les entiers de 2 à $p-1$ sont des diviseurs de p.

Une première optimisation consiste à remarquer que p n'a pas de diviseurs au delà de $p/2$. En fait, il est plus intéressant de remarquer que si i est diviseur de p alors p/i est également un diviseur de p. Il suffit donc de ne considérer comme diviseurs potentiels que les entiers compris dans l'intervalle $2..\sqrt{p}$. Il faut toutefois faire attention à ne pas comptabiliser deux fois $\sqrt{p}$.

```
1  R0 : Afficher les couples de nombres amis (N, M) avec 1 < N < M <= Max
2
3  R1 : Raffinage De « R0 »
4       | Pour m <- 2 JusquÀ m = Max Faire
5       |    | Pour n <- 2 JusquÀ n = m - 1 Faire
6       |    |    | Si n et m amis Alors
7       |    |    |    | Afficher le couple (n, m)
8       |    |    | FinSi
9       |    | FinPour
10      | FinPour
11
12 R2 : Raffinage De « n et m amis »
13      | Résultat <- (somme des diviseurs de N) = M
14      |          Et (somme des diviseurs de M) = N
15
16 R3 : Raffinage De « somme des diviseurs de p »
17      | somme <- 1
18      | Pour i <- 2 JusquÀ i = racine_carrée(p) - 1 Faire
19      |    | Si i diviseur de p Alors
20      |    |    | somme <- somme + i + (p Div i)
21      |    | FinSi
22      | FinPour
23      | Si p est un carré parfait Alors
24      |    | somme <- somme + i
25      | FinSi
```

L'algorithme est alors le suivant. Notez quelques optimisations qui font que l'algorithme de respecte pas complètement le raffinage.

```
1  Algorithme nb_amis
2
```

```
   -- Afficher les couples de nombres amis (N, M) avec 1 < N < M <= MAX

Variables
    n, m: Entier        -- pour représenter les couples possibles
    somme_n: Entier     -- somme des diviseurs de n
    somme_m: Entier     -- somme des diviseurs de m

Début
    Pour m <- 2 JusquÀ m = MAX Faire
        -- calculer la somme des divieurs de m
        -- Remarque : on peut déplacer cette étape à l'extérieur de la
        -- boucle car elle ne dépend pas de n (optimisation).
        somme_m <- 1
        Pour i <- 2 JusquÀ i = racine_carrée(m) - 1 Faire
            Si i diviseur de m Alors
                somme_m <- somme_m + i + (m Div i)
            FinSi
        FinPour
        Si m est un carré parfait Alors
            somme_m <- somme_m + i
        FinSi


        Pour n <- 2 JusquÀ n = m - 1 Faire
            -- calculer la somme des divieurs de n
            somme_n <- 1
            Pour i <- 2 JusquÀ i = racine_carrée(n) - 1 Faire
                Si i diviseur de n Alors
                    somme_n <- somme_n + i + (n Div i)
                FinSi
            FinPour
            Si n est un carré parfait Alors
                somme_n <- somme_n + i
            FinSi

            -- déterminer si n et m sont amis
            Si (somme_n = m) Et (somme_m = n) Alors { n et m sont amis }
                Afficher le couple (n, m)
            FinSi
        FinPour
    FinPour

Fin.
```

```
/*****************************************************************
 *  Auteur   : Xavier Crégut <cregut@enseeiht.fr>
 *  Version  : 1.4
 *
 *  Objectif : Afficher les couples de nombres amis (N, M)
 *      avec 1 < N < M <= MAX.
 *****************************************************************/

```

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

#define MAX 1000000

int main()
{
    int n, m;           /* pour représenter les couples possibles */
    int somme_n;        /* somme des diviseurs de n */
    int somme_m;        /* somme des diviseurs de m */
    int i;              /* varaible de boucle */

    for (m = 2; m <= MAX; m++) {
        /* calculer la somme des divieurs de m */
        /* Remarque : on peut déplacer cette étape à l'extérieur de la boucle
        ** car elle ne dépend pas de n (optimisation). */
        somme_m = 1;
        i = 2;
        while (i < m / i) {
            if (m % i == 0) {   /* i diviseur de m */
                somme_m = somme_m + i + (m / i);
            }
            i++;
        }
        if (m == i * i) {       /* m est un carré parfait */
            somme_m = somme_m + i;
        }

        for (n = 2; n < m; n++) {
            /* calculer la somme des divieurs de n */
            somme_n = 1;
            i = 2;
            while (i < n / i) {
                if (n % i == 0) {       /*  i diviseur de n */
                    somme_n = somme_n + i + (n / i);
                }
                i++;
            }
            if (n == i * i) {   /* n est un carré parfait */
                somme_n = somme_n + i;
            }

            /* déterminer si n et m sont amis */
            if (somme_n == m && somme_m == n) { /* n et m sont amis */
                printf("(%d, %d)\n", n, m);
            }
        }
    }

    return EXIT_SUCCESS;
}
```

Une solution plus efficace consiste à constater que pour un entier M compris entre 2 et MAX, le seul nombre ami possible est la somme de ses diviseurs que l'on note `somme_m`. Il reste alors à vérifier si la somme des diviseurs de `somme_m` est égale à M pour savoir si `somme_m` et M sont amis. Le fait de devoir afficher les couples dans l'ordre croissant nous conduit à ne considérer que les sommes de diviseurs inférieures à M.

À titre indicatif, pour Max = 1000000, ce deuxième algorithme termine en 1 minute 23 alors que dans le même temps, seuls les sept premiers résultats sont trouvés avec le premier algorithme. Les solutions suivantes sont trouvées au bout d'une minute 32 secondes, 2 minutes 46, 5 minutes 35, 20 minutes, etc.

```
1  R0 : Afficher les couples de nombres parfaits (N, M) avec 1 < N < M <= Max
2
3  R1 : Raffinage De « R0 »
4        | Pour m <- 2 JusquÀ m = Max Faire
5        |   | Déterminer la somme des diviseurs de m
6        |   |                                      m: in ; somme_m: out Entier
7        |   | n <- somme_m
8        |   | Si n < m Alors       { n est un nb amis potentiel }
9        |   |   | Déterminer la somme des diviseurs de n (somme_n)
10       |   |   | Si somme_n = m Alors      { somme_m et m amis }
11       |   |   |   | Afficher le couple (n, m)
12       |   |   | FinSi
13       |   | FinPour
14       | FinPour
15
16 R2 : Raffinage De « somme des diviseurs de p »
17       | somme <- 1
18       | Pour i <- 2 JusquÀ i = racine_carrée(p) - 1 Faire
19       |   | Si i diviseur de p Alors
20       |   |   | somme <- somme + i + (p Div i)
21       |   | FinSi
22       | FinPour
23       | Si p est un carré parfait Alors
24       |   | somme <- somme + i
25       | FinSi
```

Voici l'algorithme correspondant.

```
1  Algorithme nb_amis
2
3     -- Afficher les couples de nombres amis (N, M) avec 1 < N < M <= MAX
4
5  Variables
6      m: Entier           -- pour parcourir les entiers de 2 à MAX
7      n: Entier           -- l'amis candidat
8      somme_m: Entier     -- somme des diviseurs de m
9      somme_n: Entier     -- somme des diviseurs de somme_m correspondant à n
10
11 Début
12     Pour m <- 2 JusquÀ m = MAX Faire
13         -- Déterminer la somme des divieurs de m
```

```
14          somme_m <- 1
15          Pour i <- 2 JusquÀ i = racine_carrée(m) - 1 Faire
16              Si i diviseur de m Alors
17                  somme_m <- somme_m + i + (m Div i)
18              FinSi
19          FinPour
20          Si m est un carré parfait Alors
21              somme_m <- somme_m + i
22          FinSi
23
24          { somme_m est la candidat pour n }
25
26          n <- somme_m
27          Si n < m Alors  { on s'intéresse au cas  n <= m }
28              -- Déterminer la somme des diviseurs de n (somme_n)
29              somme_n <- 1
30              Pour i <- 2 JusquÀ i = racine_carrée(n) - 1 Faire
31                  Si i diviseur de n Alors
32                      somme_n <- somme_n + i + (n Div i)
33                  FinSi
34              FinPour
35              Si n est un carré parfait Alors
36                  somme_n <- somme_n + i
37              FinSi
38
39
40              Si somme_n = m Alors        { somme_m et m amis }
41                  Afficher le couple (somme_m, m)
42              FinSi
43          FinPour
44
45      FinPour
46  Fin.
```

```
1   /*****************************************************************************
2    *  Auteur   : Xavier Crégut <cregut@enseeiht.fr>
3    *  Version  : 1.4
4    *
5    *  Objectif : Afficher les couples de nombres amis (N, M)
6    *      avec 1 < N < M <= MAX.
7    *****************************************************************************/
8
9   #include <stdio.h>
10  #include <stdlib.h>
11  #include <math.h>
12
13  #define MAX 1000000
14
15  int main()
16  {
17      int m;              /* parcourir les entiers de 2 à MAX */
18      int n;              /* l'amis possible de m */
```

```c
    int somme_m;        /* somme des diviseurs de m */
    int somme_n;        /* somme des diviseurs de somme_m correspondant à n */
    int i;              /* variable de boucle */

    for (m = 2; m <= MAX; m++) {
        /* Déterminer la somme des divieurs de m */
        somme_m = 1;
        i = 2;
        while (i < m / i) {
            if (m % i == 0) {   /* i diviseur de m */
                somme_m = somme_m + i + (m / i);
            }
            i++;
        }
        if (m == i * i) {       /* m est un carré parfait */
            somme_m = somme_m + i;
        }

        n = somme_m;
        if (n < m) {    /* on s'intéresse au cas  n <= m */
            /* Déterminer la somme des diviseurs de n (somme_n) */
            somme_n = 1;
            i = 2;
            while (i < n / i) {
                if (n % i == 0) {       /*  i diviseur de n */
                    somme_n = somme_n + i + (n / i);
                }
                i++;
            }
            if (n == i * i) {   /* n est un carré parfait */
                somme_n = somme_n + i;
            }

            /* déterminer si n et m sont amis */
            if (somme_n == m) {         /* n et m sont amis */
                printf("(%d, %d)\n", n, m);
            }
        }
    }

    return EXIT_SUCCESS;
}

/*  Remarque : pour la structuration du programme, il serait préférable
 *  d'utiliser un sous-programme (une fonction) qui calcule la somme des
 *  diviseurs d'un entier n.
 */
```

Exercice 5 : Puissance

Calculer et afficher la puissance entière d'un réel.

Solution : On peut, dans un premier temps, se demander si la puissance entière n d'un réel x a toujours un sens. En fait, ceci n'a pas de sens si x est nul et si n est strictement négatif. D'autre part, le cas $x = 0$, $n = 0$ est indéterminé. On choisit de contrôler la saisie de manière à garantir que nous ne sommes pas dans l'un de ces cas. Nous souhaitons donner à l'utilisateur un message le plus clair possible lorsque la saisie est invalide.

Nous envisageons les jeux de tests suivants :

```
1     x    n  -->   x^n
2
3     2    3  -->     8      -- cas nominal
4     3    2  -->     9      -- cas nominal
5     1    1  -->     1      -- cas nominal
6     2   -3  --> 0.125      -- cas nominal (puissance négative)
7     0    2  -->     0      -- cas nominal (x est nul)
8     0   -2  --> ERREUR     -- division par zéro
9     0    0  --> Indéterminé
10  1.5    2  --> 2.25       -- x peut être réel
```

Voyons maintenant le principe de la solution. Par exemple, pour calculer 2^3, on peut faire $2 * 2 * 2$. Plus généralement, on multiplie n fois x par lui-même (donc une boucle **Pour**).

Si on essaie ce principe, on constate qu'il ne fonctionne pas dans le cas d'une puissance négative. Prenons le cas de 2^{-3}. On peut le réécrire $(1/2)^3$. On est donc ramené au cas précédent. Le facteur multiplicatif est dans ce cas $1/x$ et le nombre de fois est $-n$.

Nous introduisons donc deux variables intermédiaires qui sont :

```
facteur: Réel    -- facteur multiplicatif pour obtenir les puissances
                 -- successives de x
puissance: Réel -- abs(n).  On a : facteur^puissance = x^n
```

On peut maintenant formaliser notre raisonnement sous la forme du raffinage suivant.

```
1  R0 : Afficher la puissance entière d'un réel
2
3  R1 : Raffinage De « R0 »
4       | Saisir avec contrôle les valeurs de x et n      x: out Réel; n: out Entier
5       | { (x <> 0) Ou (n > 0) }
6       | Calculer x à la puisssance n                    x, n: in ; xn: out Réel
7       | Afficher le résultat                            xn: in Réel
8
9  R2 : Raffinage De « Saisir avec contrôle les valeurs de x et n »
10      | Répéter
11      |   | Saisir la valeur de x et n          x: out Réel; n: out Entier
12      |   | Contrôler x et n                    x, n: in ; valide: out Booléen
13      | JusquÀ valide                           valide: in
14
15 R1 : Raffinage De « Calculer x à la puisssance n »
16      | Si x = 0 Alors
17      |   | xn := 0
18      | Sinon
```

```
19      |   | Déterminer le facteur multiplicatif et la puissance
20      |   |                     x, n: in ;
21      |   |                     facteur out Réel ;
22      |   |                     puissance: out Entier
23      |   | Calculer xn par itération (accumulation)
24      |   |                     n, facteur, puissance: in ; xn : out
25      | FinSi
26
27  R3 : Raffinage De « Calculer xn par itération (accumulation) »
28      | xn <- 1;
29      | Pour i <- 1 JusquÀ i = puissance Faire
30      |   | { Invariant : xn = facteur^i }
31      |   | xn <- xn * facteur
32      | FinPour
```

On peut alors en déduire l'algorithme suivant :

```
1   Algorithme puissance
2
3       -- Afficher la puissance entière d'un réel
4
5   Variables
6       x: Réel             -- valeur réelle lue au clavier
7       n: Entier           -- valeur entière lue au clavier
8       valide: Booléen     -- x^n peut-elle être calculée
9       xn: Réel            -- x à la puissance n
10      facteur: Réel       -- facteur multiplicatif pour obtenir
11                          -- les puissances successives
12      puissance: Entier;  - abs(n).  On a facteur^puissance = x^n
13      i: Entier           -- variable de boucle
14
15  Début
16      -- Saisir avec contrôle les valeurs de x et n
17      Répéter
18          -- saisir la valeur de x et n
19          Écrire("x = ")
20          Lire(x)
21          Écrire("n = ")
22          Lire(n)
23
24          -- contrôler x et n
25          valide <- VRAI
26          Si x = 0 Alors
27              Si n = 0 Alors
28                  ÉcrireLn("x et n sont nuls. x^n est indéterminée.")
29                  valide <- FAUX
30              SinonSi n < 0 Alors
31                  ÉcrireLn("x nul et n négatif. x^n n'a pas de sens.")
32                  valide <- FAUX
33              FinSi
34          FinSi
35
```

```
36          Si Non valide Alors
37              ÉcrireLn("␣␣Recommencez␣!")
38          FinSi
39      JusquÀ valide
40
41      -- Calculer x à la puisssance n
42      Si x = 0 Alors      -- cas trivial
43          xn <- 0
44      Sinon
45          -- Déterminer le facteur multiplicatif et la puissance
46          Si n >= 0 Alors
47              facteur <- x
48              puissance <- n
49          Sinon
50              facteur <- 1/x
51              puissance <- -n
52          FinSi
53
54          -- Calculer xn par itération (accumulation)
55          xn <- 1;
56          Pour i <- 1 JusquÀ i = puissance Faire
57              { Invariant : xn = facteur^i}
58              xn <- xn * facteur
59          FinPour
60      FinSi
61
62      -- Afficher le résultat
63      ÉcrireLn(x, "^", n, "␣=␣", xn)
64  Fin.
```

```
1   /*****************************************************************
2    *  Auteur   : Xavier Crégut <cregut@enseeiht.fr>
3    *  Version  : 1.1
4    *  Objectif : Afficher la puissance entière d'un réel
5    *****************************************************************/
6
7   #include <stdio.h>
8   #include <stdlib.h>
9   #include <stdbool.h>
10
11  int main()
12  {
13      double x;           /* valeur réelle lue au clavier */
14      int n;              /* valeur entière lue au clavier */
15      bool valide;        /* x^n peut-elle être calculée */
16      double xn;          /* x à la puissance n */
17      double facteur;     /* facteur multiplicatif pour
18                           * obtenir les puissances successives
19                           */
20      int puissance;      /* abs(n).  On a : facteur^puissance == x^n */
21      int i;              /* variable de boucle */
```

```
22
23      /* Saisir avec contrôle les valeurs de x et n */
24      do {
25          /* saisir la valeur de x et n */
26          printf("x = ");
27          scanf("%lf", &x);
28          printf("n = ");
29          scanf("%d", &n);
30
31          /* contrôler x et n */
32          valide = true;
33          if (x == 0) {
34              if (n == 0) {
35                  printf("x et n sont nuls. x^n est indéterminée.");
36                  valide = false;
37              }
38              else if (n < 0) {
39                  printf("x nul et n négatif. x^n n'a pas de sens.");
40                  valide = false;
41              }
42          }
43          if (!valide) {
44              printf("  Recommencez !\n");
45          }
46      } while (!valide);
47
48      /* Calculer x à la puisssance n */
49      if (x == 0) {       /* cas trivial */
50          xn = 0;
51      }
52      else {
53          /* Déterminer le facteur multiplicatif et la puissance */
54          if (n >= 0) {
55              facteur = x;
56              puissance = n;
57          }
58          else {
59              facteur = 1/x;
60              puissance = -n;
61          }
62
63          /* Calculer xn par itération (accumulation) */
64          xn = 1;
65          for (i = 1; i <= puissance; i++) {
66              /*{ Invariant : xn = facteur^{*i} }*/
67              xn = xn * facteur;
68          }
69      }
70
71      /* Afficher le résultat */
72      printf("%f^%d = %f\n", x, n, xn);
73
```

```
74      return EXIT_SUCCESS;
75  }
```

Exercice 6 : Amélioration du calcul de la puissance entière

Améliorer l'algorithme de calcul de la puissance (exercice 5 du Exercices corrigés en C, Semaine 1) en remarquant que

$$x^n = \begin{cases} (x^2)^p & \text{si } n = 2p \\ (x^2)^p \times x & \text{si } n = 2p+1 \end{cases}$$

Ainsi, pour calculer 3^5, on peut faire $3 * 9 * 9$ avec bien sûr $9 = 3^2$.

Solution : Nous nous appuyons sur les mêmes variables que pour le calcul « naturel » de la puissance (exercice 5). Nous allons continuer à calculer x^n par accumulation. Nous avons l'invariant suivant :

$$x^n = \text{xn} * \text{facteur}^{\text{puissance}}$$

Lors de l'initialisation, cet invariant est vrai (`xn` vaut 1 et facteur et puissance sont tels qu'ils valent x^n).

D'après la formule donnée dans l'énoncé, à chaque itération de la boucle, deux cas sont à envisager :

– soit `puissance` est paire. On peut alors l'écrire $2 * p$. On a alors :

$$\begin{aligned} x^n &= \text{xn} * \text{facteur}^{\text{puissance}} \\ &= \text{xn} * \text{facteur}^{(2\ *\ \text{p})} \\ &= \text{xn} * (\text{facteur}^2)^{\text{p}} \end{aligned}$$

On peut donc faire :

```
facteur <- facteur * facteur
puissance <- puissance Div 2    -- car p = puissance Div 2
```

On constate que l'invariant est préservé d'après les égalités ci-dessus et la puissance a strictement diminué (car divisée par 2).

– soit `puissance` est impaire. On peut alors l'écrire $2 * p + 1$. On a alors :

$$\begin{aligned} x^n &= \text{xn} * \text{facteur}^{\text{puissance}} \\ &= \text{xn} * \text{facteur}^{(2\ *\ \text{p}\ +\ 1)} \\ &= \text{xn} * (\text{facteur} * \text{facteur}^{(2*\text{p})}) \\ &= (\text{xn} * \text{facteur}) * \text{facteur}^{(2*\text{p})} \end{aligned}$$

On peut donc faire :

```
xn <- xn * facteur
puissance <- puissance - 1
```

On constate que l'invariant est préservé d'après les égalités ci-dessus et la puissance a strictement diminué (car diminuée de 1).

On peut alors en déduire le raffinage suivant :

```
1 R3 : Raffinage De « Calculer xn par itération (accumulation) »
2       | xn <- 1;
3       | TantQue puissance > 0 Faire
```

```
4         |   | { Variant : puissance }
5         |   | { Invariant : x^n = xn * facteur^puissance }
6         |   | Si puissance Div 2 = 0 Alors        { puissance = 2 * p }
7         |   |  | puissance <- puissance Div 2
8         |   |  | facteur <- facteur * facteur
9         |   | Sinon                               { puissance = 2 * p + 1 }
10        |   |  | puissance <- puissance - 1
11        |   |  | xn <- xn * facteur
12        |   | FinSi
13        | FinTQ
```

```c
/*****************************************************************
 *  Auteur   : Xavier Crégut <cregut@enseeiht.fr>
 *  Version  : 1.1
 *  Objectif : Afficher la puissance entière d'un réel
 *****************************************************************/

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

int main()
{
    double x;           /* valeur réelle lue au clavier */
    int n;              /* valeur entière lue au clavier */
    bool valide;        /* x^n peut-elle être calculée */
    double xn;          /* x à la puissance n */
    double facteur;     /* facteur multiplicatif pour
                         * obtenir les puissances successives
                         */
    int puissance;      /* abs(n).  On a : facteur^puissance == x^n */

    /* Saisir avec contrôle les valeurs de x et n */
    do {
        /* saisir la valeur de x et n */
        printf("x = ");
        scanf("%lf", &x);
        printf("n = ");
        scanf("%d", &n);

        /* contrôler x et n */
        valide = true;
        if (x == 0) {
            if (n == 0) {
                printf("x et n sont nuls. x^n est indéterminée.");
                valide = false;
            }
            else if (n < 0) {
                printf("x nul et n négatif. x^n n'a pas de sens.");
                valide = false;
            }
        }
```

```c
        if (!valide) {
            printf("  Recommencez !\n");
        }
    } while (!valide);

    /* Calculer x à la puisssance n */
    if (x == 0) {       /* cas trivial */
        xn = 0;
    }
    else {
        /* Déterminer le facteur multiplicatif et la puissance */
        if (n >= 0) {
            facteur = x;
            puissance = n;
        }
        else {
            facteur = 1/x;
            puissance = -n;
        }

        /* Calculer xn par itération (accumulation) */
        xn = 1;
        while (puissance > 0) {
            /*{ Invariant : xn * facteur ^ puissance == x ^ n }*/
            /*{ Variant : puissance }*/
            if (puissance % 2 == 0) {
                facteur = facteur * facteur;
                puissance = puissance / 2;
            }
            else {
                xn = xn * facteur;
                puissance = puissance - 1;
            }
        }
    }

    /* Afficher le résultat */
    printf("%f^%d = %f\n", x, n, xn);

    return EXIT_SUCCESS;
}
```

Exercice 7 : Nombres de Amstrong

Les *nombres de Amstrong* appelés parfois *nombres cubes* sont des nombres entiers qui ont la particularité d'être égaux à la somme des cubes de leurs chiffres. Par exemple, 153 est un nombre de Amstrong car on a :

$$153 = 1^3 + 5^3 + 3^3.$$

Afficher tous les nombres de Amstrong sachant qu'ils sont tous compris entre 100 et 499.

Indication : Les nombres de Amstrong sont : 153, 370, 371 et 407.

Solution :

```
R0 : Afficher les nombres de Amstrong compris entre 100 et 499.
```

On peut envisager (au moins) deux solutions pour résoudre ce problème.

Solution 1. La première solution consiste à essayer les combinaisons de trois chiffres qui vérifient la propriété. On prend alors trois compteurs : un pour les unités, un pour les dizaines et un pour les centaines. Les deux premiers varient de 0 à 9. Le dernier de 1 à 4. Ayant les trois chiffres, on peut calculer la somme de leurs cubes puis le nombre qu'ils forment et on regarde si les deux sont égaux.

Ceci se formalise dans le raffinage suivant :

```
R1 : Raffinage De « Afficher les nombres de Amstrong »
    Pour centaine <- 1 JusquÀ centaine = 4 Faire
        Pour dizaine <- 1 JusquÀ dizaine = 9 Faire
            Pour unité <- 1 JusquÀ unité = 9 Faire
                Déterminer le cube
                Déterminer le nombre
                Si nombre = cube Alors
                    Afficher nombre
                FinSi
            FinPour
        FinPour
    FinPour
```

On en déduit alors le programme C suivant.

```c
/*****************************************************************
 *  Auteur   : Xavier Crégut <cregut@enseeiht.fr>
 *  Version  : 1.1
 *  Objectif : Afficher les nombres de Amstrong
 *****************************************************************/

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

int main()
    /* Principe : on parcourt tous les trois chiffres du nombre
     * compris entre 100 et 499.
     */
{
    int nb;             /* le nombre considére */
```

```c
    int centaine, dizaine, unite;       /* les trois chiffres de nb */
    int cube;           /* le somme des cubes des trois chiffres */

    for (centaine = 1; centaine <= 4; centaine++) {
        for (dizaine = 0; dizaine <= 9; dizaine++) {
            for (unite = 0; unite <= 9; unite++) {
                /* déterminer la somme des cubes */
                cube = centaine * centaine * centaine
                        + dizaine * dizaine * dizaine
                        + unite * unite * unite;

                /* déterminer le nombre */
                nb = centaine * 100 + dizaine * 10 + unite;

                if (nb == cube) {       /* c'est un nombre de Amstrong */
                    printf("%d\n", nb);
                }
            }
        }
    }

    return EXIT_SUCCESS;
}
```

On remarque que les opérations peuvent être réoganisées pour augmenter les performances en temps de calcul. En particulier, il est inutile de faire des calculs à l'intérieure d'une boucle s'ils ne dépendent pas de la boucle. Ainsi, le calcul du cube des centaines peut se faire dans la boucle la plus externe au lieu de le faire dans la plus interne.

On arrive alors à une nouvelle version du programme.

```c
/*****************************************************************
 *  Auteur   : Xavier Crégut <cregut@enseeiht.fr>
 *  Version  : 1.1
 *  Objectif : Afficher les nombres de Amstrong
 *****************************************************************/

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

int main()
    /* Principe : on parcourt tous les trois chiffres du nombre
     * compris entre 100 et 499.
     * On sort de la boucle interne certains calculs (centaine3,
     * dizaine3)
     */
{
    int nb;             /* le nombre considére */
    int centaine, dizaine, unite;       /* les trois chiffres de nb */
    int centaine3, dizaine3, unite3;    /* le cube des trois chiffres */
    int cube;           /* le somme des cubes des trois chiffres */

    for (centaine = 1; centaine <= 4; centaine++) {
```

```c
        centaine3 = centaine * centaine * centaine;
        for (dizaine = 0; dizaine <= 9; dizaine++) {
            dizaine3 = dizaine * dizaine * dizaine;
            for (unite = 0; unite <= 9; unite++) {
                unite3 = unite * unite * unite;

                /* déterminer la somme des cubes */
                cube = centaine3 + dizaine3 + unite3;

                /* déterminer le nombre */
                nb = centaine * 100 + dizaine * 10 + unite;

                if (nb == cube) {       /* c'est un nombre de Amstrong */
                    printf("%d\n", nb);
                }
            }
        }
    }

    return EXIT_SUCCESS;
}
```

Solution 2. La deuxième solution consiste à parcourir tous les entiers compris entre 100 et 499 et à regarder s'ils sont égaux à la somme des cubes de leurs chiffres. La difficulté est alors d'extraite les chiffres. L'idée est d'utiliser la division entière par 10 et son reste.

On obtient le raffinage suivant.

```
R1 : Raffinage De « Afficher les nombres de Amstrong »
    Pour nombre <- 100 JusquÀ centaine = 499 Faire
        Déterminer les chiffres de nb
        Déterminer le cube des chiffres
        Si nombre = cube Alors
            Afficher nombre
        FinSi
    FinPour

R2 : Raffinage De « Déterminer les chiffres de nb »
    unité <- nombre Mod 10
    dizaine <- (nombre Div 10) Mod 10
    centaine <- nombre Div 100
```

On en déduit alors le programme C suivant.

```c
/*****************************************************************
 *  Auteur   : Xavier Crégut <cregut@enseeiht.fr>
 *  Version  : 1.1
 *  Objectif : Afficher les nombres de Amstrong
 *****************************************************************/

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

```c

int main()
    /* Principe : on parcourt tous les entiers de 100 à 499 et on
     * vérifie s'il s'agit d'un nombre de Amstrong.
     */
{
    int nb;             /* le nombre considére */
    int centaine, dizaine, unite;       /* les trois chiffres de nb */
    int cube;           /* le somme des cubes des trois chiffres */

    for (nb = 100; nb <= 499; nb++) {
        /* déterminer les chiffres de nb */
        unite = nb % 10;
        dizaine = (nb / 10) % 10;
        centaine = nb / 100;

        /* déterminer le cube des chiffres */
        cube = centaine * centaine * centaine
                + dizaine * dizaine * dizaine
                + unite * unite * unite;

        if (nb == cube) {       /* c'est un nombre de Amstrong */
            printf("%d\n", nb);
        }
    }

    return EXIT_SUCCESS;
}
```