La Recursividad en programación

Del Grecco Daiana Betania

UTN Facultad Regional Resistencia

Laboratorio de Computacion I

Ing. Carolina Vargas

06 Mayo 2023

Índice

Parte Práctica

1.La Recursividad

La recursividad es un método que define funciones en las cuales, su estructura consiste en llamarse a sí misma repetidamente hasta que se cumpla con una condición de fin. Este proceso se utiliza para cálculos repetitivos en los que cada acción se determina en función de un resultado anterior.

Una función recursiva bien diseñada incluye una estructura de decisión que evalúa si el procedimiento debe continuar repitiéndose o detenerse. Cuando se detiene, esto indica que se ha cumplido la condición base (casos básicos), lo que lleva a finalizar las llamadas a sí misma para devolver un valor y reconstruir las llamadas anteriores.

Byron Gottfried (2005) dice que un problema puede resolverse recursivamente si se cumplen dos condiciones. Primero, el problema debe escribirse en forma recursiva. Segundo, la especificación del problema debe incluir una condición de fin (p.246).

2. Ventajas de la recursividad

En algunos casos, la recursividad es una opción más simple y cómoda para diseñar una función que también podría realizarse con estructuras de iteración o bucles anidados. Una de las ventajas que presenta, es que son más fáciles de leer y no necesitan varias líneas de código para obtener un resultado, sino que la misma función retorna el valor deseado.

A continuación, la resolución de un problema utilizando recursividad y estructura iterativa:

```
/*Estructura iterativa para el factorial de un numero*/
int main(int argc, char *argv[]) {
	int num,i,resultado;
	resultado=1;
	printf("ingrese numero para calcular su factorial:");
	scanf("%d",&num);
	for(i=1;i<=num;i++){
		printf(" %d x %d =",resultado,i);
		resultado=(resultado * i);
		printf(" %d\n",resultado)
	}
	printf("El resultado es : %d",resultado);
	return 0;
```

```
}


/*Funcion recursiva para el factorial de un numero*/
//Definimos la funcion recursiva
int factorial(int n) {
	if (n == 0) {  // caso base
		return 1; //La función devuelve 1 * (1), que es igual a 1.
	} else {
		return n * factorial(n - 1); // llamada recursiva
	}
}
//llamamos a la funcion en el main
#include <stdio.h>
int main() {
	int n ;  // número para calcular el factorial
	int resultado;
	printf("ingrese numero para calcular su factorial:\n");
	scanf("%d",&n);
	resultado = factorial(n); //llamada a funcion que da solo el resultad
	printf("\nEl factorial de %d es %d ---> resultado final\n\n", n, resultado);
	return 0;
}
```

En los ejemplos anteriores, vemos que las líneas de códigos disminuyen notoriamente en el caso de la función recursiva, esta permite dividir el código en partes y hacerlo más legible ya que las instrucciones del procedimiento, que tienen que devolver un resultado, se encuentran fuera de la función main y dentro de la misma función.

La recursividad usa la pila por lo tanto facilita las llamadas recursivas. Esto significa que una función es libre de llamarse de nuevo, porque se creará un nuevo stack frame para todas sus variables locales.

Una pila es una estructura «lasí-in, first-out» (último en entrar, primero en salir) en la que los datos sucesivos se «colocan encima» de los anteriores. Los datos se «sacan» después en orden inverso de la pila, como indica la designación «last-in, first-out». (Byron Gottfried,2005, p.246)

3. Recursividad e Iteración

Generalmente, los procesos de recursividad son una forma práctica de solucionar problemas dentro del diseño de algoritmos, pero no siempre son eficientes. Habrá casos donde será mucho más coherente utilizar las estructuras de iteración. Ya que estas no requieren de espacios de almacenamiento como las pilas, que hacen su proceso más lento a la hora de ejecutarse.

La estructura de iteración puede requerir más líneas de código o un análisis estricto sobre las variables que la controlan (variables de inicialización, terminación, contadores, etc.) pero nada de eso es un factor limitante cuando la eficiencia de su uso consiste en no generar accidentes por la recursión infinita de una función, que puede genera un bloqueo del sistema, o una ejecución más lenta para conseguir un valor.

La función recursiva es fácil de escribir, pero no tienen un buen rendimiento en comparación con la iteración mientras que la iteración es difícil de escribir, pero su rendimiento es bueno en comparación con la recursión.

4.Programar recursivamente un proceso iterativo

Si se programa recursivamente un proceso iterativo, ¿será necesariamente más eficiente el programa que el correspondiente a una versión no recursiva?

Hay casos donde la recursión puede ser más eficiente que una estructura de iterativa, especialmente cuando se necesita descomponer en partes el código para enfocarse en resolver los problemas con procesos recursivos. Entonces, la función recursiva puede conducir a un código más simple y fácil de entender, lo que puede ser beneficioso para la mantenibilidad, la escalabilidad del software y prolijidad del algoritmo.

Se debe evaluar detenidamente que parámetros se desean seguir a la hora de crear un programa y evaluar en cada caso si una opción recursiva es posible en reemplazo de una iterativa.

5.Variables locales dentro de una función recursiva.

Las variables locales son útiles porque permiten a las funciones realizar cálculos y almacenar datos sin afectar el estado de las variables fuera de la función. Además, también permiten reutilizar el mismo nombre de variable en diferentes partes de un programa sin conflictos. Por ejemplo, podríamos tener una función fn(int a), donde el parámetro se llama a, así como también dentro de la función main() definir una variable "a" sin que esto cause problemas.

Entonces ¿cómo se interpretan las variables locales dentro de una función recursiva?

(Byron Gottfried,2005, p.246) Si una función recursiva contiene variables locales, se creará un conjunto diferente de variables locales durante cada llamada. Los nombres de las variables locales serán, por supuesto, siempre los mismos, como se hayan declarado en la función. Sin embargo, las variables representarán un conjunto diferente de valores cada vez que se ejecute la función. Cada conjunto de valores se almacenará en la pila; así se podrá disponer de ellas cuando el proceso recursivo se «deshaga», es decir, cuando las llamadas a la función se «saquen» de la pila y se ejecuten.

En resumen, cada llamada recursiva tendrá su propia copia de las variables locales de la función, que se inicializarán con los valores proporcionados en esa llamada específica. A medida que se realizan más llamadas recursivas, se crean nuevas copias de las variables locales para cada instancia de la función.

6. Ejemplo y explicación de la recursividad.

EJEMPLO ENCONTRAR EL FACTORIAL

Ejemplos a partir n = 4

4!

1 x 2 =2

2 X 3 =6

6 X 4!=24

La función recursiva factorial resta sucesivamente 1 para recorrer los valores anteriores del 4. La función se llama a sí misma y se utiliza múltiples veces hasta que llega al valor base (primer número a multiplicar)

Despues de la primera llamada, automáticamente se llama a si misma hasta que se cumpla una condición que la detenga.

n * factorial (n - 1)

4 * factorial (4 - 1) -->(3) PRIMER LLAMADO A LA FUNCION

--> 3 * factorial (3 - 1) -->(2) SUCESION AUTOMATICA

--> 2 * factorial (2 - 1) -->(1) SUCESION AUTOMATICA

--> 1 * factorial (1- 1) = factorial (0) SUCESION AUTOMATICA

Cuando n==0,se consiguieron todos los números a multiplicar y se multiplican. Sí n==0 En la función factorial(int n) entonces devuelve el valor 1.

--> Entonces 1 * (factorial (0)) = 1 Valor por el cual termina la sucesión

--> Entonces 2 * (2 - 1) = 2

--> Entonces 3 * (3 - 1) = 6

Entonces 6 * (4) = 24

```
#include <stdio.h>
int factorial(int n);//devuelve solo el resultado final
void mostrarfactorial(int n); //muestra por pantalla el procedimiento
void mostrarmultiplicacion(int n);//muestra por pantalla la multiplicacion
//--MAIN----------
int main() {
        int n ;  // número para calcular el factorial
        int resultado;
```

```
	printf("ingrese numero para calcular su factorial:\n");
	scanf("%d",&n);
	resultado = factorial(n); //llamada a funcion que da solo el resultado

	printf("Procedimiento de la funcion factorial:\n");
	mostrarfactorial(n);	//llamada a funcion que muestra el procedimiento
	mostrarmultiplicacion(n); //llamada a funcion que muestra la multiplicacion
	printf("\nEl factorial de %d es %d ---> resultado final\n\n", n, resultado);
	return 0;}

//---FUNCIONES Y VOID--
int factorial(int n) {
	if (n == 0) {  // caso base
		return 1; //La función devuelve 1 * (1), que es igual a 1.
	} else {
		return n * factorial(n - 1); // llamada recursiva
	}
}
void mostrarfactorial(int n){
int i;
for(i=n;n>=1;i--){
	if(n==1){
		printf("%d * factorial(%d - 1) ----->  finaliza sucesion de la funcion\n\n",n,n);
		}else{
			printf("%d * factorial(%d - 1)\n",n,n);
		}
	}
n=n-1;
```

```
}
void mostrarmultiplicacion(int n){
	int i,resultado;
	resultado=1;

	for(i=1;i<=n;i++){

		if(i==1){
			printf("\t %d x %d =",resultado,i);
			resultado=(resultado * i);
			printf(" %d   -----> Comienza a multiplicar desde el ultimo valor que dio la funcion\n",resultado);
		}else{
			printf("\t %d x %d =",resultado,i);
			resultado=(resultado * i);
			printf(" %d\n",resultado);
		}
	}
}
```

Parte práctica: Ejercicios de Recursividad

1.Calcular el factorial de un número

```
#include <stdio.h>
int factorial(int n); //Definicion de funcion

int main() {
	int numero, resultado;
	printf("Ingresa un numero para calcular su factorial: ");
	scanf("%d", &numero);
	resultado = factorial(numero);// resultado contiene el valor final

	printf("El factorial de %d es %d\n", numero, resultado);
	return 0;
}
//Funcion recursiva
int factorial(int n) {
	if (n == 0 || n == 1) {
		return 1;
	} else {
		return n * factorial(n - 1);
	}
}
```

2. Calcular la serie de Fibonachi, los 10 primeros números.

```
#include <stdio.h>
int fibonachi(int n);
int main(int argc, char *argv[]) {
	int num,resultado;

	printf("Ingrese cantidad de num fibonachi\n");
	scanf("%d",&num);

	resultado=fibonachi(num);
	printf("\nresultado: %d",resultado);

	return 0;
}

int fibonachi(int n) {
	if (n <2) {  // caso base

		return n; //La función devuelve n(1 o 0)

	} else {

		return (fibonachi(n-1))+(fibonachi(n-2)); // llamada recursiva

	}
}
```

3. Escriba una función recursiva MCD que regrese el máximo común divisor de x y de y. El máximo común divisor de los enteros x e y es el número más grande que divide en forma completa tanto a x como a y. El gcd de x y de y, se define en forma recursiva como sigue: Si y es igual a 0, entonces gcd (de x, y) es x; de lo contrario gcd (de x, y) es igual a gcd(y, x%y).

```
#include <stdio.h>

int mcd(int a,int b); //Funcion Definida


int main(int argc, char *argv[]) {
	int x,y,resultado;


	printf("Ingrese num x\n");
	scanf("%d",&x);


	printf("Ingrese num y\n");
	scanf("%d",&y);


	resultado=mcd(x,y);


	printf("El Mayor Comun divisor de ambos es : %d",resultado);


	return 0;
}
//Funcion recursiva
int mcd(int a,int b){
	if(b==0){
		return a;
	}else{
```

```
            return mcd(b, a%b);
        }
}
```

4. Triángulo de Pascal.

```
#include <stdio.h>
#include <math.h>
/*
Triángulo de Pascal.
IMPORTANTE: Este programa fue simplificado para utilizarlo como ejemplificación
El triángulo quedara asimétrico si se ingresan valores mayores
a 6.
*/

int factorial(int n);
int combinacion(int n,int k);

int main(int argc, char *argv[]) {
        int n,i,j,resultadocombinacion;
        printf("Ingrese num: ");
        scanf("%d",&n);
        int numGuiones = n;

        for(i=0;i<=n;i++){     //bucle para fila
                if(i<n){
                        for (j = 0; j < numGuiones; j++){ //Cantidad de espacio antes de fila
                                printf("  ");
```

```
                }
                numGuiones--;
            }
            for(j=0;j<=i;j++) { //bucles para columnas
                resultadocombinacion=combinacion(i,j);
                printf("  %d ",resultadocombinacion);//con espacios en blanco
            }
            printf("\n"); // salto de línea para cada fila.
        }
        return 0;
}

int factorial(int n) {
        if (n == 0) {  // caso base

            return 1; //La función devuelve 1 * (1), que es igual a 1.

        } else {

            return n * factorial(n - 1); // llamada recursiva
        }
}
//funcion con ecuacion para triangulo de pascal
int combinacion(int n,int k){
        int resultado;
        resultado=(factorial(n))/(factorial(k)*factorial(n-k));
        return resultado;
}
```

Referencias

Gottfried, B. (2005). "Programación en C". México: McGraw-Hill Interamericana. p.246.

CódigoFacilito. (2019). ¿Qué es la recursividad y cómo funciona en programación? CódigoFacilito. Recuperado de https://codigofacilito.com/articulos/articulo_16_10_2019_16_22_35

Gadget-Info. (s.f.). Diferencia entre recursión y bucle en programación. Gadget-Info. Recuperado de https://es.gadget-info.com/difference-between-recursion

Recursividad | FÁCIL de entender y visualizar | Recursión [Archivo de video]. (2018, abril 23). YouTube. https://youtu.be/hyTShTfoPYg

La MAGIA de la RECURSIVIDAD [Archivo de video]. (2021, mayo 5). YouTube. https://youtu.be/yX5kR63Dpdw

5 Simple Steps for Solving Any Recursive Problem [Archivo de video]. (2020, septiembre 22). YouTube. https://youtu.be/e4S8zfLdLgQ