Récursivité

Lundi 19 Février 2018

Michael FRANÇOIS

francois@esiea.fr

https://francois.esiea.fr

Histoire de l'itération et de la récursivité

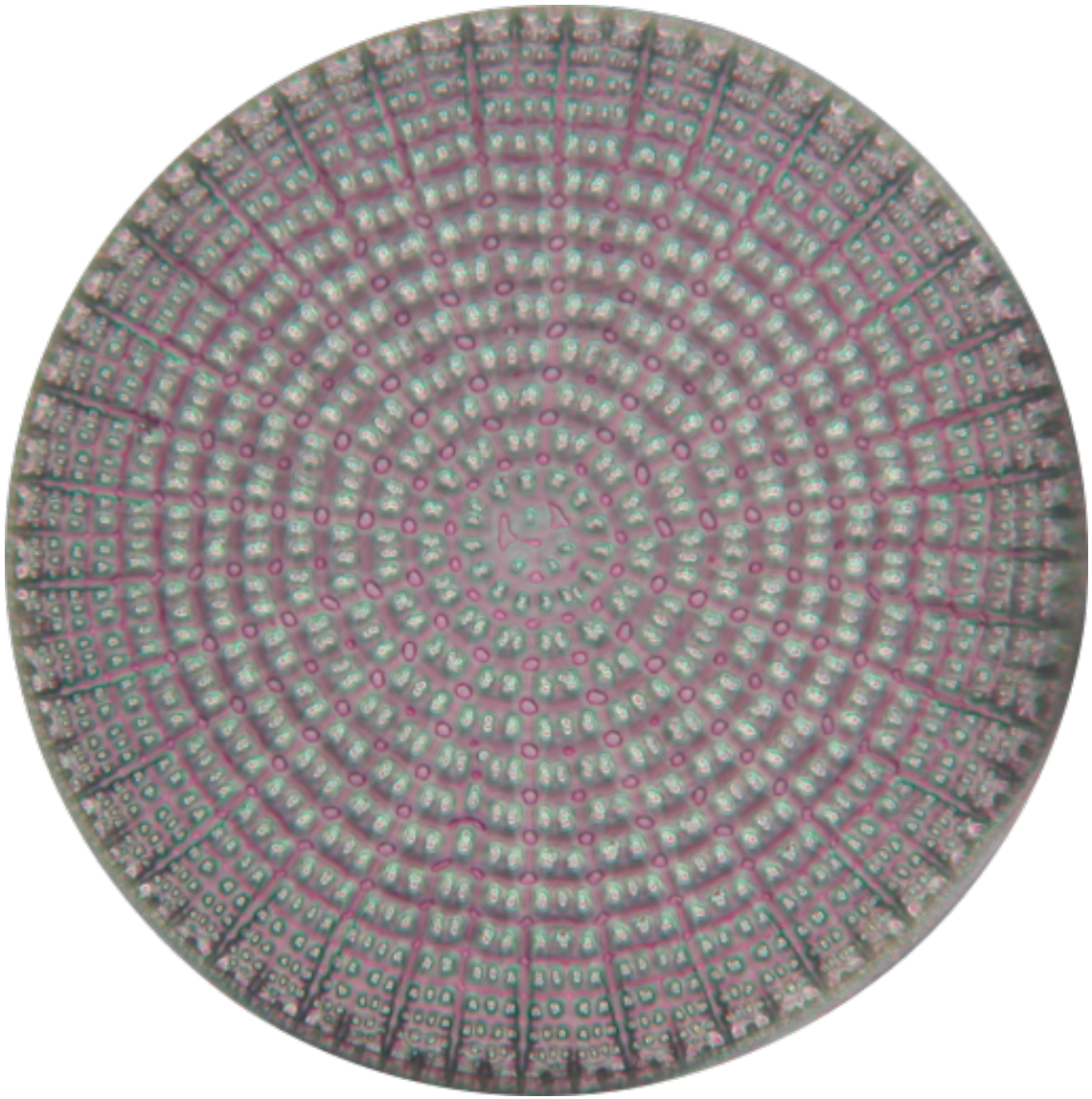
- L'**itération** est définie comme un processus appliquant à plusieurs reprises une règle, un calcul ou une procédure.
  - L'itération fait partie des processus naturels de la vie, exemple : l'année comme étant une itération de 12 mois. Elle était aussi appliquée dans le domaine de la psychanalyse, où il s'agit de poser toujours la dernière question en fonction de la dernière réponse de la personne à soigner.
- La **récursivité** décrit un processus dépendant de données, en faisant appel à ce même processus sur d'autres données.
  - Archimède a proposé un algorithme qui calcule le périmètre d'un polygone à 2^n cotés en fonction de celui d'un polygone à 2^{n-1} cotés. Ce processus a permis d'encadrer la valeur π avec plus de précision. Cette valeur a été déjà découverte dans le papyrus de Rhind.
  - Un autre exemple bien connu des informaticiens est **GNU** (**G**NU's **N**ot **U**nix), qui est un acronyme récursif. On trouve également d'autres acronymes récursifs comme :
    - **VISA** : **V**ISA **I**nternational **S**ervice **A**ssociation
    - **EINE** : **E**INE **I**s **N**ot **E**macs
    - ...

Récursivité dans la nature

La récursivité est particulièrement présente en biologie, notamment dans les motifs de végétaux et les processus de développement.

Coupe sagittale d'une coquille de nautile

Une fleur de tournesol



Une diatomée centrale

Programmation récursive

Programmation récursive

• Un **programme récursif** est tout simplement un programme qui s'appelle lui même, ou qu'une fonction est récursive si elle est définie en référence à elle-même.

• Les appels récursifs sont de plus en plus petits jusqu'à arriver au cas limite (cas de base de la récursion qui doit être bien traité par l'algorithme) qui termine immédiatement le processus.

• Une condition de terminaison est donc nécessaire pour indiquer au programme à ne plus faire appel à lui même ou à la fonction à ne plus être définie en référence à elle-même.

Une fonction récursive est une fonction qui s'appelle elle-même. On distingue 4 types de récursivité :

1. Simple
2. Multiple
3. Terminale
4. Mutuelle

1. Une fonction qui s'appelle elle même une seule fois est dite **simplement récursive**.
2. Une fonction qui s'appelle elle même plus d'une fois est dite **récursive multiple** (les plus dangereuses).
3. Une fonction récursive peut également être **terminale**, dans le cas où tout appel récursif est de la forme : `return F(...);`
4. Quand une fonction F appelle une fonction G et que la fonction G appelle également la fonction F, dans ce cas on parle de **récursivité mutuelle**.

Remarque : tout programme récursif peut être écrit par itération et tout programme itératif peut être écrit récursivement. On peut évidement passer de l'itération à la récursivité. De la même manière on peut passer de la récursivité à l'itération.

Récursivité simple

• Considérons un ensemble de n objets distincts, combien existe-t-il de façons de les classer ?

• Initialement on a n possibilités de choisir un objet quelconque parmi n objets, puis $n-1$ possibilités de choisir le second, puis $n-2$ ... Ce qui donne $n(n-1)(n-2)\cdots 1 = n!$ possibilités.

• D'après la formule mathématique $n! = n(n-1)!$, on voit que la fonction factorielle existe dans les deux cotés de l'équation, on dit que cette fonction est auto-référente.

Exemple de fonction factorielle en C :

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

int FACT_REC_SIMPLE(int n)
{
  if (n==0 || n==1) return 1;
  return n*FACT_REC_SIMPLE(n-1);
}

int main(int argc, char** argv)
{
  printf("%d! = %d\n", abs(atoi(argv[1])), FACT_REC_SIMPLE(abs(atoi(argv[1]))));
  return 0;
}
```

Que veut dire ?

- `argv[1]` ?
- `atoi` ?
- `abs` ?

Exemple de fonction factorielle en C :

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

int FACT_REC_SIMPLE(int n)
{
  if (n==0 || n==1) return 1;
  return n*FACT_REC_SIMPLE(n-1);
}

int main(int argc, char** argv)
{
  printf("%d! = %d\n", abs(atoi(argv[1])), FACT_REC_SIMPLE(abs(atoi(argv[1]))));
  return 0;
}
```

Remarques :

- `argv[1]` $\Longrightarrow$ donne le premier argument qui suit le nom de l'exécutable.
- `atoi` $\Longrightarrow$ pour convertir l'argument en entier qui est à la base une chaîne de caractères.
- `abs` $\Longrightarrow$ valeur absolue.

Exécution :

```
$ gcc -o EXEC test.c -Wall
$ ./EXEC 2
2! = 2
$ ./EXEC 5
5! = 120
$ ./EXEC -5
5! = 120
$ ./EXEC
Segmentation fault (core dumped)
$ ./EXEC 21
21! = -1195114496
$
```

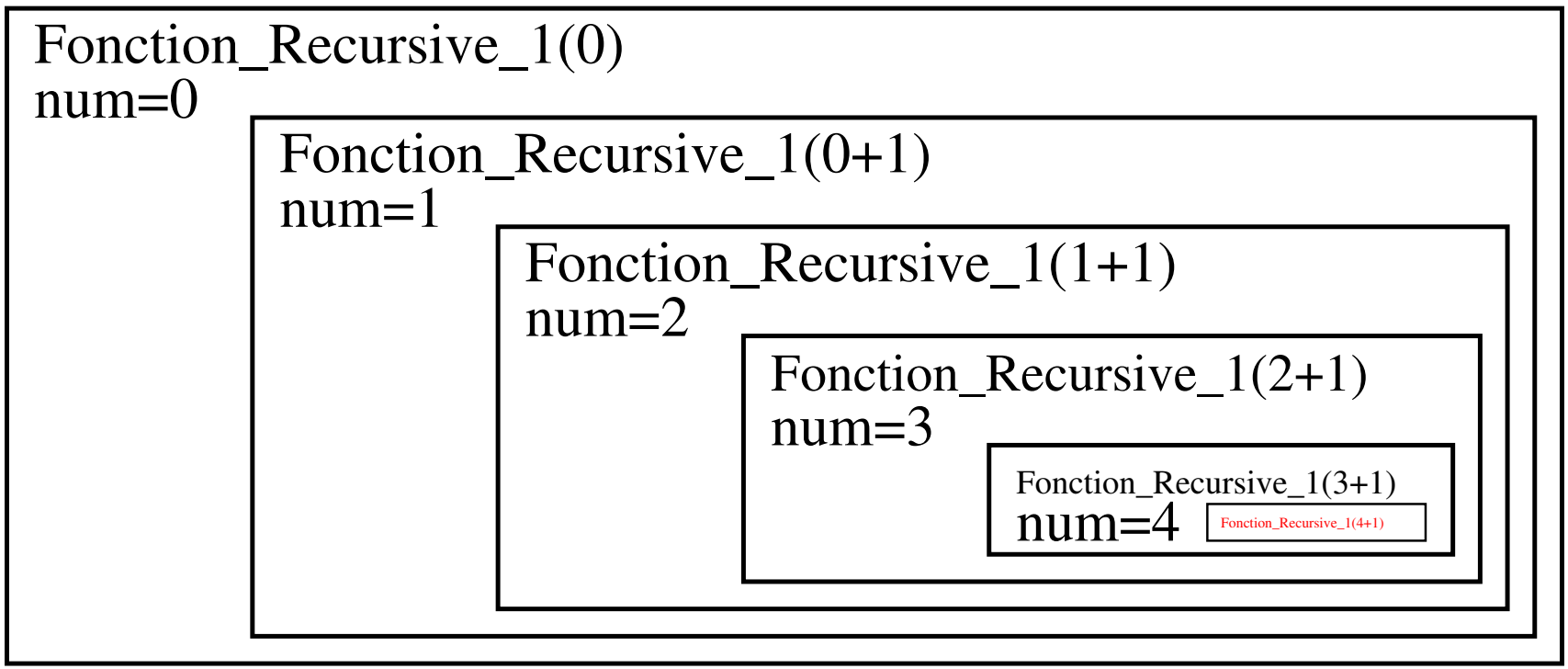
Considérons l'exemple suivant :

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

void Fonction_Recursive_1(int num)
{
 printf("Lancement => Fonction_Recursive_1(%d)\n", num);
 if(num < 5)
 {
   printf("Exécution pour num = %d\n\n", num);
   Fonction_Recursive_1(num+1);
 }
}

int main(int argc, char** argv)
{
  Fonction_Recursive_1(0);
  return 0;
}
```

Exécution :

```
$ gcc -o EXEC test.c -Wall
$ ./EXEC
Lancement => Fonction_Recursive_1(0)
Exécution pour num = 0

Lancement => Fonction_Recursive_1(1)
Exécution pour num = 1

Lancement => Fonction_Recursive_1(2)
Exécution pour num = 2

Lancement => Fonction_Recursive_1(3)
Exécution pour num = 3

Lancement => Fonction_Recursive_1(4)
Exécution pour num = 4

Lancement => Fonction_Recursive_1(5)
$
```

Pour mieux comprendre la succession des appels sur la fonction : `Fonction_Recursive_1`



Remarque : on voit que dans la fonction `Fonction_Recursive_1`, toutes les instructions sont placées **avant** l'appel récursif qui se situe en dernier lieu. Maintenant comment ça se passe dans le cas où l'appel récursif est situé avant d'autres instructions ?

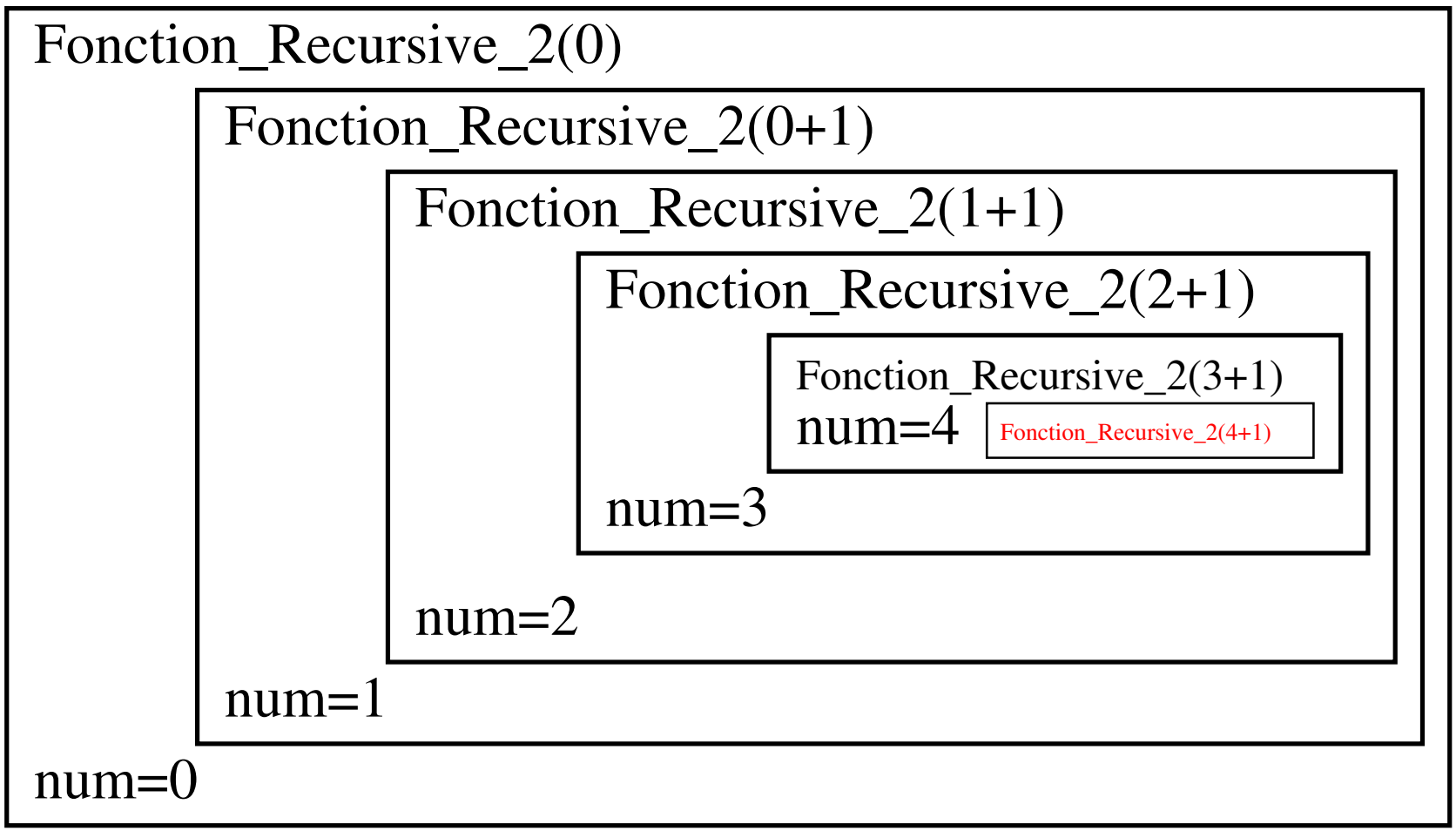
Considérons maintenant la version modifiée suivante :

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

void Fonction_Recursive_2(int num)
{
 printf("Lancement => Fonction_Recursive_2(%d)\n", num);
 if(num < 5)
 {
   Fonction_Recursive_2(num+1);
   printf("Exécution pour num = %d\n\n", num);
 }
}

int main(int argc, char** argv)
{
  Fonction_Recursive_2(0);
  return 0;
}
```

Exécution :

```
$ gcc -o EXEC test.c -Wall
$ ./EXEC
Lancement => Fonction_Recursive_2(0)
Lancement => Fonction_Recursive_2(1)
Lancement => Fonction_Recursive_2(2)
Lancement => Fonction_Recursive_2(3)
Lancement => Fonction_Recursive_2(4)
Lancement => Fonction_Recursive_2(5)
Exécution pour num = 4

Exécution pour num = 3

Exécution pour num = 2

Exécution pour num = 1

Exécution pour num = 0
```

Pour mieux comprendre la succession des appels sur la fonction : `Fonction_Recursive_2`



Remarque : au moment des premiers passages, le printf n'est pas exécuté car le programme rentre dans une sorte de traitement interne en retournant à chaque fois au tout début de la fonction. Donc par ordre de priorité, les printf qui sont en interne seront d'abord exécutés pour finir avec le printf initial. C'est pourquoi *num* est affiché de manière décroissante.

Récursivité multiple

• On parle de récursivité multiple dans le cas où il y a plusieurs appels récursifs dans le corps de la procédure.

• Prenons l'exemple de la suite de **Fibonacci** qui est définie par :

$$\begin{cases} F(0) = 0, F(1) = 1 \\ F(n) = F(n-1) + F(n-2), n > 1 \end{cases}$$

F(0)	F(1)	F(2)	F(3)	F(4)	F(5)	F(6)	...
0	1	1	2	3	5	8	...

Code C

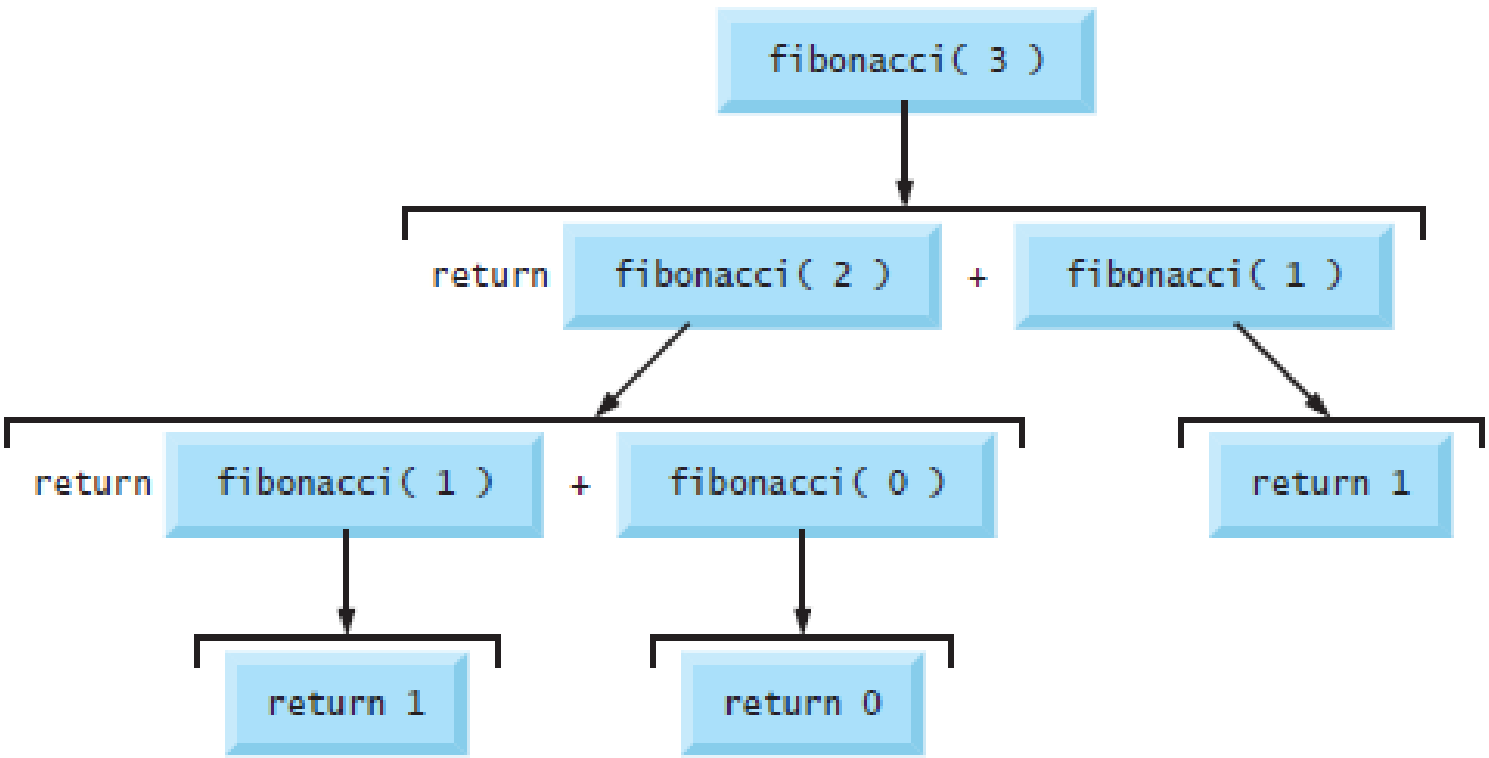
```c
#include <stdio.h>
#include <stdlib.h>

int FIBONACCI(int n)
{
  printf("Lancement de FIBONACCI(%d)\n", n);
  if (n==0) return 0;
  if (n==1) return 1;
  return FIBONACCI(n-1) + FIBONACCI(n-2);
}

int main(int argc, char** argv)
{
  printf("FIBONACCI(%d) = %d\n", abs(atoi(argv[1])), FIBONACCI(abs(atoi(argv[1]))));
  return 0;
}
```

Exécution

```
$ gcc -o EXEC test.c -Wall
$ ./EXEC 2
Lancement de FIBONACCI(2)
Lancement de FIBONACCI(1)
Lancement de FIBONACCI(0)
FIBONACCI(2) = 1
$ ./EXEC 3
Lancement de FIBONACCI(3)
Lancement de FIBONACCI(2)
Lancement de FIBONACCI(1)
Lancement de FIBONACCI(0)
Lancement de FIBONACCI(1)
FIBONACCI(3) = 2
$
```

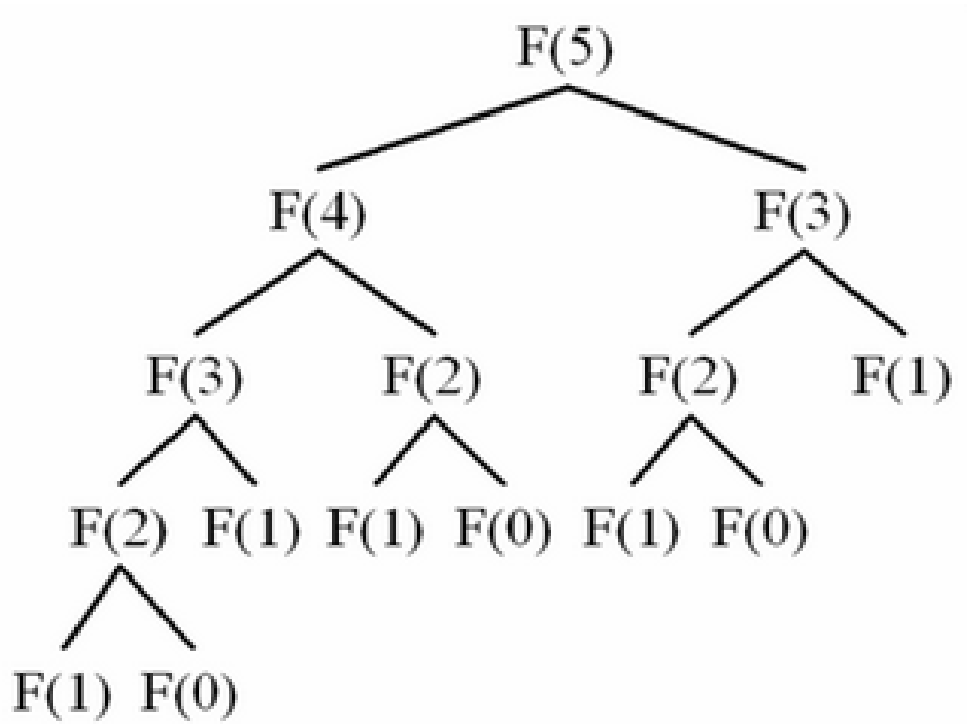
Pour $n = 3$

```
$ ./EXEC 3
Lancement de FIBONACCI(3)
Lancement de FIBONACCI(2)
Lancement de FIBONACCI(1)
Lancement de FIBONACCI(0)
Lancement de FIBONACCI(1)
FIBONACCI(3) = 2
```



Pour $n = 5$

```
$ ./EXEC 5
Lancement de FIBONACCI(5)
Lancement de FIBONACCI(4)
Lancement de FIBONACCI(3)
Lancement de FIBONACCI(2)
Lancement de FIBONACCI(1)
Lancement de FIBONACCI(0)
Lancement de FIBONACCI(1)
Lancement de FIBONACCI(2)
Lancement de FIBONACCI(1)
Lancement de FIBONACCI(0)
Lancement de FIBONACCI(3)
Lancement de FIBONACCI(2)
Lancement de FIBONACCI(1)
Lancement de FIBONACCI(0)
Lancement de FIBONACCI(1)
FIBONACCI(5) = 5
```



Récursivité terminale

• On parle de récursivité terminale ou finale, dans le cas où l'appel récursif est la dernière instruction à être évaluée dans la fonction récursive.

• Revenons sur l'exemple de la fonction factorielle et donnons à présent la version récursive terminale.

Exemple pour la factorielle en version récursive terminale

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


unsigned long int FACT_REC_TERMINALE(int n, unsigned long int resultat)
{
  if (n==0 || n==1) return resultat;
  return FACT_REC_TERMINALE(n-1, n*resultat);
}

int main(int argc, char** argv)
{
  printf("%d! = %lu\n", abs(atoi(argv[1])), FACT_REC_TERMINALE(abs(atoi(argv[1])),1));
  return 0;
}
```

NB : le paramètre `resultat` joue le rôle d'un accumulateur.

Exécution

```
$ gcc -o EXEC test.c -Wall
$ ./EXEC 1
1! = 1
$ ./EXEC 3
3! = 6
$ ./EXEC 10
10! = 3628800
$ ./EXEC 21
21! = 14197454024290336768
$
```

NB : l'évaluation de `FACT_REC_TERMINALE(3,1)` conduit à la suite d'appels :

`FACT_REC_TERMINALE(3,1)`
↓
`FACT_REC_TERMINALE(2,3)`
↓
`FACT_REC_TERMINALE(1,6)`
⇓
6

Récursivité mutuelle

• On parle de récursivité mutuelle dans le cas où deux fonctions s'appellent mutuellement :

• Récursivité mutuelle :
  - F appelle la fonction G
  - G appelle la fonction F

Exemple : les fonctions `PAIR` et `IMPAIR` s'appellent mutuellement :

```c
#include <stdio.h>

int PAIR (int n);

int IMPAIR (int n)
{
  if (n == 0) return 0;
  else return PAIR (n-1);
}

int PAIR (int n)
{
  if (n == 0) return 1;
  else return IMPAIR (n-1);
}

int main()
{
  printf("%d %d\n", PAIR (4), IMPAIR(4));
  return 0;
}
```

Exécution :

```
$ gcc -o EXEC test.c -Wall
$ ./EXEC
1 0
```

Stratégies gauche-droite / droite-gauche et récursivité

Exemple de fonction récursive permettant de calculer le nombre de chiffres constituant un nombre (en base 10) :

* Quelle va être la condition d'arrêt du processus ?
* Sur quelle quantité de donnée en entrée sera effectué l'appel récursif ?

Exemple de fonction récursive permettant de calculer le nombre de chiffres constituant un nombre (en base 10) :

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

int Long_Nbre(int n)
{
  if (n<10) return 1;
  else return 1+Long_Nbre(n/10);
}

int main(int argc, char** argv)
{
  printf("%d ---> %d\n", abs(atoi(argv[1])), Long_Nbre(abs(atoi(argv[1]))));
  return 0;
}
```

Exécution

```
$ gcc -o EXEC test.c -Wall
$ ./EXEC 8
8 ---> 1
$ ./EXEC 5613
5613 ---> 4
$ ./EXEC -81035415
81035415 ---> 8
$ ./EXEC 1000001
1000001 ---> 7
$
```

Exemple de fonction récursive terminale permettant de calculer le miroir d'un nombre (en base 10) :

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

int Miroir_Rec_Term(int n, int resultat)
{
  if (n==0)
    return(resultat);
  else
    return Miroir_Rec_Term(n/10, 10*resultat + n%10);
}

int main(int argc, char** argv)
{
  printf("%d ---> %d\n", abs(atoi(argv[1])), Miroir_Rec_Term(abs(atoi(argv[1])), 0));
  return 0;
}
```

Exécution

```
$ gcc -o EXEC test.c -Wall
$ ./EXEC 2
2 ---> 2
$ ./EXEC 201593
201593 ---> 395102
$ ./EXEC -02381
2381 ---> 1832
$ ./EXEC 62000
62000 ---> 26
$
```

Exemple de fonction récursive permettant de calculer le miroir d'un nombre (en base 10), proposée par un étudiant (BROQUET Nicolas - 17/02/2010) :

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

int Length(int n)
{return ceil(log(n)/log(10));}

int Miroir_Rec_Prop_Par_Etu(int n)
{
  if(n>=10)
    return (n%10)*pow(10, Length(n)-1) + Miroir_Rec_Prop_Par_Etu(n/10);
  else
    return n;
}

int main(int argc, char** argv)
{
  printf("%d ---> %d\n", abs(atoi(argv[1])), Miroir_Rec_Prop_Par_Etu(abs(atoi(argv[1]))));
  return 0;
}
```

NB : cette fonction n'est bien sur pas terminale.

Exécution

```
$ gcc -o EXEC test.c -Wall -lm
$ ./EXEC 2
2 ---> 2
$ ./EXEC 10
10 ---> 1
$ ./EXEC 845169
845169 ---> 961548
$ ./EXEC 020085040
20085040 ---> 4058002
$ ./EXEC -94
94 ---> 49
```

Avantages et inconvénients des fonctions récursives

Récursivité

• La notion de récursivité est une notion importante en algorithmique. Vous allez certainement utiliser des fonctions récursives dans vos futurs projets en programmation, et même au delà.

• C'est une notion qui est également très proche du raisonnement par récurrence vu au lycée.

• Cependant, l'utilisation des fonctions récursives présente des points positifs comme négatifs.

Avantages des fonctions récursives

- La programmation récursive est très élégante et permet de traiter des problèmes complexes avec des outils simples.
- Elle nécessite souvent peu de variables, ce qui donne un programme beaucoup plus lisible et compréhensible.
- En cas d'imbrication complexe dans le code, on peut simplifier en traitant récursivement les sous-problèmes liés.

Inconvénients des fonctions récursives

- La fonction récursive est souvent difficile à mettre en place par rapport à une fonction itérative qui est plus intuitive.
- Une fonction récursive mal-écrite donne aussitôt des résultats absurdes, car les erreurs s'accumulent rapidement via les appels récursifs.
- Si les appels ne sont pas bien maîtrisés, la fonction ne terminera pas en général.
- Il est souvent difficile de déboguer un programme récursif.
- Les appels récursifs successifs nécessitent de sauvegarder le contexte de l'appel dans la pile (chaque appel à la fonction possède son propre espace mémoire), ce qui est très coûteux en mémoire.

Exercice donné au contrôle TD : 18/03/2015

1. Écrire une fonction itérative qui prend en entrée deux entiers a et b et qui renvoie la somme de tous les entiers multiples de 3 compris strictement entre a et b.
2. Écrire une version récursive terminale de la fonction précédente.

Fonction itérative :

```
fonction SOMME_MULTI_3_ITER(a, b)
    Res ⟵ 0 /*Initialisation du résultat*/
    pour i allant de a+1 à b-1 inclus faire
        si (i mod 3 = 0) alors
            Res ⟵ Res + i /*On ajoute i à Res que quand i est un multiple de 3*/
        fin
    fin
    Retourner Res
```

Fonction récursive terminale :

```
fonction SOMME_MULTI_3_REC(a, b, Res)
    E ⟵ a+1
    si (E = b) alors
        Retourner Res
    fin
    si (E mod 3 ≠ 0) alors
        E ⟵ 0
    fin
    Retourner SOMME_MULTI_3_REC(a+1, b, E+Res)
```

Bibliographie

• R. ERRA, "Cours 4 d'informatique", 1A-S2 2013-2014 ESIEA.

• R. Sedgewick, "Algorithmes en langage C", 2005, DUNOD.

• V. PRINCE, "Différence épistémologique entre itération et récursion", TER Master Informatique, Univ. de Montpellier II, Novembre 2007.

• http://fr.wikipedia.org/wiki/Récursivité

• https://stackoverflow.com/questions/20874888/tracing-recursion-for-fibonacci-series