Recursividad

introduction to recursion

Autor 1: Luigi Davian Patiño Cardona

*Universidad tecnológica de Pereira*

Correo-e: [luigi.patino@utp.edu.co](mailto:luigi.patino@utp.edu.co)

*Resumen*— La recursividad es una técnica de programación importante. Se utiliza para realizar una llamada a una [función](https://www.monografias.com/trabajos7/mafu/mafu.shtml) desde la misma función. Como ejemplo útil se puede presentar el [calculos](https://www.monografias.com/trabajos7/caes/caes.shtml) de números factoriales. Él factorial de 0 es, por definición, 1. Los factoriales de números mayores se calculan mediante la multiplicación de 1 \* 2 \* ..., incrementando el número de 1 en 1 hasta llegar al número para el que se está calculando el factorial.

<https://www.monografias.com/trabajos14/recursividad/recursividad.shtml#BIBLIO>

***Palabras clave—* Términos-Acerca del índice de cuatro palabras o frases clave en orden alfabético, separadas por comas. Para obtener una lista de palabras claves sugeridas, visite.**

**de Dios Murillo Morera, Juan, Caamaño Polini, Santiago, COMPARACIÓN ENTRE ALGORITMOS RECURSIVOS E ITERATIVOS Y SU MEDICIÓN EN TÉRMINOS DE EFICIENCIA. Uniciencia [en linea] 2013, 27 (Enero-Junio) : [Fecha de consulta: 1 de junio de 2019] Disponible en:**[**<http://www.redalyc.org/articulo.oa?id=475947762020>**](http://www.redalyc.org/articulo.oa)**ISSN**

***Abstract*— Recursion is an important programming technique.**

**is used to make a call to a function from the same function. As a useful example, the calculations of factorial numbers can be presented. The factorial of 0 is, by definition, 1. The factorials of major numbers are calculated by multiplying 1 \* 2 \* ..., increasing the number of 1 in 1 until reaching the number for which the factorial is being calculated.**

**Keywords- Terms-About the index of four words or key phrases in alphabetical order, separated by commas. For a list of suggested keywords, visit..**

de Dios Murillo Morera, Juan, Caamaño Polini, Santiago, COMPARISON BETWEEN RECURSIVE AND ITERATIVE ALGORITHMS AND ITS MEASUREMENT IN TERMS OF EFFICIENCY. Uniciencia [online] 2013, 27 (January-June): [Date of consultation: June 1, 2019] Available at: <http://www.redalyc.org/articulo.oa?id=475947762020> ISSN

### INTRODUCCIÓN

El área de la [programación](https://www.monografias.com/Computacion/Programacion/) es muy amplia y con muchos detalles. Los programadores necesitan ser capaces de resolver todos los [problemas](https://www.monografias.com/trabajos15/calidad-serv/calidad-serv.shtml#PLANT) que se les presente a través del [computador](https://www.monografias.com/trabajos15/computadoras/computadoras.shtml) aun cuando en [el lenguaje](https://www.monografias.com/trabajos16/desarrollo-del-lenguaje/desarrollo-del-lenguaje.shtml) que utilizan no haya una manera directa de resolver los problemas.

En el [lenguaje](https://www.monografias.com/trabajos35/concepto-de-lenguaje/concepto-de-lenguaje.shtml) de programación C, así como en otros lenguajes de programación, se puede aplicar una técnica que se le dio el nombre de recursividad por su funcionalidad. Esta técnica es utilizada en la programación estructurada para resolver problemas que tengan que ver con el factorial de un número, o [juegos](https://www.monografias.com/trabajos15/metodos-creativos/metodos-creativos.shtml) de [lógica](https://www.monografias.com/trabajos15/logica-metodologia/logica-metodologia.shtml).

Las asignaciones de [memoria](https://www.monografias.com/trabajos13/memor/memor.shtml) pueden ser dinámicas o estáticas y hay diferencias entre estas dos y se pueden aplicar las dos en un [programa](https://www.monografias.com/Computacion/Programacion/) cualquiera.

### CONTENIDO

**1)** **Propiedades de las definiciones o**[**algoritmos**](https://www.monografias.com/trabajos15/algoritmos/algoritmos.shtml)**recursivos:**

Un requisito importante para que sea correcto un algoritmo recursivo es que no genere una secuencia infinita de llamadas así mismo. Claro que cualquier algoritmo que genere tal secuencia no termina nunca. Una función recursiva *f* debe definirse en términos que no impliquen a *f* al menos en un argumento o [grupo](https://www.monografias.com/trabajos14/dinamica-grupos/dinamica-grupos.shtml) de argumentos. Debe existir una "salida" de la secuencia de llamadas recursivas.

Si en esta salida no puede calcularse ninguna función recursiva. Cualquier caso de definición recursiva o invocación de un algoritmo recursivo tiene que reducirse a la larga a alguna manipulación de uno o casos más simples no recursivos.

**2)  Cadenas recursivas:**

Una función recursiva no necesita llamarse a sí misma de manera directa. En su lugar, puede hacerlo de manera indirecta como en el siguiente ejemplo:

a (formales parámetros) b (formales parámetros)

{ {

. .

b (argumentos); a (argumentos);

. .

} /\*fin de a\*/} /\*fin de b\*/

En este ejemplo la función *a* llama a *b,* la cual puede a su vez llamar a *a*, que puede llamar de nuevo a *b.* Así, ambas [funciones](https://www.monografias.com/trabajos7/mafu/mafu.shtml) *a* y *b*, son recursivas, dado que se llamas a sí mismo de manera indirecta. Sin embargo, el que lo sean no es obvio a partir del examen del cuerpo de una de las rutinas en forma individual. La rutina *a*, parece llamar a otra rutina *b* y es imposible determinar que se puede llamar así misma de manera indirecta al examinar sólo a *a*.

Pueden incluirse más de dos rutinas en una cadena recursiva. Así, una rutina *a* puede llamar a *b*, que llama a *c*, ..., que llama a *z*, que llama a *a*. Cada rutina de la cadena puede potencialmente llamarse a sí misma y, por lo tanto, es recursiva. Por supuesto, el programador debe asegurarse de que un programa de este tipo no genere una secuencia infinita de llamadas recursivas.

**3)** **Definición recursiva de expresiones algebraicas:**

Como ejemplo de cadena recursiva consideremos el siguiente grupo de definiciones:

1. una **expresión** es un término seguido por un signo más seguido por un término, o un término solo
2. un **término** es un factor seguido por un asterisco seguido por un factor, o un factor solo.
3. Un **factor** es una letra o una expresión encerrada entre paréntesis.

Antes de ver algunos ejemplos, obsérvese que ninguno de los tres elementos anteriores está definido en forma directa en sus propios términos. Sin embargo, cada uno de ellos se define de manera indirecta. Una expresión se define por medio de un término, un término por medio de un factor y un factor por medio de una expresión. De manera similar, se define un factor por medio de una expresión, que se define por medio de un término que a su vez se define por medio de un factor. Así, el conjunto completo de definiciones forma una cadena recursiva.

La forma más simple de un factor es una letra. Así A, B, C, Q, Z y M son factores. También son términos, dado que un término puede ser un factor solo. También son expresiones dado que una expresión puede ser un término solo. Como A es una expresión, (A) es un factor y, por lo tanto, un término y una expresión. A + B es un ejemplo de una expresión que no es ni un término ni un factor, sin embargo (A + B) es las tres cosas. A \* B es un término y, en consecuencia, una expresión, pero no es un factor. A \* B + C es una expresión, pero no es un factor.

Cada uno de los ejemplos anteriores es una expresión valida. Esto puede mostrarse al aplicar la definición de una expresión de cada uno. Considérese, sin embargo, la cadena A + \* B. No es ni una expresión, ni un término, ni un factor. Sería instructivo para el lector intentar aplicar la definición de expresión, término y factor para ver que ninguna de ellas describe a la cadena A + \* B. De manera similar, (A + B\*) C y A + B + C son expresiones nulas de acuerdo con las definiciones precedentes.

A continuación, se codificará un programa que lea e imprima una cadena de caracteres y luego imprima "valida" si la expresión lo es y "no valida" de no serlo. Se usan tres funciones para reconocer expresiones, términos y factores, respectivamente. Primero, sin embrago se presenta una función auxiliar *getsymb* que opera con tres parámetros: *str, length* y *ppos*. *Str* contiene la entrada de la cadena de cadena de caracteres; *length* representa el número de caracteres en *str*. *Ppos* apunta a un puntero *pos* cuyo valor es la posición *str* de la que obtuvimos un [carácter](https://www.monografias.com/trabajos34/el-caracter/el-caracter.shtml) la última vez. Si *pos* < *length*, *getsymb*regresa el carácter *cadena str [pos]* e incrementa *pos* en 1. Si *pos* > = *length*, *getsymb* regresa un espacio en blanco.

getsymb (str, length, ppos)

char str[];

int length, \*ppos;

{

char C;

if (\*ppos < length)

c = str [\*ppos];

else

c = ‘ ‘ ;

(\*ppos) ++;

return ( c );

} /\* fin de getsymb\*/

La función que reconoce una expresión se llama *expr*. Regresa *TRUE* (o1) (VERDADERO) si una expresión valida comienza en la posición *pos* de *str* y FALSE (o0) FALSO en caso contrario. También vuelve a colocar *pos* en la posición que sigue en la expresión de mayor longitud que puede encontrar. Suponemos también una función *readstr* que lee una cadena de caracteres, poniendo la cadena en *str* y su largo en *length*.

Una vez descritas las funciones *expr* y *readst,* puede escribirse la rutina principal como sigue. La [biblioteca](https://www.monografias.com/trabajos10/ponency/ponency.shtml) estándar *ctype.h* incluye una función *isalpha*que es llamada por una de las funciones siguientes.

# include <stdio.h>

# include <ctype.h>

# define TRUE 1

# define FALSE =

# define MAXSTRINGSIZE 100

main ()

{

char str [MAXSTRINGSIZE];

int length, pos;

readstr (str, &length);

pos = 0;

if (expr (str, length, &pos) = = TRUE && por >= length)

printf ("%s", "valida");

else

printf ("%s", "no valida");

/\* la condición puede fallar por una de dos razones (o ambas). Si expr(str, length, &pos) = = FALSE entonces no hay una expresión valida al inicio de pos. Si pos < length puede que se encuentre una expresión valida, comenzando en pos, pero no ocupa la cadena completa \*/

} /\*fin del main\*/

Las funciones *factor* y *term* se parecen mucho a *expr* excepto en que son responsables del reconocimiento de factores y término, respectivamente. También reinicializan *pos* en la posición que sigue al factor o término de mayor longitud que se encuentra en la cadena *str*.

Los códigos para estas rutinas se apegan bastantes a las definiciones dadas antes. Cada una intenta satisfacer uno de los criterios para la entidad que se reconoce. Si se satisface uno de esos criterios el resultado es TRUE (VERDADERO). Si no satisface ninguno, el resultado es FALSE (FALSO).

expr (str. length, ppos)

char str [];

int length, \*ppos;

{

/\* buscando un término \*/

if (term( str, length, ppos) = = FLASE)

return (FLASE);

/\* se ha encontrado un término; revisar el siguiente símbolo \*/

if (getsymb(str, length, ppos) ! = ‘+’) {

/\* se encontró la mayor expresión (un solo término). Reposicionar pos para que señale la última posición de la expresión \*/

(\*ppos) - - ;

return (TRUE);

} /\* fin del if \*/

/\* en este punto, se a encontrado un termino y su signo mas. Se deberá buscar otro termino \*/

return (term(str, length, ppos));

} /\*fin de expr \*/

La rutina *term* que reconoce términos, es muy similar y será presentada sin comentarios.

term (str, length, ppos)

char str[];

int length, \*ppos;

{

if (factor(str, length, ppos) = = FALSE)

return (FALSE);

if (getsymb (str, length, ppos) ! = ‘+’) {

(\*ppos) -- ;

return (TRUE) ;

} /\* fin del if \*/

return (factor(str, length, ppos));

} /\* fin de term \*/

La función *factor* reconoce factores y debería ser ahora bastante sencilla. Usa el programa común de biblioteca *isalpha* (esta función se encuentra en la biblioteca *ctype.h*), que regresa al destino de cero si su carácter de parámetro es una letra y cero (o FALSO) en caso contrario.

factor (str, length, ppos)

char str[];

int length, \*ppos;

{

int c;

if ((c = getsymb (str, length, ppos)) ! = ‘)’ )

return (isalpha(c));

return (expr(str, length, ppos) && getsymb (str, length, ppos) == ‘)’ );

} /\* fin de factor \*/

Las tres rutinas son recursivas, dado que cada una puede llamar a sí misma da manera indirecta. Pos-ejemplo, si se sigue la acción del programa para la cadena de entrada " (a \* b + c \* d) + (e \* (f) + g) " se encontrará que cada una de las tres rutinas *expr*, *term* y *factor* se llama a sí misma.

**4) Programación Recursiva:**

Es mucho mas difícil desarrollar una solución recursiva en C para resolver un problema especifico cuando no se tiene un algoritmo. No es solo el programa sino las definiciones originales y los algoritmos los que deben desarrollarse. En general, cuando encaramos la tarea de escribir un programa para resolver un problema no hay razón para buscar una solución recursiva. La mayoría de los problemas pueden resolverse de una manera directa usando métodos no recursivos. Sin embargo, otros pueden resolverse de una manera mas lógica y elegante mediante la recursión.

**5)** **Asignación**[**estática**](https://www.monografias.com/trabajos5/estat/estat.shtml)**y**[**dinámica**](https://www.monografias.com/trabajos34/cinematica-dinamica/cinematica-dinamica.shtml)**de memoria:**

Hasta este momento solamente hemos realizado asignaciones estáticas del programa, y más concretamente estas asignaciones estáticas no eran otras que las declaraciones de [variables](https://www.monografias.com/trabajos12/guiainf/guiainf.shtml#HIPOTES) en nuestro programa. Cuando declaramos una variable se reserva [la memoria](https://www.monografias.com/trabajos16/memorias/memorias.shtml) suficiente para contener la [información](https://www.monografias.com/trabajos7/sisinf/sisinf.shtml) que debe almacenar. Esta memoria permanece asignada a la variable hasta que termine la ejecución del programa (función main). Realmente las variables locales de las funciones se crean cuando éstas son llamadas pero nosotros no tenemos [control](https://www.monografias.com/trabajos14/control/control.shtml) sobre esa memoria, el compilador genera el [código](https://www.monografias.com/trabajos12/eticaplic/eticaplic.shtml) para esta operación automáticamente.

En este sentido las variables locales están asociadas a asignaciones de memoria dinámicas, puesto que se crean y destruyen durante la ejecución del programa.

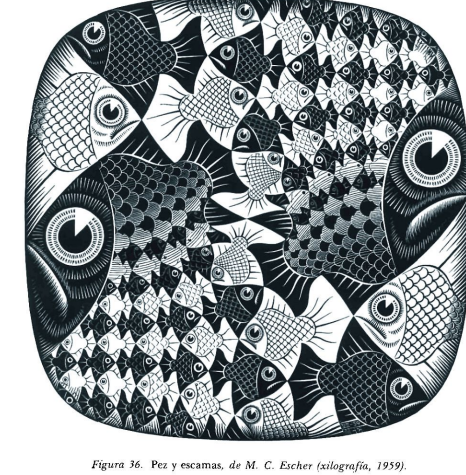
Así entendemos por asignaciones de memoria dinámica, aquellas que son creadas por nuestro programa mientras se están ejecutando y que por tanto, cuya [gestión](https://www.monografias.com/trabajos15/sistemas-control/sistemas-control.shtml) debe ser realizada por el programador.

El [lenguaje C](https://www.monografias.com/trabajos4/lenguajec/lenguajec.shtml) dispone, como ya indicamos con anterioridad, de una serie de librerías de funciones estándar. El fichero de cabeceras stdlib.h contiene las declaraciones de dos funciones que nos permiten reservar memoria, así como otra función que nos permite liberarla.

Las dos funciones que nos permiten reservar memoria son:

* malloc (cantidad\_de\_memoria);
* calloc (número\_de\_elementos, tamaño\_de\_cada\_elemento);

ejemplo 1. **Pez y escamas**

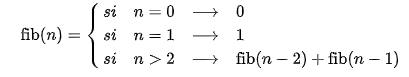


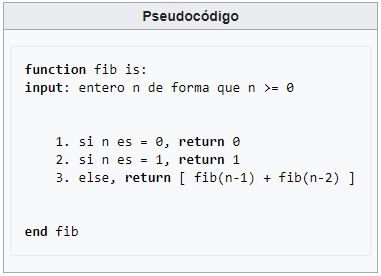
Escher adoptó la idea de hacer, de partes de un objeto, réplicas del objeto mismo, y la concretó en una obra: su grabado Pez y escamas (figura 36). Por supuesto, estos peces y escamas resultan similares sólo si son examinados en un plano lo suficientemente abstracto.

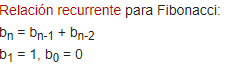
#### Ejemplo 2. Fibonacci

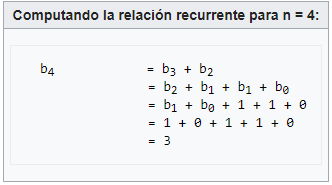
Otra popular secuencia recursiva es el [Número de Fibonacci](https://es.wikipedia.org/wiki/N%C3%BAmero_de_Fibonacci). Los primeros elementos de la secuencia son: 0, 1, 1, 2, 3, 5, 8, 13, 21...

Definición de la función









Este algoritmo de Fibonacci es especialmente malo pues cada vez que se ejecuta la función, realizará dos llamadas a la función a sí misma, cada una de las cuales hará a la vez dos llamadas más y así sucesivamente hasta que terminen en 0 o en 1. El ejemplo se denomina "recursión de árbol", y sus requisitos de tiempo crecen de forma exponencial y los de espacio de forma lineal.

1. CONCLUSIONES

### Se puede decir que la recursividad es una técnica de programación bastante útil y muy interesante de estudiar. A través de los ejemplos que el [individuo](https://www.monografias.com/trabajos28/aceptacion-individuo/aceptacion-individuo.shtml) pueda revisar, aprenderá con más rapidez y sencillez lo que es programar recursivamente e incluir esta técnica cuando se le presente un problema como los que fueron mencionados anteriormente. La asignación de memoria sea estática o dinámica, en realidad se tendrá que aplicar en cualquier programa al momento de su codificación; tomando en cuenta que cada programador tiene su estilo de programar. El ejemplo de las torres de Hanoi tanto como el ejemplo de las ocho reinas son problemas claves que tienen que ver directamente con lo que es la recursividad.

### RECOMENDACIONES

Esta sección sigue el formato regular del resto del documento. La única observación es notar que el título no está numerado.

En esta sección se agregan agradecimientos a personas que colaboraron en el proyecto pero que no figuran como autores del paper.

### REFERENCIAS

[1]. Tenenbaum, Aaron. [Estructura de Datos](https://www.monografias.com/trabajos10/esda/esda.shtml) en C. [México](https://www.monografias.com/trabajos/histomex/histomex.shtml) 1995.

**[2].**Páginas [Web](https://www.monografias.com/trabajos5/laweb/laweb.shtml) Visitadas: <http://www.ulpgc.es/otros/tutoriales/mtutor/ej-d.html>

**[3].**<http://www.uhu.es/nieves.pavon/p2/tema5/5punto11.html>

**[4].**<http://193.145.132.15/otros/tutoriales/mtutor/mt2f.html>

**[5].**<http://microsoft.com/spain/scripting/jscript/doc/recurse.htm>

**[6].**<http://avata.utadeo.edu.co/Lecturas/Hofstadter_Douglas_Un_Eterno_y_Gracil_Bucle.pdf>

**Enlaces externos**

[Harold Abelson and Gerald Sussman: "Structure and Interpretation Of Computer Programs"](http://mitpress.mit.edu/sicp/full-text/book/book.html)

[IBM DeveloperWorks: "Mastering Recursive Programming"](https://developer.ibm.com/articles/l-recurs/)

[David S. Touretzky: "Common Lisp: A Gentle Introduction to Symbolic Computation"](http://www.cs.cmu.edu/~dst/LispBook/)

[Matthias Felleisen: "How To Design Programs: An Introduction to Computing and Programming"](http://www.htdp.org/2003-09-26/Book/)

**Observaciones generales:**

Este documento de ejemplo, en Microsoft Word, para la elaboración de artículos para la revista SCIENTIA ET TECHNICA podrá ser descargado de la página:

<http://revistas.utp.edu.co/index.php/revistaciencia/pages/view/formatos>.

Haciendo clic en la pestaña *Formatos*.

**Presentación de trabajos:**

Los artículos deben venir acompañados por los formatos de datos del autor, el cual se puede descargar en la página *web* de la revista <http://revistas.utp.edu.co/index.php/revistaciencia/pages/view/formatos>. haciendo clic en la pestaña *Formatos*. Estos formatos deben ser cargados en la plataforma Open Journal Systems. Los datos allí consignados serán incorporados en la Base Bibliográfica *Publindex* de Colciencias.

Los artículos deben estar presentados en el formato de la revista, el cual se puede descargar en la página *web* de la revista <http://revistas.utp.edu.co/index.php/revistaciencia/pages/view/formatos> haciendo clic en la pestaña *Formatos*. El no uso de este formato descalifica el artículo y no será tenido en cuenta en la convocatoria.

**Envío de artículos**

La recepción de artículos se realizará por medio de Open Journal Systems - OJS en las fechas en que están abiertas las convocatorias.