Travaux dirigés 11 : fonctions, fonctions récursives

1 Fonctions récursives

```
1  /* Declaration de fonctionnalites supplementaires */
2  #include <stdlib.h> /* EXIT_SUCCESS */
3  #include <stdio.h> /* printf() */
4
5  /* Declarations constantes et types utilisateurs */
6
7  /* Declarations de fonctions utilisateurs */
8  int factorielle(int n);
9
10 /* Fonction principale */
11 int main()
12 {
13     /* Declaration et initialisation des variables */
14     int x = 3; /* argument */
15     int res;   /* resultat */
16
17     /* calcul */
18     res = factorielle(x);
19
20     /* affichage */
21     printf("%d! = %d\n", x, res);
22
23     /* Valeur fonction */
24     return EXIT_SUCCESS;
25 }
26
27 /* Definitions de fonctions utilisateurs */
28 int factorielle(int n)
29 {
30     int res; /* resultat */
31     if (n > 1) /* cas récursif */
32     {
33         res = n * factorielle(n - 1);
34     }
35     else /* cas de base */
36     {
37         res = 1;
38     }
39     return res;
40 }
```

1. Faire la trace du programme précédent.

   Correction.

   Trace. Voir le tableau 1 page 3.

2. La suite de Fibonacci est définie récursivement par la relation $u_n = u_{n-1} + u_{n-2}$. Cette définition doit être complété par une condition d'arrêt, par exemple : $u_1 = u_2 = 1$. Écrire une fonction qui calcule et renvoie le n-ième terme de la suite de Fibonacci ($n\mathbb{N}^*$ donné en argument de la fonction).

   Correction.

   ```
   int Fibonacci(n)
   {
       if (n < 3) /* cas de base */
       {
           return 1;
       }
       return Fibonacci(n - 1) + Fibonacci(n - 2); /* /!\ double appel récursif */
   }
   ```

   Vous pouvez en profiter pour dire rapidement que ce genre de doubles appels récursifs prennent du temps en montrant par exemple que pour calculer `Fibonacci(n)` il faut revenir `Fibonacci(n)` fois au cas de base (puisque le résultat est calculé comme une somme $1 + 1 + \ldots + 1$). Et la suite croît très vite, pour $n = 31$ on est déjà au delà du million.

3. Il n'est parfois pas suffisant d'avoir un bon cas de base, voici un exemple. En C, que vaut `Morris(1, 0)` ?

   ```
   int Morris(int a, int b)
   {
       if (a == 0)
       {
           return 1;
       }
       else
       {
           return Morris(a - 1, Morris(a, b));
       }
   }
   ```

   Correction.

   ```
   Segmentation fault
   ```

   En effet `Morris(1, 0)` lance le calcul de l'expression `Morris(0, Morris(1, 0))` qui induit le calcul préalable de la valeur de `Morris(1, 0)` et ainsi indéfiniement, jusqu'à ce que le programme ne puisse plus lancer de nouveaux sous calculs faute de mémoire disponible à cet effet. (Sur mon système ça s'arrête au bout de 255×1024 appels, ça doit être quelque chose comme 8 Mo alloués à la pile, avec 8 mots de 32 bits alloués à chaque appel mais il ne faut pas s'étendre sur la structure de la mémoire d'un processus).

4. Les fonctions récursives mêmes simples donnent parfois des résultats difficiles à prévoir. Pour s'en convaincre voici un exemple. Pour $n > 100$ la fonction 91 de McCarthy vaut $n - 10$. Mais pour $n < 100$? (Tester sur un exemple... pas trop mal choisi).

`main()`

ligne	x	res	Affichage
initialisation	3	?	
18			
18		6	
18			`3! = 6`
24	renvoie `EXIT_SUCCESS`		

`factorielle(3)`

ligne	n	res	Affichage
initialisation	3	?	
33			
33		6	
39	renvoie 6		

`factorielle(2)`

ligne	n	res	Affichage
initialisation	2	?	
33			
33		2	
39	renvoie 2		

`factorielle(1)`

ligne	n	res	Affichage
initialisation	1	?	
37		1	
39	renvoie 1		

Table 1 – Trace du programme de l'exercice 1.

```
int McCarthy(int x)
{
    if (x > 100)
    {
        return (x-10);
    }
    else
    {
        return McCarthy(McCarthy(x + 11));
    }
}
```

Correction. Prenons 91 comme exemple et notons f la fonction. Comme $91 < 100$ le résultat de $f(91)$ sera $f(f(91+11)) = f(f(102))$. Mais $f(102) = 92$ donc $f(91) = f(f(102)) = f(92)$. Maintenant $f(92) = f(f(92+11)) = f(f(103)) = f(93)$ et par le même raisonnement $f(93) = f(94) = \ldots = f(99) = f(f(110)) = f(100) = f(f(111)) = f(101) = 91$. Donc $f(91) = \ldots = f(100) = 91$. Finalement, pour 90 on a $f(90) = f(f(101)) = f(91) = 91$. Voyons ce que ça donne pour 89. On a $f(89) = f(f(100)) = f(91) = 91$. On peut éventuellement continuer en donnant une preuve de $f(n) = 91 (\forall n \leq 100)$. Il faut surtout profiter de leur éventuelle curiosité pour leur demander de tester expérimentalement le résultat pour de nombreuses valeurs de n en TP (la question est reprise plus loin).
Une preuve de $f(n) = 91$ pour $n \leq 100$.

Démonstration. On a $f(101) = 91$. On montre que $f(n) = f(n+1)$ pour $90 \leq n \leq 100$. Si $90 \leq n \leq 100$ alors

$$\begin{aligned} f(n) &= f(f(n+11)) \text{ et } n+11 \geq 101 \\ &= f(n+11-10) \\ &= f(n+1) \\ &\ldots \\ &= f(101) \end{aligned}$$

On montre alors par récurrence que lorsque $90 - k \times 11 \leq n \leq 100 - k \times 11$, $f(n) = 91$. Le cas de base $k = 0$ est démontré. Supposons l'assertion vraie pour k on montre qu'elle l'est encore pour $k+1$. Soit n un entier dans l'intervalle $[90 - (k+1) \times 11, 100 - (k+1) \times 11]$. Alors $n \leq 100$ donc $f(n) = f(f(n+11))$. Mais $f(n+11)$ est dans l'intervalle $[90 - k \times 11, 100 - k \times 11]$ donc par hypothèse de récurrence $f(n+11) = 91$ et ainsi $f(n) = f(91)$ qui est égal à 91 (cas de base). Finalement le résultat est prouvé pour tous les intervalles $[90 - k \times 11, 100 - k \times 11]$ et leur union $([90, 100] \cup [79, 89] \cup [68, 78] \cup \ldots)$ correspond à l'ensemble des entiers inférieurs à 100. □

2 PGCD

Rappel : pour a et d deux entiers naturels, on dit que d divise a, et on note $d \mid a$, s'il existe un entier q tel que $a = dq$. En particulier $d \mid 0$ quel que soit d. Le plus grand diviseur commun (PGCD) de deux entiers naturels a et b est le plus grand entier d tel que $d \mid a$ et $d \mid b$. Cette définition donne un moyen effectif de trouver le PGCD de deux entiers (un algorithme) : il suffit de chercher tous les diviseurs de a et de b et de prendre le plus grand d'entre eux.

1. Que vaut $\text{PGCD}(0, 0)$?

Correction. PGCD$(0, 0)$ est mal défini car il n'existe pas de plus grand entier naturel.

2. Si seulement un des entiers a et b est nul, que vaut PGCD(a, b) ?

Correction. L'autre entier. C'est à dire que PGCD$(0, x) =$ PGCD$(x, 0) = x$.

3. Si aucun des deux entiers a et b n'est nul, jusqu'a quelle valeur doit-on chercher des diviseurs de a et de b ?

Correction. Lorsque x est non nul un diviseur de x est nécessairement plus petit que x. On peut donc certainement s'arrêter au minimum de a et b.

4. Comment tester si un nombre en divise un autre, en C ?

Correction. Il faut les inciter à formuler leur réponse complétement, puis à essayer de l'écrire en C, même si ça n'est pas optimal.
Une réponse possible. Si $a, d > 0$, pour tester si $d \mid a$ on peut multiplier tour à tour d par tous les entiers en commençant à 1 tant que le résultat est strictement inférieur a. Lorsqu'on s'arrête soit on a trouvé a comme résultat de la multiplication et dans ce cas d divise a soit on a trouvé un nombre strictement supérieur à a et d ne divise pas a. (On peut aussi faire des additions successives de d dans un accumulateur).

```
int divise(int d, int a)
{
    int q = 1;
    if (a == 0) /* cas dégénéré */
    {
        return TRUE;
    }
    if (d == 0) /* autre cas dégénéré */
    {
        return FALSE;
    }
    /* cas général */
    while (q * d < a)
    {
         q = q + 1;
    }
    /* q est le premier entier tel que q * d >= a */
    if (q * d == a)
    {
        return TRUE;
    }
    return FALSE; /* on peut aussi leur montrer return q * d == a. */
}
```

Autre solution plus économe : tester si $a \mod d$ vaut zéro (`a % d == 0` ou bien `a == d *(a/d)`).

5. Écrire une fonction non récursive qui calcule le PGCD de deux entiers naturels.

Correction.

```
int pgcd(int a, int b)
{
```

```
    int min;     /* minimum entre a et b */
    int d = 1;   /* var. de boucle */
    int res = 0; /* resultat */

    if (a == 0) /* premier cas dégénéré */
    {
        return b;
    }
    if (b == 0) /* second cas dégénéré */
    {
        return a;
    }
    /* a > 0 et b > 0*/

    /* calcul de minimum(a, b) */
    if (a > b)
    {
        min = b;
    }
    else
    {
        min = a;
    }
    /* min est le minimum entre a et b */

    for (d = 1; d < min; d = d + 1)
    {
         if ( (a % d == 0) && (b % d == 0) ) /* d | a et d | b */
         {
              res = d;
         }
    }
    /* res est le plus grand d tq d | a et d | b */

    return res;
}
```

6. Rappeler l'algorithme d'Euclide pour le calcul du PGCD.

Correction.

> « Étant donnés deux entiers naturels a et b, on commence par tester si b est nul. Si oui, alors le PGCD est égal à a. Sinon, on calcule c, le reste de la division de a par b. On remplace a par b, et b par c, et on recommence le procédé. Le dernier reste non nul est le PGCD. » (wikipedia.fr, dans le passé).

Autrement dit, lorsque $a \geq b$ on utilise la division euclidienne, $a = bq + r$ et on recommence avec le couple (b, r) jusqu'à tomber sur un reste nul. C'est justifié car si $d \mid a$ et $d \mid b$ alors $d \mid r$ (et $d \mid b$) et réciproquement si $d \mid b$ et $d \mid r$ alors $d \mid a = bq + r$ (et $d \mid b$).
En résumé le PGCD de deux entiers positifs a et b est défini par : $\mathrm{PGCD}(a, b) = a$ si $b = 0$ et $\mathrm{PGCD}(a, b) = \mathrm{PGCD}(b, a \mod b)$.

7. Écrire une fonction récursive pour le calcul du PGCD de deux entiers.

   Correction.

```
int pgcd(int a, int b)
{
    if (b == 0)
    {
        return a;
    }
    return pgcd(b, a % b);
}
```

   Remarque : si l'entier le plus grand est b, la fonction échange ses arguments au premier appel récursif.

8. À votre avis quelle est la méthode la plus rapide ?

   Correction. La fonction récursive sera toujours au moins aussi rapide que la fonction itérative. On peut par exemple remarquer que si les deux entiers sont égaux dans la fonction récursive le deuxième appel se fait avec zéro comme second argument et renvoie a immédiatemment tandis que l'algortihme précédent va dérouler la boucle jusqu'à a.

3 Conjecture de Syracuse

Soit la fonction mathématique $f : \mathbb{N} \to \mathbb{N}$ définie par :

$$f : x \mapsto \begin{cases} x/2 \text{ si } x \text{ pair} \\ 3x + 1 \text{ sinon.} \end{cases}$$

On appelle cette fonction la fonction de Collatz et comme vous pourrez le vérifier sur internet elle a donné et donne toujours du fil à retordre aux mathématiciens. Voici pourquoi.

Soit un entier positif n. Si on calcule $f(n)$, puis $f(f(n))$, puis $f(f(f(n)))$ etc. on finit toujours par tomber sur la valeur 1 quelle que soit la valeur de $n \in \mathbb{N}^*$. Aucun mathématicien n'est arrivé à ce jour à le démontrer (et on ne sait même pas démontrer que c'est indécidable, c'est à dire non démontrable à l'aide des mathématiques usuelles). Cela reste une conjecture.

On peut également formuler le problème avec une suite définie par récurrence u_0 vaut n, et $u_{n+1} = f(u_n)$. Les termes succesifs de la suite sont alors : x, $f(x)$, $f(f(x))$ etc. La conjecture est que la suite (u_n) finit par atteindre la valeur 1, pour $n \in \mathbb{N}^*$. Si on poursuit le calcul la suite devient périodique, de période 4, 2, 1, on dit que la suite est entrée dans un puit. Par exemple en partant de 15 on a la suite :

$$15, 46, 23, 70, 35, 106, 53, 160, 80, 40, 20, 10, 5, 16, 8, 4, 2, 1, (4, 2, 1, 4, 2, 1, \ldots)$$

Les mathématicien parlent de la trajectoire de n pour désigner cette suite et il définissent également le temps de vol (le nombre de termes avant le premier 1, ici 17) et d'altitude maximale pour la valeur maximale prise par la suite (ici 160).

1. Écrire en langage C une fonction `Collatz` calculant la valeur de la fonction f sur son argument.

   Correction.

```
int Collatz(int x)
{
    if (x % 2 == 0) /* x pair */
```

```
    {
        return x / 2;
    }
    else /* x impair */
    {
        return 3 * x + 1;
    }
}
```

2. Écrire une procédure (un fonction ne renvoyant pas de résultat) *récursive* `Syracuse` qui calcule les itérations successives de la fonction de Collatz sur un entier donné en argument, jusqu'à trouver 1. Pour le moment la fonction `Syracuse` n'affichera rien et ne renverra pas de résultat (mais elle doit calculer chacun des termes successifs de Collatz).

   Correction.

```
int Syracuse(int x)
{
    if (x > 1)
    {
        Syracuse(Collatz(x));
    }
}
```

3. Ajouter ensuite un affichage des valeurs successives trouvées dans cette fonction. Dans l'ordre direct. Puis dans l'ordre inverse.

   Correction. Il est nécessaire que ce soit corrigé en TD ou en TP. Si les étudiants sont déjà en TP à ce moment, il faut qu'ils passent à la partie TP.

```
int Syracuse(int x)
{
    printf("%d ", x); /* <-- ordre direct (a enlever pour l'ordre inverse) */
    if (x > 1)
    {
        Syracuse(Collatz(x));
    }
    /* printf("%d ", x); /* <-- ordre inverse */
}
```

4. En reprenant le code de Syracuse écrire une fonction récursive `temps_de_vol` qui renvoie le nombre d'itérations de la fonction de Collatz nécessaire pour atteindre 1.

   Correction.

```
int temps_de_vol(int x)
{
    if (x == 1)
    {
        return 0;
    }
    else
    {
        return 1 + temps_de_vol(Collatz(x));
```

```
    }
}
```

5. Question facultative. Comment faire en sorte que le temps de vol soit affiché plutôt que renvoyé comme valeur de retour ?

   Correction.

```
void temps_de_vol(int x)
{
    temps_de_vol_aux(x, 0);
}

void temps_de_vol_aux(int x, int temps)
{
    if (x == 1)
    {
        printf("Temps de vol : %d\n", temps);
    }
    else
    {
        temps_de_vol_aux(Collatz(x), temps + 1);
    }
}
```

6. Question facultative. En reprenant le code précédant écrire une nouvelle fonction `altitude` qui renvoie la valeur maximum prise par les termes successifs de l'itération (x compris).

   Correction.

```
int max(int a, int b)
{
    if (a > b)
    {
        return a;
    }
    return b;
}

int altitude(int x)
{
    if (x == 1)
    {
         return 1;
    }
    else
    {
        return max(x, altitude(Collatz(x)));
    }
}
```

   Et s'il y en a qui avancent vite ils peuvent continuer en mélangeant les deux dernières questions de différentes façons (faire la dernière comme l'avant dernière, intégrer l'avant dernière dans la dernière etc.).