Sommaire

[Introduction 2](#_Toc202975082)

[A l’assaut du bastion récursif ! 3](#_Toc202975083)

[PARTIE 1 : QUAND L’UTILISER 3](#_Toc202975084)

[Exemple 1 : Une calculatrice « en ligne de commande » 3](#_Toc202975085)

[Exemple 2 : Résolution d’une grille de Sudoku 4](#_Toc202975086)

[PARTIE 2: QUAND NE PAS L’UTILISER 5](#_Toc202975087)

[Exemple 1: La suite de Fibonacci 6](#_Toc202975088)

[Exemple 2 : Décomposition en produits de facteurs premiers 7](#_Toc202975089)

[Exemple 3 : Le pathfinding 8](#_Toc202975090)

[Conclusion 10](#_Toc202975091)

La récursivité : quand l’utiliser et quand ne pas l’utiliser

# Introduction

Vous avez découvert avec le tutorial de ??? la récursivité ?

Cette méthode bien utilisée peut diviser la taille de votre code, faciliter sa compréhension …mais elle possède aussi quelques revers qui ne doivent pas être négligés.

Alors avant de foncer tête baissée, pensez à tout : la méthode récursive est-elle vraiment la plus adaptée ? C’est ce que nous allons essayer de découvrir à travers quelques études de cas. Les exemples seront données dans un langage proche de l’algorithmique, le BASIC. Ce langage offre très peu de fonctions, et vous pourrez donc convertir sans aucun problème n’importe quel code dans un langage plus évolué que vous maitrisez mieux.

Les codes seront fournis pour chaque exemple, et un fichier exe compilé. Malheureusement, les amateurs de Linux ne pourront pas tester ces codes…ils devront se rabattre sur la capture d’écran !

PETIT RAPPEL : Avant de commencer, un petit rappel de vocabulaire. Un algorithme est dit itératif lorsqu’il utilise des boucles (for…next, do…loop).

Il est dit récursif lorsqu’il utilise des appels d’une fonction vers la même fonction en réduisant la complexité à chaque appel.

# A l’assaut du bastion récursif !

## PARTIE 1 : QUAND L’UTILISER

Avant de diaboliser la récursivité, voyons quelques exemples ou elle peut s’avérer bien pratique.

Quelques critères qui déterminent une utilisation possible et justifiée :

* La complexité du problème se réduit bien en utilisant la récursivité
* La méthode itérative ne peut pas se mettre en place, est trop lourde ou peu adaptée.
* Plusieurs solutions à votre problème sont envisageables.
* La complexité du problème est imprévisible, et pas forcément linéaire (explorer des tableaux contenant des tableaux, par exemple)

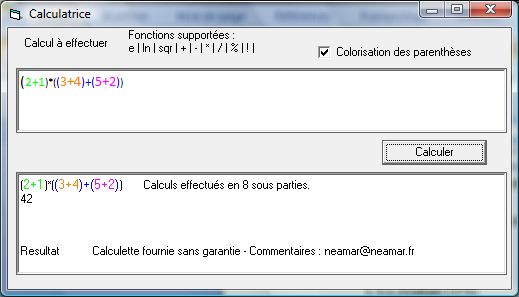
### Exemple 1 : Une calculatrice « en ligne de commande »

Imaginons que l’on souhaite réaliser une calculette évoluée, prenant en compte priorité des opérateurs, parenthèses…avec la méthode itérative, ceci est proprement infaisable : pour évaluer l’expression (2+1)\*[(3+4)+(5+2)], seule la méthode récursive se révèle fiable : essayez donc de réaliser ce calcul en itératif…vous m’en donnerez des nouvelles.

En revanche, avec la récursivité, le problème se décompose en sous-problèmes beaucoup plus simples :

1. Fonction Calculer : reçoit en paramètre (2+1)\*[(3+4)+(5+2)]
   1. Evaluer (2+1)
   2. En déduire (2+1) = 3
   3. Evaluer [(3+4)+(5+2)]
      1. Evaluer (3+4)
      2. En déduire (3+4)=7
      3. Evaluer (5+2)
      4. En déduire (5+2)=7
   4. En déduire [(3+4)+(5+2)] = 14
2. En déduire (2+1)\*[(3+4)+(5+2)] = 42

Comme on le voit, il suffit maintenant d’écrire un code gérant la décomposition du calcul selon les parenthèses (facile), puis un sous module utilisant la priorité des opérations (légèrement plus complexe, mais réalisable). Un problème insoluble à première vue s’avère ainsi « facile » à résoudre.



Dans ce cas, l’usage de la récursivité s’avère donc plus que justifiée, mais ce n’est pas le cas tout le temps…

LIEN

Voyons une autre étude de cas.

### Exemple 2 : Résolution d’une grille de Sudoku

Pour ceux ayant vécu dans des grottes troglodytes ces trois dernières années, voilà un petit rappel des règles du Sudoku, jeu qui vient de connaitre un succès planétaire. (cf. Wikipedia)

La résolution d’une grille de Sudoku standard peut se concevoir de plusieurs façons, et la méthode itérative fait partie des solutions possibles.

Mais imaginons maintenant que l’on souhaite résoudre une grille qui peut posséder plus d’une solution. Ce n’est pas le cas des grilles de jeux données dans les magazines, mais cela peut être l’objet d’un petit défi mathématique.

Cette fois, la méthode itérative s’avère beaucoup plus lourde, voire ingérable : il faudrait utiliser une méthode de Brute-Force, et les résultats risqueraient de prendre beaucoup de temps pour sortir…examinons donc la méthode récursive, qui se révélera un petit peu plus fine, et qui ne demandera pas d’acheter un ordinateur Deep Blue 4 ☺ !

Prenons le raisonnement suivant :

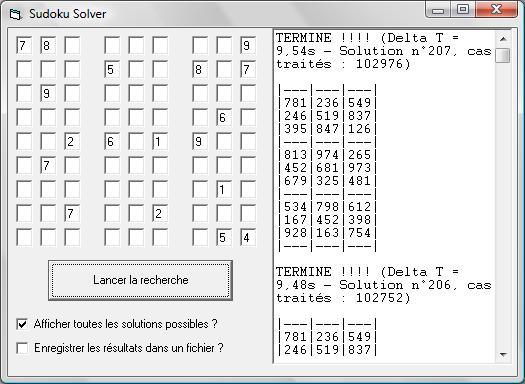
1. Fonction Sudoku\_Solver : reçoit une grille de sudoku
2. Si la grille est pleine, l’enregistrer,
3. Sinon,
   1. Trouver UNE case vide
   2. Tester pour tous les chiffres de 1 à 9
      1. Si le chiffre testé est envisageable pour la case vide choisie, placer le chiffre sur la grille à cet emplacement. Envoyer la grille à la fonction Sudoku\_Solver
   3. Si aucun chiffre n’est envisageable pour cette position, c’est que la grille est incorrecte. Ce n’est pas grave : cette branche de la récursivité s’arrêtera ici.
4. Arrêter la fonction

A chaque itération, la grille se remplit d’une case. (Comme on teste pour tous les nombres de 1 à 9, on est sur qu’au moins une des grilles contient la bonne réponse…si la bonne réponse existe)

On peut voir cet algorithme de résolution comme un arbre : à chaque passage, on rajoute quelques branches. Si aucune solution n’est possible, la grille est fausse : la branche n’engendre pas d’enfants, et meurt donc.

On peut remarquer que si on envoie à la fonction Sudoku\_Solver une grille complètement vide, on peut en théorie obtenir toutes les grilles de Sudoku du monde. Cependant, au vu du nombre de possibilités ( ???) et des défauts de l’algorithme listés ci-dessous, il faudrait un ordinateur disposant d’une énorme quantité de mémoire vive.

A première vue, cet algorithme parait optimal…mais ne nous emballons pas : le nombre de possibilités testées est impressionnant et loin d’être optimal. Pour ne rien faciliter, le type de retour de la fonction est Array (tableau) : a chaque appel on envoie plusieurs données (on doit utiliser un passage par valeur et non par référence : sinon on agit sur la référence mémoire du parent, donc sur la grille du parent, ce qui empêcherait les enfants de fonctionner correctement).



Dans ce cas là, l’avantage principal est que l’on peut recevoir autant de solutions que l’on veut. Dans le cas d’une grille standard à solution unique, cette méthode perd beaucoup d’intérêt par rapport à une résolution classique itérative.

[Lien]

## PARTIE 2: QUAND NE PAS L’UTILISER

Dans la majorité des cas, dire qu’une fonction récursive n’est pas optimale n’est pas faisable au premier coup d’œil : cela demande de réfléchir à la façon dont on pourrait réaliser la solution en itératif, si la fonction récursive est bien employée…car il est facile de tomber dans l’excès ! En croyant faire le bien avec la méthode récursive, on peut obtenir bien pire…Quelque cas :

* La méthode récursive implique une redondance des calculs (cf. Exemple 1)
* Le langage utilisé ne permet pas l’emploi de fonctions tail-rec
* La méthode récursive s’avère plus complexe que la méthode itérative (même si la version récursive peut être beaucoup plus courte en lignes de code, il ne faut pas perdre de vue que ce qui importe au final c’est la version compilée, et il faut qu’elle soit optimale)
* On n’a pas forcément besoin d’explorer toutes les possibilités (ce que l’on cherche n’est pas forcément situé sur une des branches finales, mais peut se situer sur une branche secondaire, ou même directement sur le tronc)

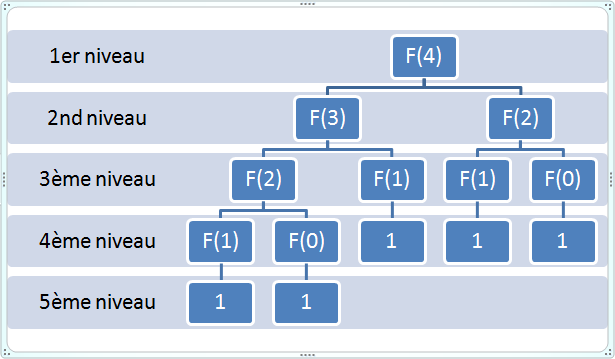
### Exemple 1: La suite de Fibonacci

Nous venons de voir deux exemples ou l’utilisation de la récursivité était acceptable, à défaut d’être optimale. Mais ce n’est pas toujours le cas ! Je souhaite commencer ce réquisitoire par quelque chose de très simple.

Je citerais ainsi sans m’y attarder la suite de Fibonacci, souvent donnée en exemple de récursivité sous cette forme :

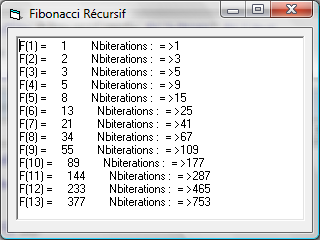
1. Fonction Fibonacci : Recevoir un nombre n
   1. Si n<2, renvoyer 1 car <math>U\_1=U\_0=1</math>
   2. Sinon, renvoyer Fibonacci(n-1) + Fibonacci(n-2)

Regardons rapidement l’enchainement des calculs : pour Fibonacci(4) on obtient :



Que remarque-t-on ? Que F(2) est calculé plusieurs fois. Dans ce cas, cela reste minime. Mais quand n augmente, chaque nombre sera calculé plusieurs fois. Ainsi, là ou une solution itérative impliquerait n-1 calculs, la solution récursive en demande beaucoup plus (une étude plus approfondie montre que le nombre de calculs effectués avec la méthode récursive est très liée à la suite de Fibonacci...et cela n’a aucun rapport avec la suite ! mais cela rentre dans le cadre de mathématiques légèrement plus poussées, nous n’en parlerons donc pas).

Certes, on peut optimiser la méthode récursive en utilisant des tableaux, mais cela devient au final plus lourd que la méthode itérative.



### Exemple 2 : Décomposition en produits de facteurs premiers

[XKCD]

Nous allons maintenant nous pencher sur un nouvel exemple : la décomposition en facteurs premiers. Il s’agit d’un théorème mathématique : tout entier peut être décomposé en un produit de facteurs premier.

Quelques exemples :

* 12 = 2²\*3
* 54 = 2 \* 3 \* 3 \* 3
* 53 = 53 (car 53 est premier !)

Nous allons maintenant tenter de réaliser un algorithme qui réalisera cette tache fastidieuse pour nous. (On admet que cette décomposition est unique)

Rappel : Pour trouver la décomposition d’un nombre, il faut et il suffit de tester les nombres inférieurs à la racine du nombre. (Cela se démontre facilement en observant les propriétés du produit et du carré)

Voyons le résultat, codé de façon naïve

1. Fonction DecoFact : reçoit un nombre K
2. Calculer sa racine
3. Pour tous les nombres inférieurs à cette racine (appelons les « i »),
   1. Si i est premier
      1. Si le nombre testé est divisible par i (i.e K % i = 0 ou encore K/i = E(K/i) avec % le modulo, et E la fonction partie entière)
         1. Trouver le complément j tel que i\*j=k
         2. Ajouter i à la liste des diviseurs à retourner
         3. Envoyer j à la fonction DecoFact, récupérer le résultat et l’ajouter au tableau préexistant
4. Fin de la fonction : renvoyer la liste crée

Comme on le voit, cet algorithme possède de nombreuses contraintes :

* Il faut réaliser une fonction qui teste si le nombre est premier, ce qui obligera encore à faire une boucle (on ne sait pas, dans le cas général, dire si un nombre est premier sans tester exhaustivement tous les nombres inférieurs à sa racine).
* Il utilise la récursivité : ce n’est pas forcément un défaut, mais dans ce cas là c’est inutile et lourd à mettre en place. N’oubliez pas que le processeur doit retenir la pile d’appel, ce qui alourdit inutilement les opérations.

Réfléchissons ensemble : comment améliorer cet algorithme ? Je vous laisse prendre une longueur d’avance…

Vous avez trouvés ? Félicitations, dans ce cas là vous n’avez plus besoin de moi…ah si, le monsieur qui lève le doigt au fond ! Vous voulez plus d’explications ? Bon très bien.

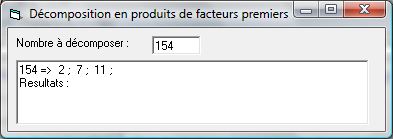
On peut éviter le recours à la récursivité : que diriez-vous de l’algorithme suivant :

1. Fonction DecoFact : reçoit un nombre K
2. Calculer sa racine
3. Pour tous les nombres inférieurs à cette racine (appelons les « i »),
   1. Si le nombre testé est divisible par i
      1. L’ajouter dans la liste des diviseurs.
      2. Faire K = K / i
      3. Faire i = i -1
   2. Si K>2, continuer la boucle, sinon en sortir
4. Si K>2, le nombre restant est premier : l’ajouter à la liste
5. Fin de la fonction : Renvoyer la liste.

On obtient strictement le même résultat qu’avec l’algorithme précédent. Seule différence : on ne fait plus le test de primalité, on n’a plus besoin de la récursivité, et le nombre d’itérations dans les boucles est réduit au strict minimum.

Attardons nous d’abord sur le test de primalité. Il est complètement inutile dans ce nouveau programme, puisque si i divise K, alors i est forcément premier puisque K est ajusté en permanence. Nous venons de gagner plusieurs cycles machines ! (faites des tests, vous verrez !)

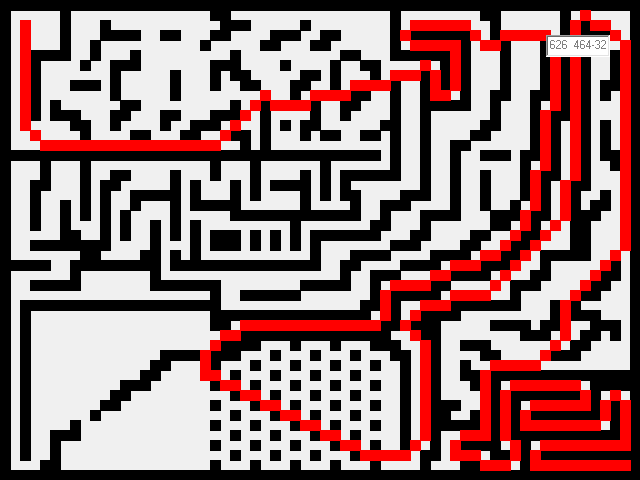
Quant à la récursivité, une observation attentive du premier algorithme nous montre qu’il est absolument inutile de refaire les tests pour les numéros inférieurs à i : si K n’est pas divisible par z, alors K/i ne sera pas non plus divisible par z ! En fait, l’appel récursif n’a pas complètement disparu : on a juste profité du fait que les tests pouvaient se « continuer » à l’intérieur de la fonction, sans avoir besoin de remettre en place la fonction (le fait que la fonction du 1er algorithme « s’arrête» une fois trouvé un nombre aide bien à le simplifier).



On l’a vu, cette fois, la récursivité était loin d’être adaptée ! Ne foncez pas dedans tête baissée à tout moment, cela peut amener à des codes beaucoup plus complexes.

### Exemple 3 : Le pathfinding

Cette dernière section couvrira rapidement le pathfinding, et plus particulièrement l’algorithme A\*. Elle sera légèrement plus complexe que les précédentes : je considérerais que vous connaissez le fonctionnement d’un pathfinder standard, qu’il soit A\* ou Djikstra. Sinon, vous pourrez découvrir ces algorithmes ici : [Lien]



L’algorithme A\* fonctionne sur le principe des listes fermées et ouvertes. L’une des opérations les plus couteuses en temps est la maintenance de liste ouverte triée selon la distance F, somme de la distance parcourue et de la distance heuristique.

Imaginons que l’on souhaite « améliorer » cet algorithme en y ajoutant de la récursivité.

On pourrait imaginer deux façons :

* 1ère idée : au lieu de maintenir une liste ouverte, on appelle une fonction qui prend en paramètres :
  + La carte du terrain, avec ses spécificités
  + Le carré Parent
  + Le cout F.

Dans une variable globale, on stockerait le cout F que l’on accepte.

Au tout début de la fonction, on ferait une boucle tant que le cout F global est inférieur au cout F de la fonction. Tant que l’on a cela, on laisserait la main aux autres événements, tandis qu’une fonction maitresse contrôlerait l’incrémentation de la variable F.

Dois-je vraiment donner tous les inconvénients de cette méthode ? Variables Globales, Multi threading, encombrement de la pile d’appel, tests inutiles et répétitifs, gestion de la fonction maitresse…on obtiendrait certes un pathfinder beaucoup plus propre, mais ô combien moins performant ! Ca tombe mal, c’est justement le but principal d’un pathfinder…

* 2ème idée : cette fois, focalisons nous sur l’algorithme de Djikstra, dont le but n’est pas de se rendre en un point fixe X,Y, mais en un point contenant un certain type d’item : par exemple, un villageois lancerait l’algorithme de Djikstra pour trouver la mine d’or la plus proche. On pourrait penser que la méthode récursive est une alternative possible. Malheureusement, la récursivité oblige à « fouiller » entièrement une branche avant de pouvoir passer à la suivante. C’est justement ce que l’algorithme de Djikstra essaie de ne pas faire : il faut fouiller le moins de nœuds possibles ! Si on utilisait la récursivité, une immense partie de la carte serait explorée à chaque recherche. Pour pallier à cet inconvénient, il faudrait mettre en place un paramètre qui indique à la fonction son niveau sur l’arbre (branche de niveau 1, de niveau 2…) et ne pas commencer un niveau tant que le précédant n’est pas fini. On retombe à nouveau sur un système de gestion de priorités en multithreading : infaisable pour un algorithme supposé être « real-time » !

Bref, le pathfinding est bien mieux tel qu’il est : des générations de programmeurs s’y sont cassés les dents, il était peu probable qu’une modification aussi « évidente » n’ait pas déjà été envisagée. Et si elle avait plus intéressante, cela se serait ! Comme le dit un célèbre proverbe geek : « pas d’intérêt à réinventer la roue »

# Conclusion

J’espère que vous avez compris le message : la récursivité peut s’avérer un outil précieux pour la simplification de vos codes.

Malheureusement, le verso des choses est moins brillant : il peut s’avérer que la méthode récursive soit plus complexe, plus lourde, et qu’au final elle rende l’algorithme incompréhensible pour un lecteur externe.

Ce n’est pas une raison pour oublier la récursivité : il ne faut simplement pas l’assaisonner à toutes les sauces. Tous comme le slogan « Manger Bouger », n’oubliez pas que l’arsenal du programmeur comporte d’autres outils, qui peuvent s’avérer plus pratiques et/ou plus adaptés que la récursivité.

Bonne continuation dans la programmation,

Neamar