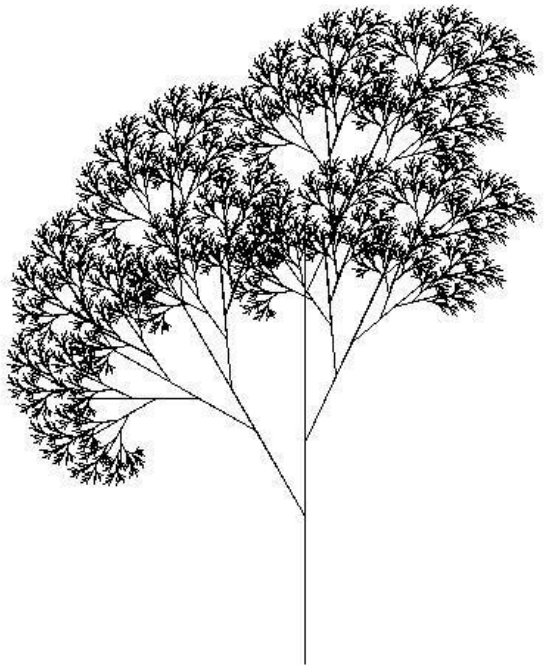
Recursión (ciencias de computación)

Recursión es, en ciencias de computación, una forma de atajar y solventar problemas. De hecho, recursión es una de las ideas centrales de ciencia de computación.[1] Resolver un problema mediante recursión significa que la solución depende de las soluciones de pequeñas instancias del mismo problema.[2]

> El poder de la recursión evidentemente se fundamenta en la posibilidad de definir un conjunto infinito de objetos con una declaración finita. Igualmente, un número infinito de operaciones computacionales puede describirse con un programa recursivo finito, incluso en el caso de que este programa no contenga repeticiones explícitas."[3]

La mayoría de los lenguajes de programación dan soporte a la recursión permitiendo a una función llamarse a sí misma desde el texto del programa. Los lenguajes imperativos definen las estructuras de *loops* como while y for que son usadas para realizar tareas repetitivas. Algunos lenguajes de programación funcionales no definen estructuras de *loops* sino que posibilitan la recursión llamando código de forma repetitiva. La teoría de la computabilidad ha demostrado que estos dos tipos de lenguajes son matemáticamente equivalentes, es decir que pueden resolver los mismos tipos de problemas, aunque los lenguajes funcionales carezcan de las típicas estructuras while y for.



Árbol basado en la recursión creado usando el lenguaje de programación Logo.

1 Algoritmos recursivos

Un método frecuente para simplificar es dividir un problema en problemas más derivados de menor tamaño del mismo tipo. Esto se conoce como *dialecting*. Cómo técnica de programación se denomina divide y vencerás y es pieza fundamental para el diseño de muchos algoritmos de importancia, así como parte esencial de la programación dinámica.

Virtualmente todos los lenguajes de programación modernos permiten la especificación directa de funciones y subrutinas recursivas. Cuando se llama una función de este tipo, el ordenador, para la mayoría de los lenguajes en casi todas las arquitecturas basadas en una pila (*stack*) o en la implementación del lenguaje, lleva la cuenta de las distintas instancias de la función, en numerosas arquitecturas mediante el uso de un *call stack*, aunque no de forma exclusiva. A la inversa, toda función recursiva puede transformarse en una función iterativa usando un *stack*.

La mayoría (aunque no todas) de las funciones y subrutinas que pueden ser evaluadas por un ordenador, pueden expresarse en términos de una función recursiva (sin tener que utilizar una iteración pura); a la inversa, cualquier función recursiva puede expresarse en términos de una iteración pura, dado que la recursión es, de por sí, también iterativa. Para evaluar una función por medio de la recursión, tiene que definirse como una función de si misma (ej. el factor n! = n * (n - 1)! , donde 0! se define como 1). Resulta evidente que no todas las evaluaciones de funciones se prestan a un acercamiento recursivo. Por lo general, todas las funciones finitas pueden describirse directamente de forma recursiva; las funciones infinitas (ej. las series de e = 1/1! + 2/2! + 3/3!...) necesitan un criterio extra para detenerse, ej. el número de iteraciones, o el número de dígitos significativos, en caso contrario una iteración recursiva resultaría en un bucle infinito.

A modo de ilustración: Si se encuentra una palabra desconocida en un libro, el lector puede anotar la página actual en un papel y ponerlo en una pila (hasta entonces vacía).

El lector consulta la palabra en otro artículo y, de nuevo, descubre otra palabra desconocida, la anota y la pone en la pila, y así sucesivamente. Llega un momento que el lector lee un artículo que donde todas las palabras son conocidas. El lector retorna entonces a la última página y continua la lectura desde ahí, y así hasta que se retira la última nota de la pila retornando entonces al libro original. Este *modus operandi* es recursivo.

Algunos lenguajes diseñados para programación lógica y programación funcional ofrecen la recursión como el único medio de repetición directa disponible para el programador. Estos lenguajes suelen conseguir que la recursión de cola sea tan eficiente como la iteración, permitiendo a los programadores expresar otras estructuras repetitivas (tales como map y for de scheme) en términos de recursión.

La recursión está profundamente anclada en la teoría de computación, con la equivalencia teórica de función microrecursiva y máquinas de Turing en la cimentación de ideas sobre la universalidad del ordenador moderno.

2 Programación recursiva

Crear una subrutina recursiva requiere principalmente la definición de un "caso base", y entonces definir reglas para subdividir casos más complejos en el caso base. Para una subrutina recursiva es esencial que con cada llamada recursiva, el problema se reduzca de forma que al final llegue al caso base.

Algunos expertos clasifican la recursión como "generativa" o bien "estructural". La distinción se hace según de donde provengan los datos con los que trabaja la subrutina. Si los datos proceden de una estructura de datos similar a una lista, entonces la subrutina es "estructuralmente recursiva"; en caso contrario, es "generativamente recursiva".[4]

> Muchos algoritmos populares generan una nueva cantidad de datos a partir de los datos aportados y recurren a partir de ahí. HTDP (*How To Design Programs*), al español, "Cómo diseñar programas", se refiere a esta variante como recursión generativa. Ejemplos de recursión generativa incluyen: máximo común divisor, quicksort, búsqueda binaria, mergesort, Método de Newton, fractals e integración adaptiva.[5]

2.1 Ejemplos de subrutinas definidas recursivamente (recursión generativa)

2.1.1 Factorial

Un ejemplo clásico de una subrutina recursiva es la función usada para calcular el factorial de un entero.

Definición de la función:

$$\text{fact}(n) = \begin{cases} si & n = 0 & \longrightarrow & 1 \\ si & n > 0 & \longrightarrow & n \cdot \text{fact}(n-1) \end{cases}$$

Una relación recurrente es una ecuación que relaciona términos posteriores en la secuencia con términos previos.[6]

Relación recurrente de un factorial:

$$b_n = nb_{n-1}$$

$$b_0 = 1$$

Esta función factorial también puede describirse sin usar recursión haciendo uso de típicas estructuras de bucle que se encuentran en lenguajes de programación imperativos:

El lenguaje de programación scheme es, sin embargo, un lenguaje de programación funcional y no define estructuras de *loops* de cualquier tipo. Se basa únicamente en la recursión para ejecutar todo tipo de *loops*. Dado que scheme es recursivo de cola, se puede definir una subrutina recursiva que implementa la subrutina factorial como un proceso iterativo, es decir, usa espacio constante pero tiempo lineal.

2.1.2 Fibonacci

Otra popular secuencia recursiva es el Número de Fibonacci. Los primeros elementos de la secuencia son: 0, 1, 1, 2, 3, 5, 8, 13, 21...

Definición de la función:

$$\text{fib}(n) = \begin{cases} si & n = 0 & \longrightarrow & 0 \\ si & n = 1 & \longrightarrow & 1 \\ si & n > 2 & \longrightarrow & \text{fib}(n-2) + \text{fib}(n-1) \end{cases}$$

Relación recurrente para Fibonacci:
$b_n = b_{n-1} + b_{n-2}$
$b_1 = 1, b_0 = 0$

Este algoritmo de Fibonacci es especialmente malo pues cada vez que se ejecuta la función, realizará dos llamadas a la función a si 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.[7]

2.1.3 Máximo común divisor

Otro famosa función recursiva es el algoritmo de Euclides, usado para computar el máximo común divisor de dos enteros.

Definición de la función:

$$\text{mcd}(x,y) = \begin{cases} si & y = 0 & \longrightarrow & x \\ si & y > 0 \ \wedge \ x \geq y & \longrightarrow & \text{mcd}(y, mod(x,y)) \end{cases}$$

Relación recursiva del máximo común denominador, donde $x\%y$ expresa el resto de la división entera x/y :

$$\gcd(x,y) = \gcd(y, x\%y)$$

$$\gcd(x,0) = x$$

Nótese que el algoritmo "recursivo" mostrado arriba es, de hecho, únicamente de cola recursiva, lo que significa que es equivalente a un algoritmo iterativo. En el ejemplo siguiente se muestra el mismo algoritmo usando explícitamente iteración. No acumula una cadena de operaciones deferred, sino que su estado es, más bien, mantenido completamente en las variables *x* e *y*. Su "*number of steps grows the as the logarithm of the numbers involved.* ",[8] al español "número de pasos crece a medida que lo hace el logaritmo de los números involucrados."

El algoritmo iterativo requiere una variable temporal, e incluso supuesto el conocimiento del Algoritmo de Euclides es más difícil de entender el proceso a simple vista, aunque los dos algoritmos son muy similares en sus pasos.

2.1.4 Torres de Hanói

Para una detallada discusión de la descripción de este problema, de su historia y de su solución, consúltese el artículo principal.[9][10] El problema, puesto de forma simple, es el siguiente: Dadas 3 pilas, una con un conjunto de N discos de tamaño creciente, determina el mínimo (óptimo) número de pasos que lleva mover todos los discos desde su posición inicial a otra pila sin colocar un disco de mayor tamaño sobre uno de menor tamaño.

Definición de la función:

$$\text{hanoi}(n) = \begin{cases} si & n = 1 & \longrightarrow & 1 \\ si & n > 1 & \longrightarrow & 2 \cdot \text{hanoi}(n-1) + 1 \end{cases}$$

Relación de recurrencia para hanoi:

$$h_n = 2h_{n-1} + 1$$

$$h_1 = 1$$

Ejemplos de implementación:

Aunque no todas las funciones recursivas tienen una solución explícita, la secuencia de la Torre de Hanói puede reducirse a una fórmula explícita.[11]

2.1.5 Búsqueda binaria

El algoritmo de búsqueda binaria es un método de búsqueda de un dato en un vector de datos ordenado dividiendo el vector en dos tras cada pasada. El truco es escoger un punto cerca del centro del vector, comparar en ese punto el dato con el dato buscado para responder entonces a una de las siguientes 3 condiciones: se encuentra el dato buscado, el dato en el punto medio es mayor que el valor buscado o el dato en el punto medio es menor que el valor buscado.

Se usa recursión en este algoritmo porque tras cada pasada se crea un nuevo vector dividiendo en orginal en dos. La subrutina de búsqueda binaria se llama entonces de forma recursiva, cada vez con un vector de menor tamaño. El tamaño del vector se ajusta normalmente cambiando el índice inicial y final. El algoritmo muestra un orden logaritmo de crecimiento porque divide esencialmente el dominio del problema en dos tras cada pasada.

Ejemplo de implementación de la búsqueda binaria:

/* Call binary_search with proper initial conditions. Entrada: Los datos se presentan en forma de vector de [[número enterolnúmeros enteros]] ordenado de forma ascendente, "toFind" es el número entero a buscar, "count" es el número total de elementos del vector Salida: resultado de la búsqueda binaria */ int search(int *data, int toFind, int count) { // Start = 0 (índice inicial) // End = count - 1 (índice superior) return binary_search(data, toFind, 0, count-1); } /* Algoritmo de la búsqueda binaria. Entrada: Los datos se presentan en forma de vector de [[número enterolnúmeros enteros]] ordenado de forma ascendente, "toFind" es el número entero a buscar, "start" es el índice mínimo del vector, "end" es el índice máximo del vector Salida: posición del número entero "toFind" dentro del vector de datos, -1 en caso de búsqueda fallida */ int binary_search(int *data, int toFind, int start, int end) { //Averigua el punto medio. int mid = start + (end - start)/2; //División de enteros //Condición para detenerse. if (start > end) return -1; else if (data[mid] == toFind) //Encontrado? return mid; else if (data[mid] > toFind) //El dato es mayor que "toFind", se busca en la mitad inferior return binary_search(data, toFind, start, mid-1); else //El dato es menor que "toFind", se busca en la mitad superior return binary_search(data, toFind, mid+1, end); }

2.2 Estructuras de datos recursivo (recursión estructural)

Una aplicación de importancia de la recursión en ciencias de la computación es la definición de estructuras de datos dinámicos tales como listas y árboles. Las estructuras de datos recursivos pueden crecer de forma dinámica hasta un tamaño teórico infinito en respuesta a requisitos del tiempo de ejecución; por su parte, los requisitos del ta-

maño de un vector estático deben declararse en el tiempo de complicación.

> "Los algoritmos recursivos son especialmente apropiados cuando el problema que resolver o los datos que manejar son definidos en términos recursivos."[12]

Los ejemplos en esta sección ilustran lo que se conoce como "recursión estructural". Este término se refiere al hecho de que las subrutinas recursivas se aplican a datos que se definen de forma recursiva.

> En la medida en que un programador deriva una plantilla de una definición de datos, las funciones emplean recursión estructural. Es decir, las recursiones en el cuerpo de una función consumen una determinada cantidad de un compuesto dado de forma inmediata.[13]

2.2.1 Listas enlazadas

A continuación se describe una definición simple del nodo de una lista enlazada. Nótese como se define el nodo por si solo. El siguiente elemento del nodo del *struct* es un puntero a un nodo de *struct*.

struct node { int n; // algún tipo de datos struct node *next; // puntero a otro nodo de "struct" }; // LIST no es otra cosa que un nodo de "struct" *. typedef struct node *LIST;

Las subrutinas que operan en la estructura de datos de LIST pueden implementarse de forma natural como una subrutina recursiva porque la estructura de datos sobre la que opera (LIST) es definida de forma recursiva. La subrutina *printList* definida a continuación recorre la lista hacia abajo hasta que ésta se vacía (NULL), para cada nodo imprime el dato (un número entero). En la implementación en C, la lista permanece inalterada por la subrutina *printList*.

void printList(LIST lst) { if (!isEmpty(lst)) // caso básico { printf("%d ", lst->n); // imprime el entero seguido por un espacio printList(lst->next); // llamada recursiva } }

2.2.2 Árboles binarios

Más abajo se muestra una definición simple de un nodo de árbol binario. Al igual que el nodo de listas enlazadas, se define a sí misma (de forma recursiva). Hay dos punteros que se refieren a sí mismos – *left* (apuntando a l aparte izquierda del subárbol) y *right* (a la parte derecha del subárbol).

struct node { int n; // algún tipo de datos struct node *left; // puntero al subárbol izquierdo struct node *right; // puntero al subárbol derecho }; // TREE no es otra cosa que un nodo " struct " typedef struct node *TREE;

Las operaciones en el árbol pueden implementarse usando recursión. Nótese que, debido al hecho de que hay dos punteros que se referencian a sí mismos (izquierda y derecha), esas operaciones del árbol van a necesitar dos llamadas recursivas. Para un ejemplo similar, véase la función de Fibonacci y la explicación siguiente.

void printTree(TREE t) { if (!isEmpty(t)) { // caso básico printTree(t->left); // ir a la izquierda printf("%d ", t->n); // imprimir el entero seguido de un espacio printTree(t->right); // ir a la derecha } }

El ejemplo descrito ilustra un árbol binario de orden transversal. Un árbol de búsqueda binaria es un caso especial de árbol binario en el cual los datos de cada árbol están en orden.

2.3 Recursión frente a iteración

En el ejemplo "factorial" la implementación iterativa es probablemente más rápida en la práctica que la recursiva. Esto es casi definido por la implementación del algoritmo euclideano. Este resultado es lógico, pues las funciones iterativas no tienen que pagar el exceso de llamadas de funciones como en el caso de las funciones recursivas, y ese exceso es relativamente alto en muchos lenguajes de programación (nótese que mediante el uso de una *lookup table* es una implementación aún más rápida de la función factorial).

Hay otros tipos de problemas cuyas soluciones son inherentemente recursivas, porque estar al tanto del estado anterior. Un ejemplo es el árbol transversal; otros incluyen la función de Ackermann y el algoritmo divide y vencerás tales como *Quicksort*. Todos estos algoritmos pueden implementarse iterativamente con la ayuda de una *pila*, pero la necesidad del mismo, puede que anule las ventajas de la solución iterativa.

Otra posible razón para la utilización de un algoritmo iterativo en lugar de uno recursivo es el hecho de que en los lenguajes de programación modernos, el espacio de *stack* disponible para un hilo es, a menudo, mucho menos que el espacio disponible en el montículo, y los algoritmos recursivos suelen requerir más espacio de *stack* que los algoritmos iterativos. Véase, por otro lado, la sección siguiente que trata el caso especial de la recursión de cola.

3 Funciones de recursión de cola

Funciones de recursión de cola son funciones que finalizan con una llamada recursiva que no crea ninguna operación deferida. Por ejemplo, la función gcd (se muestra de nuevo más abajo) es recursiva de cola; sin embargo,

la función factorial (que también se muestra más abajo) **no** es recursiva de cola porque crea operaciones diferidas que tienen que realizarse incluso después de que se complete la última llamada recursiva. Con un compilador que automáticamente optimiza llamadas recursivas de cola, una función recursiva de cola, como por ejemplo gcd, se ejecutará usando un espacio constante. Así, el proceso que genera es esencialmente iterativo y equivalente a usar estructuras de control de lenguaje imperativo como los bucles for y while.

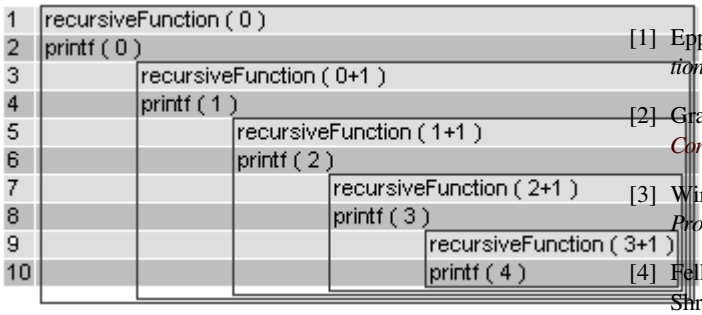
La importancia de recursión de cola es que cuando se realiza una llamada recursiva de cola, la posición de retorno de la función que llama necesita grabarse en el call stack; cuando la función recursiva retorna, continuará directamente a partir de la posición de retorno grabada previamente. Por ello, en compiladores que dan soporte a optimización de recursión de cola, este tipo de recursión ahorra espacio y tiempo.

4 Orden en el llamamiento de una función

El orden de llamamiento de una función puede alterar la ejecución de una función, véase este ejemplo en C:

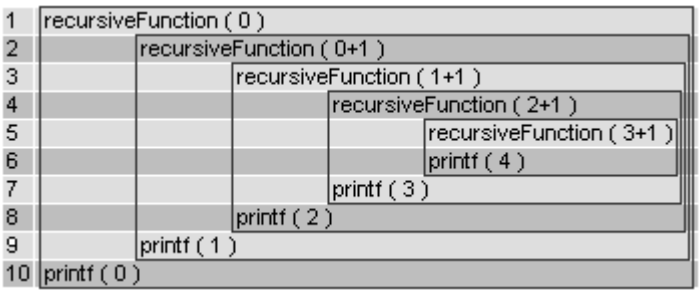
4.1 Función 1

```
void recursiveFunction(int num) { if (num < 5) { printf("%d\n", num); recursiveFunction(num + 1); } }
```



4.2 Función 2 con líneas cambiadas

```
void recursiveFunction(int num) { if (num < 5) { recursiveFunction(num + 1); printf("%d\n", num); } }
```



5 Recursión directa e indirecta

Se habla de recursión directa cuando la función se llama a sí misma. Se habla de recursión indirecta cuando, por ejemplo, una función A llama a una función B, que a su vez llama a una función C, la cual llama a la función. De esta forma es posible crear largas cadenas y ramificaciones, véase Parser descendente recursivo.

6 Véase también

- Recursión primitiva
- Programación funcional
- Función de Ackermann

7 Notas y referencias

- Esta obra deriva de la traducción de *Recursion (computer science)* de Wikipedia en inglés, concretamente de esta versión, publicada por sus editores bajo la Licencia de documentación libre de GNU y la Licencia Creative Commons Atribución-CompartirIgual 3.0 Unported.

[1] Epp, Susanna (1995). *Discrete Mathematics with Application* (2. edición). p. 427.

[2] Graham, Ronald; Donald Knuth, Oren Patashnik (1990). *Concrete Mathematics*. Capítulo 1: Recurrent Problems.

[3] Wirth, Niklaus (1976). *Algorithms + Data Structures = Programs*. Prentice-Hall. p. 126.

[4] Felleisen, Matthias; Robert Bruce Findler, Matthew Flatt, Shriram Krishnamurthi (2001). *How to Design Programs: An Introduction to Computing and Programming*. Cambridge, MASS: MIT Press. p. art V “Generative Recursion”.

[5] Felleisen, Matthias (2002). «Developing Interactive Web Programs». En Jeuring, Johan. *Advanced Functional Programming: 4th International School*. Oxford, Reino Unido: Springer. p. 108..

[6] Epp, Susanna (1995). *Discrete Mathematics with Applications*. Brooks-Cole Publishing Company. p. 424.

[7] Abelson, Harold; Gerald Jay Sussman (1996). *Structure and Interpretation of Computer Programs*. Sección 1.2.2.

[8] Abelson, Harold; Gerald Jay Sussman (1996). *Structure and Interpretation of Computer Programs*. Section 1.2.5.

[9] Graham, Ronald; Donald Knuth, Oren Patashnik (1990). *Concrete Mathematics*. Chapter 1, Section 1.1: The Tower of Hanoi.

[10] Epp, Susanna (1995). *Discrete Mathematics with Applications* (2nd edición). pp. 427–430: The Tower of Hanoi.

[11] Epp, Susanna (1995). *Discrete Mathematics with Applications* (2nd edición). pp. 447–448: An Explicit Formula for the Tower of Hanoi Sequence.

[12] Wirth, Niklaus (1976). *Algorithms + Data Structures = Programs*. Prentice-Hall. p. 127.

[13] Felleisen, Matthias (2002), «Developing Interactive Web Programs», *Advanced Functional Programming: 4th International School*, Oxford, UK: Springer, pp. 108.

8 Enlaces externos

- Harold Abelson and Gerald Sussman: “Structure and Interpretation Of Computer Programs”
- IBM DeveloperWorks: “Mastering Recursive Programming”
- David S. Touretzky: “Common Lisp: A Gentle Introduction to Symbolic Computation”
- Matthias Felleisen: “How To Design Programs: An Introduction to Computing and Programming”

9 Origen del texto y las imágenes, colaboradores y licencias

9.1 Texto

- **Recursión (ciencias de computación)** *Fuente:* https://es.wikipedia.org/wiki/Recursi%C3%B3n_(ciencias_de_computaci%C3%B3n)?oldid=82065862 *Colaboradores:* Chobot, CEM-bot, Marianov, Escarbot, JAnDbot, Rafa3040, Shooke, Muro Bot, Drinibot, Bigsus-bot, Dnu72, Poco a poco, Juan Mayordomo, Luckas-bot, MalchikGii, Jkbw, AstaBOTh15, TiriBOT, RedBot, KamikazeBot, GrouchoBot, EmausBot, Iwèr, Xerox 5B, KLBot2, MetroBot, Acratta, Carocad y Anónimos: 17

9.2 Imágenes

- **Archivo:RecursiveFunction1_execution.png** *Fuente:* https://upload.wikimedia.org/wikipedia/commons/8/8a/RecursiveFunction1_execution.png *Licencia:* Public domain *Colaboradores:* I made it myself *Artista original:* User:Maxtremus
- **Archivo:RecursiveFunction2_execution.png** *Fuente:* https://upload.wikimedia.org/wikipedia/commons/0/0d/RecursiveFunction2_execution.png *Licencia:* Public domain *Colaboradores:* I made it myself *Artista original:* User:Maxtremus
- **Archivo:RecursiveTree.JPG** *Fuente:* https://upload.wikimedia.org/wikipedia/commons/f/f7/RecursiveTree.JPG *Licencia:* Public domain *Colaboradores:* Trabajo propio *Artista original:* Brentsmith101

9.3 Licencia del contenido