# La récursivité

 La récursivité est une notion importante de la programmation qui permet de régler des problèmes extrêmement complexes avec seulement quelques lignes. C’est cependant une méthode avec laquelle il est facile de se perdre et d’avoir des résultats imprévisibles ou erronés. Lorsque bien utilisée, c’est cependant un outil simplifiant, lisible, efficace et souvent sous estimé.

 Entrons dans le vif du sujet avec de se perdre. La récursivité est un mécanisme assez naturel qui permet à une fonction d’effectuer un appel d’elle même. Par exemple :

|  |  |
| --- | --- |
| 01234 | Addiction (N) SI (N > 1) RETOURNE N + Addiction(N-1) SINON RETOURNE 1 |

##  La fonction Addition est récursive puisqu’elle comprend un appel de Addiction à la ligne 3. Mais pour qu’une fonction récursive fonctionne, elle doit nécessairement contenir une condition sur cet appel. En effet, si une fonction récursive n’a pas de condition, elle effectuera l’appel à chaque itération et une erreur se produira lorsque la pile (voir plus loin) sera pleine. Cette condition peut-être un SI… ALORS…, un POUR…FAIRE… ou un FAIRE … TANT QUE… FAIRE… Quoiqu’il soit beaucoup plus naturel de retrouver un simple SI… ALORS…SINON. Analysons maintenant la récursion lors de l’implantation de la fonction Addiction en C.

|  |  |
| --- | --- |
| 012345678910111213141516171819202122232425262728 | #include <stdio.h>#include <stdlib.h>int Addiction (int N);int main (){ int Nombre =3; printf ("Quel nombre voulez vous?"); scanf ("%i", &Nombre); if (Nombre > 1000000) exit (-1); printf ("Le resultat est : %i", Addiction (Nombre)); return 0;}int Addiction (int N){ int Retour; printf ("Debut, valeur: %i\n",N); if (N > 1) Retour = N + Addiction (N-1); else Retour = 1; printf ("Fin, valeur: %i, Retour: %i\n", N, Retour); return Retour;} |

Voyons ce qui se passe. Premièrement lorsqu’un appel d’une fonction est fait, l’information sur cette fonction (ligne en exécution, adresse des variables, etc) sont mises dans la pile et le saut est fait. Lorsque la fonction est terminée, l’information est pris dans la pile. Prenons un exemple : la fonction Addiction ayant une valeur de 3 reçue.

Premièrement, voici ce qui apparaît sur la fenêtre console :

Quel nombre voulez vous?3

Debut, valeur: 3

Debut, valeur: 2

Debut, valeur: 1

Fin, valeur: 1, Retour: 1

Fin, valeur: 2, Retour: 3

Fin, valeur: 3, Retour: 6

Le resultat est : 6

Deuxièmement, effectuons une trace d’exécution sur cette entrée. Ajoutons cependant un nouvel élément, la pile. Lorsqu’un appel est effectué, nous allons garder dans une pile la ligne qui effectue un appel et l’adresse des variables. Lorsqu’une fonction effectue son retour, la ligne sur le dessus de la pile sera dépilée et l’exécution se continuera à cette ligne. Les variables en caractère gras sont celles dont la ligne en exécution est dans la portée. Aux lignes qui ne font que de l’impression à l’écran, nous ne copierons pas le tableau, pour simplifier.

Supposons que les lignes 1 à 11 ont été exécutées et que nous avons à la ligne 12 la situation suivante :

12 : printf ("Le resultat est : %i", Addiction (Nombre));

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Adresse | Identificateur | Valeur |  | Pile |
| AF01 | … | ? |  |  |
| AF02 | Nombre | 3 |  |  |
| AF03 | … |  |  |  |
| AF04 | … |  |  |  |
| AF05 | … |  |  |  |
| AF06 | … |  |  |  |
| AF07 | … |  |  |  |
| AF08 | … |  |  |  |

16 : int Addiction (int N)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Adresse | Identificateur | Valeur |  | Pile |
| AF01 | … |  |  |  |
| AF02 | Nombre | 3 |  |  |
| AF03 | **N** | 3 |  |  |
| AF04 | … |  |  |  |
| AF05 | … |  |  |  |
| AF06 | … |  |  |  |
| AF07 | … |  |  | 🡻 |
| AF08 | … |  |  | 12, AF02 |

18 : int Retour ;

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Adresse | Identificateur | Valeur |  | Pile |
| AF01 | … |  |  |  |
| AF02 | Nombre | 3 |  |  |
| AF03 | **N** | 3 |  |  |
| AF04 | Retour |  |  |  |
| AF05 | … |  |  |  |
| AF06 | … |  |  |  |
| AF07 | … |  |  |  |
| AF08 | … |  |  | 12, AF02 |

19 : printf ("Debut, valeur: %i\n",N);

21 : if (N>1)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Adresse | Identificateur | Valeur |  | Pile |
| AF01 | … |   |  |  |
| AF02 | Nombre | 3 |  |  |
| AF03 | **N** | 3 |  |  |
| AF04 | Retour |  |  |  |
| AF05 | … |  |  |  |
| AF06 | … |  |  |  |
| AF07 | … |  |  |  |
| AF08 | … |  |  | 12, AF02 |

22 : Retour = N + Addiction (N-1)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Adresse | Identificateur | Valeur |  | Pile |
| AF01 | … |  |  |  |
| AF02 | Nombre | 3 |  |  |
| AF03 | **N** | 3 |  |  |
| AF04 | Retour |  |  |  |
| AF05 | … |  |  |  |
| AF06 | … |  |  |  |
| AF07 | … |  |  |  |
| AF08 | … |  |  | 12, AF02 |

16 : int Addiction (int N)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Adresse | Identificateur | Valeur |  | Pile |
| AF01 | … |  |  |  |
| AF02 | Nombre | 3 |  |  |
| AF03 | N | 3 |  |  |
| AF04 | Retour |  |  |  |
| AF05 | **N** | 2 |  |  |
| AF06 | … |  |  | 🡻 |
| AF07 | … |  |  | 22, AF03, AF04 |
| AF08 | … |  |  | 12, AF02 |

18 : int Retour ;

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Adresse | Identificateur | Valeur |  | Pile |
| AF01 | … |  |  |  |
| AF02 | Nombre | 3 |  |  |
| AF03 | N | 3 |  |  |
| AF04 | Retour |  |  |  |
| AF05 | **N** | 2 |  |  |
| AF06 | **Retour** |  |  |  |
| AF07 | … |  |  | 22, AF03, AF04 |
| AF08 | … |  |  | 12, AF02 |

19 : printf ("Debut, valeur: %i\n",N);

21 : if (N > 1)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Adresse | Identificateur | Valeur |  | Pile |
| AF01 | … |  |  |  |
| AF02 | Nombre | 3 |  |  |
| AF03 | N | 3 |  |  |
| AF04 | Retour |  |  |  |
| AF05 | **N** | 2 |  |  |
| AF06 | **Retour** |  |  |  |
| AF07 | … |  |  | 22, AF03, AF04 |
| AF08 | … |  |  | 12, AF02 |

22 : Retour = N + Addiction (N-1) ;

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Adresse | Identificateur | Valeur |  | Pile |
| AF01 | … |  |  |  |
| AF02 | Nombre | 3 |  |  |
| AF03 | N | 3 |  |  |
| AF04 | Retour |  |  |  |
| AF05 | **N** | 2 |  |  |
| AF06 | **Retour** |  |  |  |
| AF07 | … |  |  | 22, AF03, AF04 |
| AF08 | … |  |  | 12, AF02 |

16 : int Addiction (int N)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Adresse | Identificateur | Valeur |  | Pile |
| AF01 | … |  |  |  |
| AF02 | Nombre | 3 |  |  |
| AF03 | N | 3 |  |  |
| AF04 | Retour |  |  |  |
| AF05 | N | 2 |  | 🡻 |
| AF06 | Retour |  |  | 22, AF05, AF06 |
| AF07 | N | 1 |  | 22, AF03, AF04 |
| AF08 | … |  |  | 12, AF02 |

18 : int Retour ;

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Adresse | Identificateur | Valeur |  | Pile |
| AF01 | … |  |  |  |
| AF02 | Nombre | 3 |  |  |
| AF03 | N | 3 |  |  |
| AF04 | Retour |  |  |  |
| AF05 | N | 2 |  |  |
| AF06 | Retour |  |  | 22, AF05, AF06 |
| AF07 | N | 1 |  | 22, AF03, AF04 |
| AF08 | Retour |  |  | 12, AF02 |

19 : printf ("Debut, valeur: %i\n",N);

21 : if (N > 1)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Adresse | Identificateur | Valeur |  | Pile |
| AF01 | … |  |  |  |
| AF02 | Nombre | 3 |  |  |
| AF03 | N | 3 |  |  |
| AF04 | Retour |  |  |  |
| AF05 | N | 2 |  |  |
| AF06 | Retour |  |  | 22, AF05, AF06 |
| AF07 | N | 1 |  | 22, AF03, AF04 |
| AF08 | Retour |  |  | 12, AF02 |

24 : Retour = 1 ;

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Adresse | Identificateur | Valeur |  | Pile |
| AF01 | … |  |  |  |
| AF02 | Nombre | 3 |  |  |
| AF03 | N | 3 |  |  |
| AF04 | Retour |  |  |  |
| AF05 | N | 2 |  |  |
| AF06 | Retour |  |  | 22, AF05, AF06 |
| AF07 | N | 1 |  | 22, AF03, AF04 |
| AF08 | Retour | 1 |  | 12, AF02 |

26 : printf (« Fin, valeur : %i, Retour : %i\n », N, Retour) ;

27 : return Retour ;

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Adresse | Identificateur | Valeur |  | Pile |
| AF01 | … |  |  |  |
| AF02 | Nombre | 3 |  |  |
| AF03 | N | 3 |  |  |
| AF04 | Retour |  |  |  |
| AF05 | N | 2 |  |  |
| AF06 | Retour |  |  | 22, AF05, AF06 |
| AF07 | N | 1 |  | 22, AF03, AF04 |
| AF08 | Retour | 1 |  | 12, AF02 |

22 : Retour = N + 1

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Adresse | Identificateur | Valeur |  | Pile |
| AF01 | … |  |  |  |
| AF02 | Nombre | 3 |  |  |
| AF03 | N | 3 |  |  |
| AF04 | Retour |  |  |  |
| AF05 | **N** | 2 |  |  |
| AF06 | **Retour** | 3 |  | 🡹 |
| AF07 |  |  |  | 22, AF03, AF04 |
| AF08 |  |  |  | 12, AF02 |

26 : printf (« Fin, valeur : %i, Retour : %i\n », N, Retour) ;

27 return Retour ;

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Adresse | Identificateur | Valeur |  | Pile |
| AF01 | … |  |  |  |
| AF02 | Nombre | 3 |  |  |
| AF03 | N | 3 |  |  |
| AF04 | Retour |  |  |  |
| AF05 | **N** | 2 |  |  |
| AF06 | **Retour** | 3 |  |  |
| AF07 |  |  |  | 22, AF03, AF04 |
| AF08 |  |  |  | 12, AF02 |

22 : Retour = N + 3

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Adresse | Identificateur | Valeur |  | Pile |
| AF01 | … |  |  |  |
| AF02 | Nombre | 3 |  |  |
| AF03 | **N** | 3 |  |  |
| AF04 | Retour | 6 |  |  |
| AF05 |  |  |  |  |
| AF06 |  |  |  |  |
| AF07 |  |  |  | 🡹 |
| AF08 |  |  |  | 12, AF02 |

26 : printf (« Fin, valeur : %i, Retour : %i\n », N, Retour) ;

27 : return Retour;

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Adresse | Identificateur | Valeur |  | Pile |
| AF01 | … |  |  |  |
| AF02 | Nombre | 3 |  |  |
| AF03 | **N** | 3 |  |  |
| AF04 | Retour | 6 |  |  |
| AF05 |  |  |  |  |
| AF06 |  |  |  |  |
| AF07 |  |  |  |  |
| AF08 |  |  |  | 12, AF02 |

12 : 12 : printf ("Le resultat est : %i", 6);

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Adresse | Identificateur | Valeur |  | Pile |
| AF01 | … |  |  |  |
| AF02 | Nombre | 3 |  |  |
| AF03 |  |  |  |  |
| AF04 |  |  |  |  |
| AF05 |  |  |  |  |
| AF06 |  |  |  |  |
| AF07 |  |  |  |  |
| AF08 |  |  |  | 🡹 |

13 : return 0 ;

Retour à l’OS

**Les problématiques récursives :**

 Il existe certains problèmes dont la définition est récursive. Ce genre de problème se programme simplement à l’aide d’une fonction récursive. Les problèmes qui se solutionnent à l’aide de la récursivité peuvent toujours être résolu à l’aide d’algorithme séquentiel non-récursif, mais souvent cela est plus complexe. Sans en apporter ici la preuve formel, donnons tout de même deux exemples précis : la suite de Fibonnace et la tour d’Hanoi.

Fibonnaci :

La suite de Fibonnaci est définie comme suit :

|  |  |
| --- | --- |
| n | fib(n) |
| 0 | 1 |
| 1 | 1 |
| 2 | 2 |
| 3 | 3 |
| 4 | 5 |
| 5 | 8 |
| … | … |
| n | fib(n-1) + fib(n-2) |

On peut facilement implanter un algorithme récursif qui calcule la suite de Fibonnaci. Voici un exemple d’algorithme :

Fib (N)

 SI (N<=1) ALORS

 RETOURNE 1

 SINON

 RETOURNE Fib(N-1) + Fib(N-2)

Il suffit en effet de décrire en code récursif une définition qui est récursif. Implantons le donc en C avec quelques impressions pour visualiser l’exécution :

#include <stdio.h>

int Fib (int N);

int MAAAAL = 0;

int main ()

{

 int Nombre;

 printf ("Quel est votre nombre?");

 scanf ("%i", &Nombre);

 int Fibonnaci = Fib (Nombre);

 printf ("La reponse est : %i\n", Fibonnaci);

 printf ("Il a fallu %i iteration.\n",MAAAAL);

 return 0;

}

int Fib (int N)

{

 MAAAAL++;

 printf ("Appel de Fib pour %i\n",N);

 if (N <= 1)

 return 1;

 else

 return Fib(N-1) + Fib(N-2);

}

Résultat d’un exécution :

Quel est votre nombre?4

Appel de Fib pour 4

Appel de Fib pour 3

Appel de Fib pour 2

Appel de Fib pour 1

Appel de Fib pour 0

Appel de Fib pour 1

Appel de Fib pour 2

Appel de Fib pour 1

Appel de Fib pour 0

La reponse est : 5

Il a fallu 9 iteration.

On voit que dans ce cas, l’algorithme récursif n’est pas très performant. Par contre l’algorithme non récursif serait beaucoup moins évident. En voici un qui fonctionne, implanté en C :

#include <stdio.h>

int Fib (int N);

int main ()

{

 int Nombre;

 printf ("Quel est votre nombre?");

 scanf ("%i", &Nombre);

 int Fibonnaci = Fib (Nombre);

 printf ("La reponse est : %i\n", Fibonnaci);

 return 0;

}

int Fib (int N)

{

 int DerniereValeur =1;

 int Total = 1;

 for (int NombreRendu = 1; NombreRendu < N; NombreRendu++)

 {

 int Tamp = Total;

 Total = Total + DerniereValeur;

 DerniereValeur = Tamp;

 }

 return Total;

}

Dans ce cas, l’algorithme récursif est très facile mais moins rapide. Certains problèmes sont cependant très difficiles à régler de façon non-récursive. La solution récusive est souvent facile à comprendre (quoique difficile à trouver).

La tour d’Hanoi (voir <http://www.cut-the-knot.com/recurrence/hanoi.html> et p 233-234)

 La tour d’Hanoi, aussi appelée la tour de Brahma ou le casse-tête de la fin du monde, a été inventé par le mathématicien français Edouard Lucas en 1883. Il a été inspiré par la légende d’un temple Hindous ou la casse-tête de la pyramide aurait servi au jeune prêtre pour aiguiser leur discipline mentale. La légende raconte qu’au début des temps, les prêtres du temple ont reçu une pile 64 disques, chacun un peu plus petit que celui en dessous. Leur tâche consistait à déplacer les 64 disques d’un pôle à un autre, en ne mettant jamais un disque plus gros sur un plus petit. Les prêtres travaillaient jour et nuit très efficacement. Lorsqu’ils finiraient, le mythe dit que le temple tomberait en cendre et que le monde disparaîtrait.[[1]](#footnote-1)

Sacré prêtre : )Voici un dessin pour comprendre :

Pour le détail des explications de la solution poursuivie, livre p234-235.

Voici donc un algorithme qui implante cette solution en C :

#include <stdio.h>

void Hanoi (int Depart, int Fin, int NombreDisque);

int main ()

{

 Hanoi (1,3,3);

 return 0;

}

void Hanoi (int Depart, int Fin, int NombreDisque)

{

 if (NombreDisque ==1)

 {

 printf ("On deplace de %i a %i\n", Depart, Fin);

 return;

 }

 Hanoi (Depart, 6 - Depart - Fin, NombreDisque-1);

 Hanoi (Depart, Fin, 1);

 Hanoi (6 - Depart - Fin, Fin, NombreDisque-1);

}

 **N’ayez pas peur** de la récursion, c’est une méthode de programmation lisible, claire et modifiable. Plusieurs jeunes programmeur trouve ces solutions difficiles, mais une fois que l’on a allumé, c’est un charme d’utilisation.

Exercice :

 p 234-235 Exercices #

3.40 (facile),

3.41 (solution plus haut),

3.42 (solution plus haut… inspirez vous sans copier,

3.43 (dur),

3.44 (bien comprendre,

3.45 (intéressant),

3.46 (répondre à la question et comprendre le problème).

Laboratoire à venir…

1. Traduction libre de <http://www.lhs.berkeley.edu/Java/Tower/towerhistory.html> [↑](#footnote-ref-1)