

# TP2: Correction

MP2I Lycée Pierre de Fermat  
guillaume.rousseau@ens-lyon.fr

## Consignes

### Exercice 2

Q1. Code :

```
1  #include <stdio.h>
2  #include <stdbool.h>
3
4
5  int main(){
6      bool a = true, b = false;
7
8      bool a_et_b = a && b;
9      bool a_ou_b = a || b;
10     bool non_a = !a;
11     /* Pour afficher un booléen, on peut utiliser %d,
12        ce qui affichera 0 ou 1 */
13     printf("a = %d\n", a);
14     printf("b = %d\n", b);
15     printf("a && b = %d\n", a_et_b);
16     printf("a || b = %d\n", a_ou_b);
17     printf("!a = %d\n", non_a);
18
19     return 0;
20 }
```

**Q2.** On peut écrire un programme qui génère les tables pour nous (en utilisant des boucles pour éviter les répétitions) :

```

1  #include <stdio.h>
2  #include <stdbool.h>
3
4
5  /*
6  Afficher les tables de valeurs des opérateurs && et ||
7  */
8
9  // on peut utiliser les boucles pour simplifier le code
10 // et éviter les répétitions
11 int main(){
12     printf("ET:\n");
13     for(int x = 0; x < 2; x++){
14         for(int y = 0; y < 2; y++){
15             printf("x=%d, y=%d, x&& y=%d\n", x, y, x&&y);
16         }
17     }
18
19     printf("\n");
20
21     printf("OU:\n");
22     for(int x = 0; x < 2; x++){
23         for(int y = 0; y < 2; y++){
24             printf("x=%d, y=%d, x||y=%d\n", x, y, x||y);
25         }
26     }
27
28     return 0;
29 }
```

On en déduit les tables de valeurs :

$x$	$y$	$x \& y$
0	0	0
0	1	0
1	0	0
1	1	1

$x$	$y$	$x    y$
0	0	0
0	1	1
1	0	1
1	1	1

Notons qu’une disjonction (un “OU”) est vraie si au moins l’une des conditions est vraie. On dit que c’est un **ou inclusif** : les deux conditions peuvent être vraies en même temps. Il existe aussi un ou exclusif, qui n’est vrai que si exactement l’une des deux conditions l’est.

### Exercice 3

Les fonctions demandées, et le main pour les tester :

```
1  #include <stdio.h>
2  #include <stdbool.h>
3  #include <assert.h>
4
5
6  // 1
7  /* Renvoie true si a divise b, false sinon.
8     a doit être non-nul */
9  bool divise(int a, int b){
10     assert(a != 0);
11     return (b % a == 0);
12 }
13
14
15 // 2
16 /* Affiche x puis son inverse. x doit être non-nul */
17 void affiche_inverse(float x){
18     assert(x != 0);
19     printf("x vaut %f, son inverse vaut %f\n", x, 1/x);
20 }
21
22
23 // 3
24 /* Calcule  $A = 3x + 5y - 6.25z + t$ .
25     Affiche A, et renvoie le carré de A */
26 float lineaire(float x, float y, float z, float t){
27     float A = 3*x + 5*y - 6.25*z + t;
28     printf("%f\n", A);
29     return A*A;
30 }
31
32 int main(){
33     // 1
34     assert(divise(9, 45)); // assert sert aussi à valider des tests !
35     assert(divise(12, 144));
36     assert(divise(7, 0));
37     assert(!divise(5, 14));
38     assert(!divise(15, 5));
39
40     // 2
41     affiche_inverse(0.4);
42     affiche_inverse(3);
43     affiche_inverse(1);
44
45     // 3
46     assert(lineaire(0, 0, 0, 0) == 0);
47     assert(lineaire(1, 2, 2, 1.5) == 4);
48     assert(lineaire(2, 0, 1, 0) == 0.0625);
49
50     printf("Fin des tests\n");
51     return 0;
52 }
```

**A retenir** : il y a une différence entre **afficher** et **renvoyer/retourner**. Lorsqu'une fonction calcule un résultat, si elle ne fait que l'afficher, ce résultat est perdu dans les abysses, il est impossible de le récupérer et de continuer à l'utiliser dans la suite du programme.

**Assertions** : les assertions peuvent servir à s'assurer que les entrées d'une fonction sont valides, qu'elles vérifient certaines hypothèses. On ne peut pas vérifier de condition sur le type d'une variable (et on n'en a pas besoin : un int est toujours un int). Les assertions peuvent aussi servir pour faire des tests, comme montré dans le code ci-dessus ! Faire des tests régulièrement, et les laisser dans le main, ou dans une fonction `test` à part, est une bonne pratique de programmation, qui vous aidera à détecter tôt certains bugs et vous fera gagner du temps.

## Exercice 6

Q1. Code :

```
1  #include <stdio.h>
2  #include <stdbool.h>
3
4  int main(){
5      int n;
6      printf("Entrez un entier:\n");
7      scanf("%d", &n);
8
9      for (int i = 1; i <= n; ++i){
10         printf("%d\n", i);
11     }
12
13     return 0;
14 }
```

Q2. Code :

```
1  #include <stdio.h>
2  #include <stdbool.h>
3  #include <assert.h>
4
5  /* Affiche la k-ème ligne de l'escalier, composée
6     de 2k+1 '-' puis un '|' */
7  void affiche_ligne(int k){
8      assert(k >= 0);
9      for (int i = 0; i < 2*k+1; ++i){
10         printf("-");
11     }
12     printf("|\\n");
13 }
14
15 /* Affiche un escalier de n marches.
16 Exemple: escalier(5) affiche:
17 -|
18 ---|
19 ----|
20 -----|
21 -----|
22 */
23 void affiche_escalier(int n){
24     assert(n >= 0);
25     for (int i = 0; i < n; ++i){
26         affiche_ligne(i);
27     }
28 }
29
30 int main(){
31     int n;
32     printf("Entrez un entier:\n");
33     scanf("%d", &n);
34     affiche_escalier(n);
35
36     return 0;
37 }
```

## Exercice 7: (à rendre)

Le code pour cet exercice :

```
1 #include <stdio.h>
2 #include <stdbool.h>
3
4 int main(){
5     // on peut initialiser n à 0 pour forcer à rentrer
6     // dans la boucle une première fois
7     int n = 0;
8     int somme = 0;
9     while (n >= 0){
10         printf("Entrez un entier positif (-1 pour sortir):\n");
11         scanf("%d", &n);
12         if (n >= 0){
13             somme += n; // équivalent à somme = somme + n
14         }
15     }
16     printf("La somme vaut %d\n", somme);
17
18     return 0;
19 }
```

## Exercice 8

Code :

```
1 #include <stdio.h>
2 #include <stdbool.h>
3 #include <assert.h>
4 #include <stdlib.h>
5 #include <time.h>
6
7 /* Renvoie un booléen indiquant si cible == essai, et affiche un
8 message selon si essai est supérieur, égal ou inférieur à cible */
9 bool verifier(int cible, int essai){
10     if (cible == essai){
11         printf("Gagné !\n");
12         return true;
13     }
14     // si l'on arrive ici, forcément cible < ou > à essai
15     if (cible < essai){
16         printf("Essayez plus bas\n");
17     } else {
18         printf("Essayez plus haut\n");
19     }
20     return false;
21 }
22
23
24 int main(){
25     srand(time(NULL));
26     int cible = rand() % 5000;
27     int essai;
28     int nb_essai = 0; // nombre d'essais réalisés
29
30     // Premier scan
```

```

31 printf("Entrez un entier:\n");
32 scanf("%d", &essai);
33
34 while (!verifier(cible, essai)){
35     printf("Entrez un entier:\n");
36     scanf("%d", &essai);
37     nb_essai++; // équivalent à nb_essai = nb_essai + 1;
38
39 }
40 // en sortie la condition de boucle est fausse, on
41 // a donc bien trouvé la cible
42 printf("Vous avez trouvé en %d essais\n", nb_essai);
43
44 return 0;
45 }

```

Une stratégie efficace dont vous avez a priori parlé en mathématiques (mais dans un autre contexte) est la **dichotomie**. Elle consiste à toujours essayer la valeur au milieu de l'intervalle de recherche. Par exemple, au départ, on cherche entre 0 et 4999, on essaie donc 2500. Si l'ordinateur dit qu'il faut essayer plus bas, on essaie 1250, sinon 3750.

De cette manière, on divise systématiquement la taille de l'intervalle de recherche par 2, ce qui permet de toujours trouver en environ  $\lceil \log_2(5000) \rceil = 13$  coups.

## Exercice 9

Cet exercice est particulièrement difficile si vous n'avez jamais programmé, il sert à comprendre qu'écrire du code avec des fonctions récursives demande de réfléchir de façon totalement différente. Néanmoins, c'est une manière assez mathématique d'écrire du code !

**Q1.** Lorsque l'on écrit une fonction récursive, on doit toujours réfléchir à deux cas :

- Le (ou les) cas de base : à quelle condition peut-on répondre immédiatement à la question. Ici, le cas de base est  $n = 0$  : pour décompter les 0 premiers entiers, on ne fait rien !
- Le reste : si l'on n'est pas dans un cas de base, comment peut-on s'en rapprocher. Ici, on peut voir que pour décompter les  $n$  premiers entiers, il faudrait afficher  $n-1, n-2, n-3, \dots, 0$ . On voit alors apparaître la suite  $n-2, \dots, 0$ , qui n'est rien d'autre que le décompte des  $n-1$  premiers entiers !

Autrement dit, on peut dire que "Pour décompter les  $n$  premiers entiers, si  $n = 0$  on ne fait rien, sinon on affiche  $n-1$  puis on décompte les  $n-1$  premiers entiers". Cette phrase nous donne en fait le code de la fonction :

```

1  /* Affiche les n premiers entiers naturels, dans l'ordre
2     décroissant. (n >= 0) */
3  void decompte(int n){
4     assert(n >= 0);
5     if (n == 0){
6         // ne rien faire
7     } else {
8         printf("%d\n", n-1);
9         decompte(n-1);
10    }
11 }

```

- Q2.** Pour compter dans l'ordre croissant, on peut raisonner de manière analogue : compter de 0 à  $n - 1$ , c'est compter de 0 à  $n - 2$ , puis afficher  $n - 1$  :

```
1 // Q2
2 /* Affiche les n premiers entiers naturels, dans l'ordre
3    croissant. (n >= 0) */
4 void compte(int n){
5     assert(n >= 0);
6     if (n == 0){
7         // ne rien faire
8     } else {
9         compte(n-1);
10        printf("%d\n", n-1);
11    }
12 }
```

- Q3.** Cette fonction est assez compliquée, il faut la traiter méthodiquement. La première chose à voir est qu'il faudra récupérer d'une part le chiffre des unités et d'autre part le reste du nombre. Par exemple, on voudrait pouvoir décomposer 91426 en 9142 et 6. Une fois que c'est fait, on peut traiter récursivement 9142. En réalité, cette opération correspond à avoir fait une division euclidienne par 10 :  $91426 = 9142 \times 10 + 6$ . On en déduit la fonction :

```
1 /* Affiche les chiffres de n sur une ligne chacun,
2    de la plus grande puissance de 10 à la plus petite */
3 void chiffres(int n){
4     // cas particulier: si n < 0, on commence par afficher
5     // un - et on traite |n| ensuite
6     if (n < 0){
7         printf("-\n");
8         chiffres(-n);
9     } else if (n == 0) {
10        // ne rien faire
11    } else {
12        // enlever le chiffre des unités,
13        // et recommencer avec ce qui reste
14        int unite = n % 10;
15        int ce_qui_reste = n / 10; // exemple: 948 / 10 = 94
16        // afficher les unités en dernier
17        chiffres(ce_qui_reste);
18        printf("%d\n", unite);
19    }
20 }
```

- Q4.** Cette question est assez proche de ce qui a été fait dans les Q1 et Q2 :

```
1 /* Affiche k '0' sur une ligne. (k >= 0) */
2 void affiche_0(int k){
3     assert(k >= 0);
4     if (k == 0){
5         printf("\n");
6     } else {
7         printf("0");
8         affiche_0(k-1);
9     }
10 }
```



**Q5.** Il suffit pour cette question de combiner les idées des questions 4 et 5 :

```
1  /* Affiche les chiffres de n sur une ligne chacun,  
2     chaque chiffre k étant représenté par k '0',  
3     de la plus grande puissance de 10 à la plus petite */  
4  void chiffres_0(int n){  
5      // cas particulier: si n < 0, on commence par afficher  
6      // un - et on traite |n| ensuite  
7      if (n < 0){  
8          printf("-\n");  
9          chiffres_0(-n);  
10     } else if (n == 0) {  
11         // ne rien faire  
12     } else {  
13         // enlever le chiffre des unités,  
14         // et recommencer avec ce qui reste  
15         int unite = n % 10;  
16         int ce_qui_reste = n / 10; // exemple: 948 / 10 = 94  
17         // afficher les unités en dernier  
18         chiffres_0(ce_qui_reste);  
19         affiche_0(unite);  
20     }  
21 }
```

## Exercice 10

Code :

```
1  #include <stdio.h>  
2  #include <unistd.h>  
3  
4  // Affiche ping - pong - ping - pong - ping - .....  
5  void ping();  
6  
7  
8  // Affiche pong - ping - pong - ping - pong - .....  
9  void pong();  
10  
11  
12 void ping() {  
13     printf("ping\n");  
14     sleep(1);  
15     pong();  
16 }  
17  
18 void pong() {  
19     printf("pong\n");  
20     sleep(1);  
21     ping();  
22 }  
23  
24  
25 // Rappel: appuyer sur CTRL-C arrête l'exécution  
26 int main() {  
27     ping();  
28 }
```

**A retenir :** Lorsque l'on écrit une fonction récursive, on commence par écrire le commentaire de documentation, dans lequel on décrit le résultat de la fonction, mais pas la manière dont elle procède récursivement. Puis, on réfléchit aux cas de base et aux cas récurifs. Il faut essayer de réfléchir en ne faisant qu'un seul "pas" récurif, comme fait au début de l'exercice pour écrire la fonction de décompte.

## Exercice 11

Pour bien réussir cet exercice et arriver à répondre aux dernières questions, il faut écrire du code qui évite de refaire plusieurs fois le même calcul. Par exemple, pour calculer le temps de vol d'un entier  $x$ , on peut être tenté de calculer `syracuse(x, k)` avec  $k$  qui varie via une boucle while, mais ce n'est pas efficace. En effet, lorsque l'on a calculé, disons, `syracuse(x, 5000)`, pour obtenir le terme suivant `syracuse(x, 5001)`, le programme va tout recalculer depuis le départ alors qu'il est plus efficace d'utiliser la fonction suivante pour calculer les termes de proche en proche :

```
1 /* Renvoie le plus petit n tel que, en notant (u_k) la suite
2    de syracuse de x, u_n = 0 */
3 long int temps_de_vol(long int x){
4     assert(x >= 0);
5     if (x == 1){
6         return 0;
7     }
8     return 1 + temps_de_vol(suivant(x));
9 }
```

Pour le plus long temps de vol, il faut faire attention au fait que l'on cherche l'entier **dont** le temps de vol est maximal, on ne cherche pas directement le temps de vol maximal! Ainsi, il faut deux variables dans le programme : une qui stocke le meilleur temps de vol vu pour l'instant, et une autre qui stocke l'entier qui réalise ce meilleur temps :

```
1 /* Renvoie l'entier entre 1 et N ayant le plus long
2    temps de vol.*/
3 long int plus_long_vol(long int N){
4     assert(N >= 1);
5     int plus_long_temps = 0;
6     int candidat = 1; // x tel que temps_de_vol(x) = plus_long_temps
7     for (int i = 1; i <= N; ++i){
8         int temps_i = temps_de_vol(i);
9         if (temps_i > plus_long_temps){
10             // avoir stocké le temps de vol dans une variable temps_i
11             // évite d'avoir à le recalculer ici !
12             plus_long_temps = temps_i;
13             candidat = i;
14         }
15     }
16     return candidat;
17 }
```

Afin de trouver les valeurs demandées dans l'énoncé, on peut utiliser des fonctions auxiliaires qui font un affichage des réponses aux différentes questions, et les appeler dans le main :

```
1 // Fonctions pour répondre aux questions
2 void repondre_Q2(long int x, int n){
3     printf("x = %ld, n = %d: syracuse(x, n) = %ld\n", x, n, syracuse(x, n));
4 }
5
6 void repondre_Q3(long int x){
7     printf("x = %ld: temps_de_vol(x) = %ld\n", x, temps_de_vol(x));
8 }
9
10 void repondre_Q4(long int N){
11     printf("N = %ld: plus_long_vol(N) = %ld\n", N, plus_long_vol(N));
12 }
13
14
15 int main(){
16     // tests (on vérifie les valeurs du tableau donné dans l'énoncé)
17     assert(suivant(5) == 16);
18     assert(suivant(16) == 8);
19     assert(suivant(3) == 10);
20     assert(suivant(2) == 1);
21     assert(suivant(1) == 4);
22
23     assert(syracuse(1, 0) == 1);
24     assert(syracuse(1, 7) == 4);
25     assert(syracuse(5, 4) == 2);
26     assert(syracuse(6, 4) == 16);
27
28     assert(temps_de_vol(1) == 0);
29     assert(temps_de_vol(5) == 5);
30     assert(temps_de_vol(6) == 8);
31     assert(temps_de_vol(2) == 1);
32
33     assert(plus_long_vol(4) == 3);
34
35     printf("TESTS OK\n");
36
37     // réponses aux questions
38     repondre_Q2(9, 6);
39     repondre_Q2(77, 128);
40     repondre_Q2(1023, 729);
41     repondre_Q2(1234567, 52697);
42
43     repondre_Q3(1);
44     repondre_Q3(26);
45     repondre_Q3(27);
46     repondre_Q3(28);
47     repondre_Q3(77030);
48     repondre_Q3(77031);
49
50     // boucle qui teste N = 10, 100, 1000, ... jusqu'à 10000000
51     for (long int N = 10; N <= 10000000; N = N * 10) {
52         repondre_Q4(N);
53     }
54 }
```

On trouve les valeurs suivantes :

```
x = 9, n = 6: syracuse(x, n) = 34
x = 77, n = 128: syracuse(x, n) = 4
x = 1023, n = 729: syracuse(x, n) = 4
x = 1234567, n = 52697: syracuse(x, n) = 2
```

```
x = 1: temps_de_vol(x) = 0
x = 26: temps_de_vol(x) = 10
x = 27: temps_de_vol(x) = 111
x = 28: temps_de_vol(x) = 18
x = 77030: temps_de_vol(x) = 107
x = 77031: temps_de_vol(x) = 350
```

```
N = 10: plus_long_vol(N) = 9
N = 100: plus_long_vol(N) = 97
N = 1000: plus_long_vol(N) = 871
N = 10000: plus_long_vol(N) = 6171
N = 100000: plus_long_vol(N) = 77031
N = 1000000: plus_long_vol(N) = 837799
N = 10000000: plus_long_vol(N) = 8400511
```

Notons que le programme prend une trentaine de secondes pour trouver la dernière valeur de la dernière questions (peut être même plus sur les machines virtuelles!). De nombreuses méthodes permettraient d'accélérer la recherche. Une optimisation simple est de se rendre compte que les entiers impairs entre 1 et  $\frac{N}{2}$  ne servent à rien dans la recherche : pour un tel entier  $k$ , son double  $2k$  aura un temps de vol strictement supérieur. On peut donc limiter la recherche aux entiers qui sont soit pairs soit entre  $\frac{N}{2}$  et  $N$ , ce qui abaisse légèrement le temps de recherche.

Nous verrons aux fils des cours d'autres stratégies assez générales qui permettraient d'améliorer ce code, comme la **mémoïsation** dont on parlera au chapitre 10 et qui divise par 10 le temps d'exécution de ce programme!.

**Types entiers** Dans le sujet, il est indiqué qu'il faut utiliser le type `long int` car le type `int` n'est pas assez grand pour stocker les entiers manipulés. Si l'on essaie de remplacer tous les `long int` par des `int` dans le code fourni, on tombe sur un problème étrange : au cours de l'exécution, on déclenche une assertion `assert(x >= 0)`, ce qui signifie que l'on se retrouve à manipuler des entiers négatifs! Ceci est dû à la manière dont les entiers relatifs sont encodés en C (voir chapitre 2) : lorsqu'un entier positif devient trop grand, il cause un **dépassement** et devient négatif. Pour les `int`, la limite est  $2^{31} - 1$  : si  $x$  vaut  $2^{31} - 1$ , alors  $x + 1$  vaut  $-2^{31}$ .