ENSEIRB 2006-2007 - Première année
PG101 - Programmation en C

TD 4 : Fonctions et récursivité (commentaires)

1 Premières fonctions

Exercice 1:

On peut citer :
- toutes les fonctions `main` que nous avons appelées «programme principal» jusqu'ici
- les fonctions d'entrée-sortie `printf` et `scanf` appartenant à la bibliothèque `stdio.h`
- la fonction `toupper` de la bibliothèque `ctype.h`
- la fonction `sqrt` de la bibliothèque `math.h`

Attention, `sizeof` n'est pas une fonction mais un opérateur du langage C.

On ignore pour le moment les prototypes de `printf` et `scanf` (voir TD suivant). À l'aide de la commande `man` (`man toupper` et `man sqrt`) nous obtenons les prototypes suivants :

```
int toupper(int c);
double sqrt(double x);
```

- la fonction `toupper` prend en paramètre une valeur de type `int` et renvoie une valeur de type `int`.
  Ceci peut sembler étrange car on pourrait s'attendre à ce qu'elle travaille avec des valeurs de type `unsigned char`. Les fonctions de manipulation de caractères ont cependant besoin de traiter non seulement l'enssemble des caractères du code ASCII (qui prend toutes les valeurs représentable par le type `unsigned char`), mais aussi une valeur spéciale : `EOF`, «End Of File», servant à gérer les fichiers. C'est pourquoi le domaines de ces fonctions est étendu au type `int`. L'utilité de `EOF` sera abordée au S2.
- la fonction `sqrt` prend une valeur de type `double` et renvoie une valeur de même type. Les fonctions de la bibliothèque `math.h` sont programmées pour travailler avec des nombres flottants de précision accrue (`double` plutôt que `float`)

Exercice 2:

Le calcul du carré de `x` consiste simplement à multiplier la valeur de `x` par elle-même. Afin de retourner cette valeur au programme, nous utilisons le mot-clé `return`. Ceci nous donne la fonction :

```
double sqr(double x) {
  return (x*x);
}
```

Nous devons maintenant écrire la fonction `main`. La lecture de la valeur de `x` se fait à l'aide de `scanf` et l'affichage du résultat à l'aide de `printf`. Ceci nous donne le programme :

```
#include <stdio.h>

double sqr(double x) {
  return (x*x);
}

int main() {
  float y;

  printf("Entrez une valeur: ");
  scanf("%f", &y);
  printf("%f au carre vaut %f\n", y, sqr(y));

  return 0;
}
```

Exercice 3:

Pour calculer x^n, nous devons calculer les puissances successives de `x` jusqu'à `n`. Nous sommes donc en présence d'une file : celle des puissances successives de `n`. La longueur de cette file est connue : c'est `n`.

Nous complétons maintenant le schéma de traitement de file, en modifiant légèrement la boucle `for` pour que `i` décrive les valeurs de 1 à `n`, ce qui correspond aux éléments de la file :

```
for (i=1; i<=n; i++) {
...
}
```

- `N` vaut ici `n`
- `<obtenir l'élément i>` correspond à la valeur de `i`
- `<traiter l'élément i>` nous utilisons une variable `p` de type `double` pour stocker la valeur de x^{i-1}. Il suffit donc de multiplier `p` par `x` pour obtenir x^i.
- `<initialisations>` la variable `p` doit être initialisée à 1 qui correspond à x^0
- `<présentation du résultat>` consiste à renvoyer la valeur calculée

Nous obtenons alors le programme :

```
#include <stdio.h>

double power(double x, unsigned int n) {
  unsigned int i;
  double p;

  p = 1;
  for (i=1; i<=n; i++)
    p = p*x;
```

```
  return p;
}

int main() {
  float y;
  unsigned int n;

  printf("Valeur: ");
  scanf("%f", &y);
  printf("Puissance: ");
  scanf("%u", &n);

  printf("%f puissance %u = %f\n", y, n, power(y,n));

  return 0;
}
```

Exercice 4:

Les traitements de file correspondants ont été étudiés lors du TD précédent. Nous vous conseillons de procéder à leur étude à nouveau à titre d'entraînement. Nous ne donnons ici que les programmes résultants, sans refaire ces analyses.

- La fonction `str_len` :

```
int str_len(char s[]) {
  int i = 0;

  while (s[i] != '\0')
    i++;

  return (i);
}
```

- La fonction `str_cmp` :

```
int str_cmp(char s1[], char s2[]) {
  int i = 0;

  while ((s1[i] == s2[i]) && (s1[i] != '\0'))
    i++;

  return (s1[i] - s2[i]);
}
```

Remarque : nous voyons ici un nouvel exemple du fait que les variables stockent une valeur que l'on peut interpréter de différentes façons. Ici, `s1[i]` comme `s2[i]` ont le type `char`, que nous avons eu l'habitude d'interpréter comme un caractère lors de l'affichage avec `printf` et le format `%c` par exemple. Mais cette valeur peut aussi être interprétée comme un entier, c'est à dire comme le code ASCII correspondant au caractère, sur laquelle nous pouvons utiliser l'arithmétique (ici, dans un but de comparaison).

- La fonction `str_cat` :

```
void str_cat(char s1[], char s2[]) {
  int i, j;

  /* Deplacement a la fin de s1 */
  i=0;
  while (s1[i] != '\0')
    i++;

  /* Concatenation de s2 */
  j=0;
  while (s2[j] != '\0') {
    s1[i+j] = s2[j];
    j++;
  }

  s1[i+j] = '\0';
}
```

- La fonction `str_ncat` :

```
void str_ncat(char s1[], char s2[], unsigned int n) {
  int i, j;

  /* Deplacement a la fin de s1 */
  i=0;
  while (s1[i] != '\0')
    i++;

  /* Concatenation de s2 */
  j=0;
  while ((j<n) && (s2[j] != '\0')) {
    s1[i+j] = s2[j];
    j++;
  }

  s1[i+j] = '\0';
}
```

Remarque : bien utilisée, la fonction `str_ncat` présente l'avantage sur `str_cat` de permettre le contrôle de non débordement de la capacité de `s1` lors de la concaténation

- La fonction `str_cpy` :

```
void str_cpy(char s1[], char s2[]) {
  int i;

  i = 0;
  while (s2[i] != '\0') {
    s1[i] = s2[i];
    i++;
  }
```

```
  s1[i] = '\0';
}
```

- La fonction `str_ncpy` :

```
void str_ncpy(char s1[], char s2[], unsigned int n) {
  int i;

  i = 0;
  while ((i<n) && (s2[i] != '\0')) {
    s1[i] = s2[i];
    i++;
  }
  s1[i] = '\0';
}
```

Remarque : bien utilisée, la fonction `str_ncpy` présente l'avantage sur `str_cpy` de permettre le contrôle de non débordement de la capacité de `s1` lors de la copie

2 Portée et visibilité

Exercice 5:

1. Il y a une variable locale : `i`, qui est locale à la fonction `main`, définie en ligne 13
2. Il y a une variable globale : `n` qui est déclarée en ligne 3
3. La fonction `incrementer` est déclarée en ligne 5 (il s'agit de son prototype), et définie de la ligne 7 à la ligne 10.

   La fonction `main` est déclarée et définie de la ligne 12 à la ligne 17. Ici, la déclaration est implicitement faite lors de la définition de la fonction
4. On constate que le programme fonctionne de la façon attendue :

```
appel numero 1
appel numero 2
appel numero 3
appel numero 4
appel numero 5
```

5. Voici un exemple d'affichage produit par le nouveau programme :

```
appel numero -1881057447
appel numero -1881057446
appel numero -1881057445
appel numero -1881057444
appel numero -1881057443
```

   La variable `n` locale à `incrementer` masque la la variable globale `n` à l'intérieur de la fonction `incrementer`. Ces deux variables correspondent chacune à deux zones mémoires distinctes. À l'intérieur de la fonction `incrementer`, c'est la zone mémoire locale à cette fonction qui est associée au nom de variable `n`.

La variable locale `n` est donc visible à l'intérieur de la fonction `incrementer` tandis que la variable globale `n` est visible partout ailleurs.

La portée de la variable locale `n` est limitée au bloc dans lequel elle est déclarée : les instructions de la fonction `incrementer`. Par contre, la portée de la variable globale `n` correspond à l'ensemble du programme : c'est la zone de texte où elle est potentiellement visible.

6. Après modification, nous obtenons les messages suivants de la part du compilateur :

```
gcc -Wall prog4.c -o prog4
prog4.c: In function 'incrementer':
prog4.c:9: error: 't' undeclared (first use in this function)
prog4.c:9: error: (Each undeclared identifier is reported only once
prog4.c:9: error: for each function it appears in.)
prog4.c: In function 'main':
prog4.c:15: warning: unused variable 't'
```

Le premier message signifie que la variable `t` utilisée dans l'intruction `t++` dans la fonction `incrementer` n'est pas déclarée. Le second message signifie que la variable `t` déclarée dans la fonction `main` n'est pas utilisée.

En effet, la portée de la variable `t` est limitée au bloc d'instruction de la fonction `main` où elle est déclarée. Elle ne peut donc pas être visible à l'intérieur de la fonction `incrémenter`.

3 Récursivité

La récursivité est un mode de traitement de file où le calcul itérative est remplacé par des appels successifs d'une fonction. Cette construction de programme est souvent plus simple pour le calcul des suites mathématiques (dont la définition est récursive) et pour certaines structures de données (les listes et les arbres notamment qui ont une définition inductive).

Prenons l'exemple d'une suite définie par :

$$\begin{cases} u_0 = 0 \\ u_n = f(u_{n-1}) \quad n > 0 \end{cases}$$

pour une fonction f quelconque. Nous voulons programmer une fonction récursive `u` de prototype `int u(unsigned int n)` qui calcule la valeur u_n. Nous supposerons l'existence de la fonction `f` de prototype : `int f(int)`. Nous avons deux cas possibles suivant la valeur de `n` :

- soit `n` $= 0$ alors, la valeur de `u(0)` est `0`
- soit `n` > 0 (notons que `n` est de type `unsigned int`) alors la valeur de `u(n)` est égale à `f(u(n-1))`.

Ceci nous donne la fonction :

```
int u(unsigned int n) {
  if (n==0)
    return 0;
  else
    return f(u(n-1));
}
```

Prenons l'exemple du calcul de `u(3)`, nous pouvons le décomposer en :

```
u(3)
  |- u(2)
  |   |- u(1)
  |   |   |- u(0)
  |   |   |   |
  |   |   |  return 0
  |   |  return f(0)
  |  return f(f(0))
return f(f(f(0)))
```

Remarque : lorsque l'appel récursif `u(n-1)` se termine, il reste le calcul `f(u(n-1))`. Nous voyons ici les appels successifs à `f` une fois les appels récursifs terminés.

Le schéma général d'une fonction récursive est le suivant :

```
<type de retour> f(<paramètres>) {
  if (<cas d'arrêt>) {
    <traitement cas d'arrêt>
    return <résultat cas d'arrêt>
  }
  else {
    <traitement récursif>
    return <résultat récursif>
  }
}
```

où les blocs à compléter sont :

- `<type de retour>` le type du résultat produit par `f`. Notons qu'il peut s'agir de `void` lorsque `f` modifie l'un de ses paramètres (voir notamment les 2 derniers exercices)
- `<cas d'arrêt>` le ou les cas de terminaison de la récursivité
- `<traitement cas d'arrêt>` comment traiter le ou les cas d'arrêt de la récursivité. En particulier, le cas d'arrêt doit-il être traité comme les autres éléments de la file ?
- `<résultat cas d'arrêt>` éventuellement, le traitement du cas d'arrêt ne produit pas de résultat lorsque le type de retour de `f` est `void`
- `<traitement récursif>` traitement récursif de la file et obtention du résultat pour la suite de la file et traitement de l'élément courant. Attention, il n'y a pas d'ordre entre ces deux phases : parfois, il faut d'abord procéder à l'appel récursif puis traiter l'élément courant. C'est le cas sur l'exemple ci-dessus, où on commence par appeler `u(n-1)` puis, une fois le résultat obtenu, on traitement l'élément courant en appliquant `f` à cette valeur : `f(u(n-1))`. Il se peut aussi qu'on fasse l'inverse : on commence par traiter l'élément courant, puis on procède à l'appel récursif (voir notamment les 2 derniers exercices)
- `<résultat récursif>` éventuellement, le traitement récursif ne produit pas de résultat lorsque le type de retour de `f` est `void`
- `<paramètres>` les valeurs donc `f` a besoin pour le calcul récursif. L'un de ces paramètres (au moins) décrit une file de valeurs à traiter (la valeur de `n` à `0` dans l'exemple ci-dessus)

Ce schéma sera généralement bousculé par les besoins spécifiques de chaque fonction (il est moins générique que les schémas de traitement de file). Cependant, les questions qu'il pose sont celles auxquelles il faut répondre quel que soit le traitement récursif à mettre en œuvre.

Exercice 6:

La définition de la factorielle est :

$$\begin{cases} 0! = 1 \\ n! = n \times (n-1)! \quad n > 0 \end{cases}$$

On complète le schéma de traitement récursif :

- `<type de retour>` la valeur de la factorielle est un naturel, nous proposons donc de choisir `unsigned int`
- `<cas d'arrêt>` la définition mathématique de la factorielle nous donne comme cas d'arrêt `n` $= 0$
- `<traitement cas d'arrêt>` le résultat ici est immédiat, c'est la valeur 1, il n'y a donc pas de traitement à faire
- `<résultat cas d'arrêt>` c'est la valeur 1, on se contente donc de l'instruction : `return 1`
- `<traitement récursif>` la définition de la factorielle nous donne la méthode de calcul. Tout d'abord nous obtenons $(n-1)!$ par appel récursif à `fact`. Puis, nous multiplions le résultat par `n` pour traiter l'élément courant
- `<résultat récursif>` il s'agit du résultat de la multiplication expliquée à la ligne précédente
- `<paramètres>` d'après la définition de la factorielle, nous avons un paramètre `n` qui donne l'élément courant à calculer. Ce paramètre est un entier naturel, nous prenons le type `unsigned int`. Les différents traitements ne nécessite pas d'autre valeur.

Nous obtenons le programme suivant :

```
#include <stdio.h>

unsigned int fact(unsigned int n) {
  if (n==0)
    return 1;
  else
    return (n*fact(n-1));
}

int main() {
  unsigned int n;

  printf("n: ");
  scanf("%u", &n);
  printf("factorielle de %u=%u\n", n, fact(n));

  return 0;
}
```

Voici pour l'exemple l'exécution de la fonction `fact` pour `n` valant 3 :

```
fact(3)
  |- fact(2)
  |    |- fact(1)
  |    |    |- fact(0)
  |    |    |    |
  |    |    |  return 1
  |    |  return 1*1
  |  return 2*1
return 3*2
```

Exercice 7:

On procède à l'analyse de traitement récursif :

- `<type de retour>` d'après la spécification, `x` est de type réel (représenté par `double` en C), donc le résultat de x^n sera aussi une valeur réelle. Nous prenons donc `double` comme type de retour
- `<cas d'arrêt>` la spécification précise que pour $n = 0$, la définition de x^n n'est pas récursive. Par ailleurs, les autres valeurs de `n` conduise à des évaluations récursives pour des valeurs strictement décroissantes de `n`. Par ailleurs, on remarque que si `n` est paire, alors la suite de ses valeurs par appel récursif sera constituée de nombres pairs, jusqu'à obtenir la valeur 1. Si `n` est impaire, alors l'appel récursif aura pour paramètre une valeur paire (car $(n-1)/2$ est alors paire), nous ramenant au cas précédent. Enfin, si $n = 1$, alors l'appel récursif suivant consistera au calcul de x^0.
  Nous en déduisons donc que quelle que soit la valeur de `n` intialement considérée, après un nombre fini d'appels récursifs, nous aboutissons au calcul de x^0 qu'il est donc suffisant de considérer comme cas d'arrêt pour garantir a terminaison du calcul de puissance
- `<traitement cas d'arrêt>` il n'y a pas de traitement lié au cas d'arrêt : la valeur de x^0 est connue indépendamment de `x`
- `<résultat cas d'arrêt>` c'est la valeur 1, on se contente donc de l'instruction : `return 1`
- `<traitement récursif>` ici, nous devons faire deux cas suivant que `n` est paire ou impaire. Dans le premier cas, il s'agit d'appeler récursivement `power` pour $n/2$, puis de calculer le carré de la valeur obtenue. Dans le second cas, il s'agit d'appeler récursivement `power` pour $(n-1)/2$, puis de calculer le carré de la valeur obtenue et finalement de la multiplier par `x`. Le test de parité se fait en considérant le reste de la division de `n` par 2 :

```
if (n%2 == 0) {
  p = power(x, n/2);
  p = p*p;
}
else {
  p = power(x, (n-1)/2);
  p = x*p*p;
}
```

où p est une variable locale à power de type double
- <résultat récursif> il s'agit simplement de retourner la valeur de p
- <paramètres> le calcul nécessite ici deux valeurs : x de type double et n de type unsigned int

Nous obtenons alors le programme suivant :

```
#include <stdio.h>

double power(double x, unsigned int n) {
  double p;

  if (n==0)
    return 1;
  else {
    if (n%2 == 0) {
      p = power(x, n/2);
      p = p*p;
    }
    else {
      p = power(x, (n-1)/2);
      p = x*p*p;
    }
    return p;
  }
}

int main() {
  float x;
  unsigned int n;

  printf("x: ");
  scanf("%f", &x);
  printf("n: ");
  scanf("%u", &n);

  printf("%f puissance %u=%f\n", x, n, power(x, n));

  return 0;
}
```

Voici un exemple d'exécution de power pour x valant 3.4 et n valant 3 :

```
power(3.4, 3)
   |- p = power(3.4, 1)
   |         |- p = power(3.4, 0)
   |         |         |
   |         |      return 1
   |      p = 3.4*1*1
```

```
   |      return 3.4
p = 3.4*3.4*3.4
return 39.304003
```

Remarque : l'utilisation de la variable p dans la fonction power n'est pas anodin. En effet, en écrivant la fonction de la façon suivante :

```
double power(double x, unsigned int n) {
  if (n==0)
    return 1;
  else {
    if (n%2 == 0) {
      return (power(x, n/2)*power(x, n/2));
    }
    else {
      return (x*power(x, (n-1)/2)*power(x, (n-1)/2));
    }
  }
}
```

nous obtenons un comportement totalement différent, puisque nous avons un nombre exponentiellement plus grand d'appels récursifs. Il y a deux appels à chaque fois :

```
power(3.4, 3)
   |- power(3.4, 1)
   |     |- power(3.4, 0)
   |     |     |
   |     |  return 1
   |     |- power(3.4, 0)
   |     |     |
   |     |  return 1
   |  return 3.4*1*1
   |- power(3.4, 1)
   |     |- power(3.4, 0)
   |     |     |
   |     |  return 1
   |     |- power(3.4, 0)
   |     |     |
   |     |  return 1
   |  return 3.4*1*1
return 3.4*3.4*3.4
```

Même pour des valeurs petites de n (pensez par exemple à n = 10), nous obtenons des temps de calcul invraissemblablement longs, et assez rapidement, l'impossibilité de procéder aux appels récursifs car la taille de la pile est limitée

Nous pouvions cependant faire encore pire en voulant remplacer les deux appels récursifs par l'utilisation de power pour calculer le carrés :

```
double power(double x, unsigned int n) {
```

```
  if (n==0)
    return 1;
  else {
    if (n%2 == 0) {
      return (power(power(x, n/2), 2));
    }
    else {
      return (x*power(power(x, (n-1)/2), 2));
    }
  }
}
```

Ce qui pour l'exemple ci-dessus provoque les appels suivants :

```
power(3.4, 3)
   |- power(3.4, 1)
   |      |- power(3.4, 0)
   |      |      |
   |      |  return 1
   |      |- power(1, 2)
   |      |      |- power(1, 1)
   |      |      |      |- power(1, 0)
   |      |      |      |      |
   |      |      |      |  return 1
   |      |      |      |- power(1, 2)
....
```

Nous voyons le problème : pour calculer `power(1, 2)`, nous avons besoin de `power(1, 1)` qui a lui-même besoin de `power(1,2)`. Notre fonction part dans une infinité (théorique) d'appels récursifs qui se traduit en partique par un débordement de pile et l'impossibilité d'effectuer le calcul.

Il est possible d'utiliser ce mode de calcul en ajoutant des cas d'arrêts pour $\texttt{n} = 1$ et $\texttt{n} = 2$. Ceci s'explique par le fait que pour garantir la terminaison de la fonction `power`, nous avons dit que les valeurs de `n` sont strictement décroissantes par les appels récursifs successifs. Or, sans ces nouveaux cas d'arrêt :

- pour $\texttt{n} = 1$, le calcul récursif s'écrit : `x*power(power(x, (n-1)/2), 2)` qui fait appel à `power(., 2)` donc `n` devient localement croissante
- pour $\texttt{n} = 2$, le calcul récursif s'écrit : `power(power(x, n/2), 2)` qui fait appel à `power(., 2)` laissant `n` inchangée

Ces deux cas mènent donc à des appels récursifs infinis. On retrouve la propriété de stricte décroissance des valeurs de `n`, et la terminaison des appels récursifs, en ajoutant les deux nouveaux cas d'arrêt.

Exercice 8:

On procède à l'analyse de traitement récursif :

- `<type de retour>` la spécification et la définition de la suite de Fibonacci nous donne le type `unsigned int`

- `<cas d'arrêt>` par la spécification, nous avons deux cas d'arrêt : `n` = 0 et `n` = 1. La suffisance de ces deux cas est trivial d'après la définition de la suite de Fibonacci
- `<traitement cas d'arrêt>` il n'y a pas de traitement lié au cas d'arrêt
- `<résultat cas d'arrêt>` pour le cas `n` = 0, nous retournons la valeur 0 : `return 0` et pour le cas `n` = 1, nous retournons la valeur 1 : `return 1`
- `<traitement récursif>` le traitement récursif consiste en deux appels : celui à `fibo(n-1)` et celui à `fibo(n-2)`, puis à sommer le résultat
- `<résultat récursif>` il s'agit de renvoyer la somme précédemment calculée
- `<paramètres>` le calcul nécessite seulement la valeur de `n` de type `unsigned int`

Nous obtenons ainsi le programme suivant :

```
#include <stdio.h>

unsigned int fibo(unsigned int n) {
  if (n == 0)
    return 0;
  else if (n == 1)
    return 1;
  else
    return (fibo(n-1)+fibo(n-2));
}

int main() {
  unsigned int n;

  printf("n: ");
  scanf("%u", &n);
  printf("le %u ieme terme de Fibo. vaut : %u\n", n, fibo(n));

  return 0;
}
```

Voici un exemple d'appels récursifs pour `n` = 4 :

```
fibo(4)
  |- fibo(3)
  |    |- fibo(2)
  |    |    |- fibo(1)
  |    |    |    |
  |    |    |  return 1
  |    |    |- fibo(0)
  |    |    |    |
  |    |    |  return 0
  |    |  return 1+0
  |    |- fibo(1)
  |    |    |
  |    |  return 1
  |  return 1+1
```

```
  |- fibo(2)
  |    |- fibo(1)
  |    |    |
  |    |  return 1
  |    |- fibo(0)
  |    |    |
  |    |  return 0
  |  return 1+0
return 2+1
```

Remarque : nous notons un grand gaspillage de calculs, par exemple `fibo(2)` est calculée deux fois, donnant lieu à de nombreux appels récursifs. Ceci empire de plus avec les valeurs de `n` croissantes. Nous discutons de ce problème dans les deux derniers exercices

Exercice 9:

La décomposition binaire d'un nombre se fait par divisions successives par 2 :

```
14 | 2
   ---
 0 | 7 | 2
       ---
     1 | 3 | 2
           ---
         1 | 1 | 2
               ---
             1 | 0
```

L'encodage binaire est donnée par la succession des restes des division par 2 : 1110 est l'encodage binaire de 14. Notons que nous l'obtenons à l'envers du fait que $14 = 1 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 0 \times 2^1$.

Nous procédons maintenant à l'analyse récursive du problème :

- `<type de retour>` d'après la spécification, il s'agit de `void` (aucune valeur retournée)
- `<cas d'arrêt>` comme nous le voyons sur l'exemple ci-dessus, la décomposition s'arrête lorsque le quotient de la division vaut 0
- `<traitement cas d'arrêt>` dans ce cas, le nombre à décomposer vaut 0 ou 1. Il correspond au digit de poids fort de la décomposition. Il faut donc l'afficher.
- `<résultat cas d'arrêt>` il n'y a pas de résultat
- `<traitement récursif>` puisque la décomposition est obtenue à l'envers et que nous voulons l'afficher, il faut tout d'abord afficher la suite de la décomposition, donc procéder à l'appel récursif, puis afficher le digit courant donné par le reste de la division par 2
- `<résultat récursif>` aucun résultat n'est produit
- `<paramètres>` le calcul nécessite seulement la valeur de `n` de type `unsigned int`

Nous obtenons alors le programme suivant :

```
#include <stdio.h>

void decompose2(unsigned int n) {
```

```
  if (n/2 != 0)
    printf("%u", n);
  else {
    decompose2(n/2);
    printf("%u", n%2);
  }
}

int main() {
  unsigned int n;

  printf("n: ");
  scanf("%u", &n);
  decompose2(n);
  printf("\n");

  return 0;
}
```

Voici un exemple d'appels récursifs de la fonction `decompose2` :

```
decompose2(14)
    |- decompose2(7)
    |       |- decompose2(3)
    |       |       |- decompose2(1)
    |       |       |       |
    |       |       |  printf("%u", 1)
    |       |  printf("%u", 3%2)
    |  printf("%u", 7%2)
printf("%u", 14%2)
```

Remarque : pour notre cas d'arrêt, nous avons `n%2=n`. Si nous pensons en terme de file, nous considérons ici la suite des quotients des divisions par 2. Nous ne connaissons pas cette suite, mais sa butée (0) doit être traitée comme les autres éléments de la file : elle doit être affichée. Ceci nous conduit a une écriture plus compacte de la fonction décompose :

```
void decompose2(unsigned int n) {
  if (n/2 != 0)
    decompose2(n/2);
  printf("%u", n%2);
}
```

Nous voyons ainsi que tous les éléments de la file sont traités de la même façon (le `printf` s'applique à tous les éléments).

Exercice 10:

Ceci n'engendre que très peu de modifications. Essentiellement, la fonction `decompose` doit connaître la valeur de b. Elle prend donc un deuxième paramètre `b` de type `unsigned`

`int`. Par ailleurs, partout où des divisions par 2 étaient utilisées, nous avons maintenant des divisions par `b`. Enfin, la fonction `main` doit lire la valeur de `b` au clavier.

Nous avons donc le programme :

```
#include <stdio.h>

void decompose(unsigned int n, unsigned int b) {
  if (n/b != 0)
    decompose(n/b, b);
  printf("%u", n%b);
}

int main() {
  unsigned int n, b;

  printf("n: ");
  scanf("%u", &n);
  printf("b: ");
  scanf("%u", &b);
  if (b <= 10) {
    decompose(n, b);
    printf("\n");
  }
  else
    printf("b doit etre inferieure a 10\n");

  return 0;
}
```

Exercice 11:

Nous considérons donc maintenant le même principe algorithmique, mais qui utilise des divisions par 16 plutôt que par 2.

La modification essentielle par rapport à la décomposition binaire est que nous ne pouvons pas afficher directement le reste de la division par 16. En effet, en héxadécimal, 10 est noté a, 11 est noté b, et ainsi de suite jusqu'à 15 qui est noté f, complétant ainsi la base. L'affichage du digit devient donc :

```
if (n%16 < 10)
  printf("%u", n%16);
else
  printf("%c", 'a'+n%16-10);
```

Nous obtenons ainsi le programme suivant :

```
#include <stdio.h>

void decompose16(unsigned int n) {
```

```
  if (n/16 != 0)
    decompose16(n/16);
  if (n%16 < 10)
    printf("%u", n%16);
  else
    printf("%c", 'a'+n%16-10);
}

int main() {
  unsigned int n;

  printf("n: ");
  scanf("%u", &n);
  decompose16(n);
  printf("\n");

  return 0;
}
```

Remarque : notez comme il est pratique que le type qui représente les caractères soit un type entier

Exercice 12:

Des fonctions mutuellement récursives sont des fonctions qui s'appellent l'une et l'autre. Nous procédons à l'analyse récursive :

- `<type de retour>` d'après la spécification, nous devons renvoyer un entier : 0 ou 1, nous pouvons donc choisir `int`
- `<cas d'arrêt>` pour la fonction `pair`, nous savons que 0 est pair, et que 1 est impair. Nous avons donc 2 cas d'arrêt pour chaque fonction
- `<traitement cas d'arrêt>` aucun traitement
- `<résultat cas d'arrêt>` la fonction `pair` renvoie 1 pour le cas 0, et 0 pour le cas 1, et la fonction `impair` fait l'inverse
- `<traitement récursif>` un nombre `n` est pair si `n-1` est impair et inversement. Nous procédons donc aux appels récursifs selon ce principe
- `<résultat récursif>` le résultat est celui de l'appel récursif réalisé
- `<paramètres>` le calcul nécessite seulement la valeur de `n` de type `unsigned int`

Nous obtenons alors le programme suivant :

```
#include <stdio.h>

int pair(unsigned int n);
int impair(unsigned int n);

int pair(unsigned int n) {
  if (n == 0)
    return 1;
```

```
  else if (n == 1)
    return 0;
  else
    return impair(n-1);
}

int impair(unsigned int n) {
  if (n == 0)
    return 0;
  else if (n == 1)
    return 1;
  else
  return pair(n-1);
}

int main() {
  unsigned int n;

  printf("n: ");
  scanf("%u", &n);
  if (pair(n))
    printf("%u est pair\n", n);
  else
    printf("%u est impair\n", n);

  return 0;
}
```

Remarque : nous devons absolument déclarer `impair` avant de définit `pair` puisque celle-ci utilise `impair`. Ceci est requis pour la compilation. Ici, nous avons déclaré les deux fonctions pour la symétrie.

Voici un exemple d'exécution :

```
pair(5)
  |- impair(4)
  |    |- pair(3)
  |    |    |- impair(2)
  |    |    |    |- pair(1)
  |    |    |    |    |
  |    |    |    |  return 0
  |    |    |  return 0
  |    |  return 0
  |  return 0
return 0
```

Remarque : cette implantation de `pair` et `impair` n'est pas à reproduire car elle est très inefficace du fait des nombreux appels récursifs. Elle sert juste ici a montrer le concept de fonctions mutuellement récursives

Exercice 13:

Les deux fonctions `f1` et `f2` calculent la valeur de factorielle `n`. Cependant, leurs implantations sont radicalement différentes. Voici leurs exécutions pour `n` valant 5, pour `f1` :

```
f1(5)
  |- f1(4)
  |    |- f1(3)
  |    |    |- f1(2)
  |    |    |    |- f1(1)
  |    |    |    |    |- f1(0)
  |    |    |    |    |    |
  |    |    |    |    |  return 1
  |    |    |    |  return 1*1
  |    |    |  return 2*1
  |    |  return 3*2
  |  return 4*6
return 5*24
```

Et pour `f2` :

```
f2(5, 1)
  |- f2(4, 5*1)
  |    |- f2(3, 4*5)
  |    |    |- f2(2, 3*20)
  |    |    |    |- f2(1, 2*60)
  |    |    |    |    |- f2(0, 1*120)
  |    |    |    |    |    |
  |    |    |    |    |  return 120
  |    |    |    |  return 120
  |    |    |  return 120
  |    |  return 120
  |  return 120
return 120
```

La différence essentielle provient du comportement lorsque le cas d'arrêt est atteint. Dans le cas de `f1` le calcul de la factorielle ne fait que commencer. Il a donc été nécessaire à chaque appel de `f1` de mémoriser la valeur de `n` sur la pile pour s'en souvenir lors de la «remontée» et effectuer le calcul.

Dans le cas de `f2`, le calcul est fait lors de la «descente» des appels récursifs. Au moment où le cas d'arrêt est atteint, `r` contient le résultat. Il n'est donc pas nécessaire de conserver la valeur de `n` à chaque appel, puisqu'il n'y a pas de véritable remontée.

Dans le cas de `f2`, nous parlons de «récursivité terminale» car l'appel récursif est la dernière instruction exécutée par la fonction. Les compilateurs modernes savent reconnaître la récursivité terminale et en tirer parti pour ne pas sauvegarder le contexte d'appel. La récursivité terminale devient alors aussi efficace qu'une implantation itérative. En outre, cette technique permet d'optimiser les fonctions récursives comme nous le voyons dans l'exercice suivant.

Exercice 14:

Nous avons vu que l'implantation triviale du calcul récursif de la suite de Fibonacci est très inefficace car il calcule de nombreuses fois le même terme de la suite. Par exemple, pour calculer $f_n = f_{n-1} + f_{n-2}$, nous allons calculer récursivement f_{n-1} et f_{n-2}. Le calcul de f_{n-1} recalculera à nouveau f_{n-2}.

Lorsque nous implantons une fonction récursive, comme la fonction `f1` du calcul précédent, le principe en est le suivant. Nous appelons récursivement la fonction jusqu'à obtenir un cas d'arrêt, puis nous faisons le calcul en «remontant». Pour cela, nous exploitons l'empilement des contextes d'exécution des fonctions qui permet de faire le calcul en arrière (du point de vue de la récursivité). Ceci est l'approche la plus simple car elle correspond à la définition usuelle des suites : à partir de u_0 on peut obtenir u_1, puis u_2, etc.

Le principe de la récursivité terminale est de ne pas attendre la «remontée» pour faire le calcul, mais d'exploiter la «descente» récursive pour cela. Sur l'exemple de la fonction `f2` ci-dessus, nous ajoutons une variable `r` qui stocke le calcule en cours de telle sorte qu'en finissant la descente, `r` contient le résultat. En conséquence de quoi `fact(n, r)` calcule 0!, `fact(n-1, r)` calcule 1!, etc.

Revenons au problème du calcul du `n`ième terme de la suite de Fibonacci. Prenons deux termes successifs (comme signifié ci-dessus, la récursivité terminale calcule en valeur de `n` croissante du point de vue de la suite) :

$$f_n = f_{n-1} + f_{n-2}$$
$$f_{n+1} = f_n + f_{n-1}$$

que nous pouvons représenter comme :

rang	position $n-1$	position $n-2$
n	f_{n-1}	f_{n-2}
$n+1$	f_n	f_{n-1}

nous voyons qu'en passant du rang n au rang $n+1$:

- la position $n-1$ contient la somme des positions $n-1$ et $n-2$ du rang précédent
- la position $n-2$ contient la position $n-1$ du rang précédent

Nous en déduisons alors le principe récursif de notre fonction `fibo`. Celle comprend, en plus du paramètres `n`, les valeurs en position $n-1$ (nommée `f1`) et $n-2$ (nommée `f2`) :

```
unsigned int fibo(unsigned int n, unsigned int f1, unsigned int f2) {
  ...
  return fibo(n-1, f1+f2, f1);
}
```

Il nous reste maintenant à définir les cas d'arrêt. Nous les connaissons d'après la définition de la suite de Fibonacci : `n` valant 0 ou 1. Il faut donc initialiser `f1` et `f2` de façon que chacune corresponde à l'un de ces cas. Pour $n \geq 1$, le résultat sera contenu dans `f1` (par le principe récursif). Nous parvenons donc à la fonction suivante :

```
unsigned int fibo(unsigned int n, unsigned int f1, unsigned int f2) {
  if (n == 0)
```

```
    return f2;
  else if (n == 1)
    return f1;
  else
    return fibo(n-1, f1+f2, f1);
}
```

Le programme global devient donc :

```
#include <stdio.h>

unsigned int fibo(unsigned int n, unsigned int f1, unsigned int f2) {
  if (n == 0)
    return f2;
  else if (n == 1)
    return f1;
  else
    return fibo(n-1, f1+f2, f1);
}

unsigned int fibo_stub(unsigned int n) {
  return fibo(n, 1, 0);
}

int main() {
  unsigned int n;

  printf("n: ");
  scanf("%u", &n);
  printf("le %u ieme terme de Fibo. vaut : %u\n", n, fibo_stub(n));

  return 0;
}
```

Remarque : la fonction `fibo_stub` sert à initialiser correctement les valeurs de `f1` et `f2`

Voici un exemple d'exécution de la fonction `fibo` :

```
fibo_stub(4)
    |- fibo(4, 1, 0)
    |    |- fibo(3, 1, 1)
    |    |    |- fibo(2, 2, 1)
    |    |    |    |- fibo(1, 3, 2)
    |    |    |    |    |
    |    |    |    |  return 3
    |    |    |  return 3
    |    |  return 3
    |  return 3
return 3
```

Ceci est à comparer aux appels récursifs du calcul naïvement implanté précédemment. Nous avons ici exactement n appels récursifs, et aucun calcul superflu, contre un nombre exponentiel de calculs dans notre première implantation