Tema 7:

Recursividad

***Objetivos**: en este tema estudiaremos funciones recursivas; esto es, funciones que se invocan a sí mismas. Estas funciones son equivalentes a estructuras tipo bucle pero permiten especificar muchos problemas de un modo más simple y natural que éstos, de ahí su importancia en computación.*

Índice

Índice 2

1 Introducción 3

2 Tipos de recursión 3

2.1 Recursión lineal 4

2.1.1 Recursión lineal no final 4

2.1.2 Recursión lineal final 5

2.2 Recursión múltiple 6

2.3 Recursión mutua 7

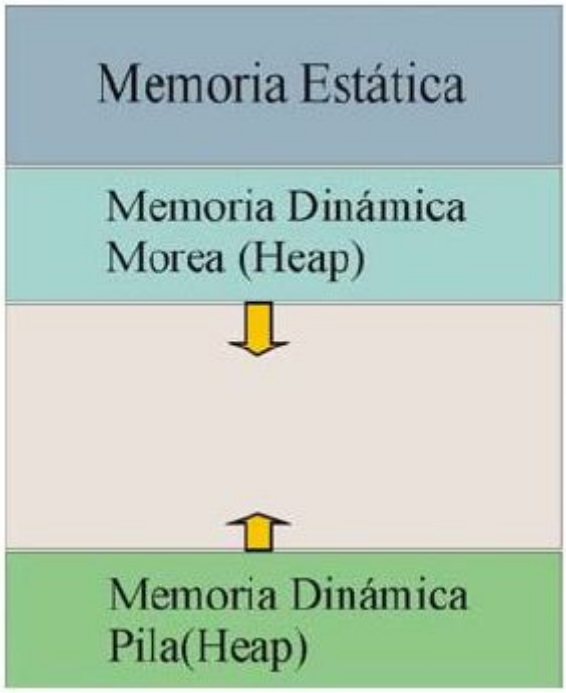
3 Ejercicios 9

4 Apéndice: las torres de Hanoi 10

4.1 Leyenda sobre las torres de Hanoi 11

1 Introducción

Una función recursiva es una función que se llama *a si misma*. Esto es, dentro del cuerpo de la función se incluyen llamadas a la propia función. Esta estrategia es una alternativa al uso de bucles. Una solución recursiva es, normalmente, menos eficiente que una solución basada en bucles. Esto se debe a las operaciones auxiliares que llevan consigo las llamadas a las funciones. Cuando un programa llama a una función que llama a otra, la cual llama a otra y así sucesivamente, las variables y valores de los parámetros de cada llamada a cada función se guardan en la pila o stack, junto con la dirección de la siguiente línea de código a ejecutar una vez finalizada la ejecución de la función invocada. Esta pila va creciendo a medida que se llama a más funciones y decrece cuando cada función termina. Si una función se llama a si misma recursivamente un número muy grande de veces existe el riesgo de que se agote la memoria de la pila, causando la terminación brusca del programa.



A pesar de todos estos inconvenientes, en muchas circunstancias el uso de la recursividad permite a los programadores especificar soluciones naturales y sencillas que sin emplear esta técnica serían mucho más complejas de resolver. Esto convierte a la recursión en una potente herramienta de la programación. Sin embargo, por sus inconvenientes, debe emplearse con cautela.

2 Tipos de recursión

Como regla básica, para que un problema pueda resolverse utilizando recursividad, el problema debe poder definirse recursivamente y, segundo, el problema debe incluir una

condición de terminación porque, en otro caso, la ejecución continuaría indefinidamente. Cuando la condición de terminación es cierta la función no vuelve a llamarse a si misma.

Pueden distinguirse distintos tipos de llamada recursivas dependiendo del número de funciones involucradas y de cómo se genera el valor final. A continuación veremos cuáles son.

2.1 Recursión lineal

En la recursión lineal cada llamada recursiva genera, como mucho, otra llamada recursiva. Se pueden distinguir dos tipos de recursión lineal atendiendo a cómo se genera resultado.

2.1.1 Recursión lineal no final

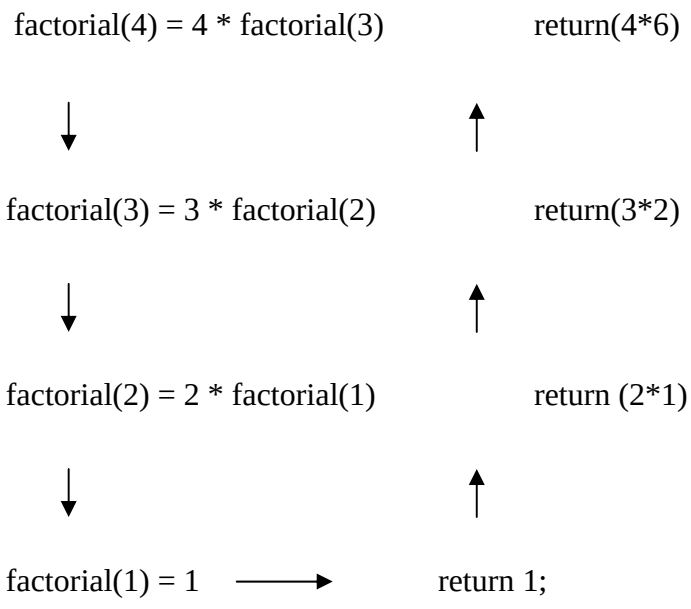
En la recursión lineal no final el resultado de la llamada recursiva se combina en una expresión para dar lugar al resultado de la función que llama. El ejemplo típico de recursión lineal no final es cálculo del factorial de un número (n! = n * (n-1) * ...* 2 * 1). Dado que el factorial de un número n es igual al producto de n por el factorial de n-1, lo más natural es efectuar una implementación recursiva de la función factorial. Veamos una implementación de este cálculo:

```
/*
*ejemplo7_1.c
*/
#include <stdio.h>

int factorial(int numero){
  if (numero > 1) return (numero*factorial(numero-1));
  else return(1);
}

int main (){
  int n;
  printf("Introduce el número: ");
  scanf("%d",&n);
  printf("El factorial es %d", factorial(n));
}
```

para calcular el factorial de 4 esta sería la secuencia de llamadas:



Cada fila del anterior gráfico supone una instancia distinta de ejecución de la función fact. Cada instancia tiene un conjunto diferente de variables locales.

2.1.2 Recursión lineal final

En la recursión lineal final el resultado que es devuelto es el resultado de ejecución de la última llamada recursiva. Un ejemplo de este cálculo es el máximo común divisor, que puede hallarse a partir de la fórmula:

$$mcd(n,m) = \begin{cases} n & n = m \\ mcd(n-m,m) & n > m \\ mcd(n,m-n) & n < m \end{cases}$$

```
/*
*ejemplo7_2.c
*/
#include <stdio.h>

long mcd(long,long);

main(int argc, char *argv[]){
   long a= 4454,b= 143052;
   printf("El m.c.d. de %ld y %ld es %ld\n",a,b,mcd(a,b));
}
```

```
long mcd(long a, long b){
   if (a==b) return a;
   else if (a<b) return mcd(a,b-a);
   else  return mcd(a-b,b);
}
```

2.2 Recursión múltiple

Alguna llamada recursiva puede generar más de una llamada a la función. Uno de los centros más típicos son los números de Fibonacci, números que reciben el nombre del matemático italiano que los descubrió. Estos números se calculan mediante la fórmula:

$$F(n) = \begin{cases} 1\,, & \text{si } n = 1\,; \\ 1, & \text{si } n = 2; \\ F(n-1) + F(n-2) & \text{si } n > 2; \end{cases}$$

Estos números poseen múltiples propiedades, algunas de las cuales todavía siguen descubriéndose hoy en día, entre las cuales están:

- La razón (el cociente) entre un término y el inmediatamente anterior varía continuamente, pero se estabiliza en un número irracional conocido como razón áurea o número áureo, que es la solución positiva de la ecuación x^2-x-1=0, y se puede aproximar por 1,618033989.
- Cualquier número natural se puede escribir mediante la suma de un número limitado de términos de la sucesión de Fibonacci, cada uno de ellos distinto a los demás. Por ejemplo, 17=13+3+1, 65=55+8+2.
- Tan sólo un término de cada tres es par, uno de cada cuatro es múltiplo de 3, uno de cada cinco es múltiplo de 5, etc. Esto se puede generalizar, de forma que la sucesión de Fibonacci es periódica en las congruencias módulo m, para cualquier m.
- Si F(p) es un número primo, p también es primo, con una única excepción. F(4)=3; 3 es primo, pero 4 no lo es.
- La suma infinita de los términos de la sucesión F(n)/10^n es exactamente 10/89.

Veamos un programa que nos permite calcular el número n de la serie de Fibonacci:

```
/*
*ejemplo7_3.c
*/
#include <stdio.h>

long fibonacci (int);

int main()
{
  int n= 30;

    printf("El %dº número de Fibonacci es %ld\n", n,
fibonacci(n));
}

long fibonacci(int n)
{
  if (1 == n || 2 == n) {
    return 1;
  } else {
    return (fibonacci(n-1) + fibonacci(n-2));
  }
}
```

2.3 Recursión mutua

Implica más de una función que se llaman mutuamente. Un ejemplo es el determinar si un número es par o impar mediante dos funciones:

```
/*
*ejemplo7_4.c
*/
#include <stdio.h>

long fibonacci (int);

int main(){
  int n= 30;
  if (par(n))
    printf("El número es par");
    else
    printf("El número es impar");
}

int par(int n){
  if (n==0) return 1;
  else return (impar(n-1));
}

int impar(int n){
  if (n==0) return 0;
```

```
  else return(par(n-1));
}
```

Veamos un ejemplo de ejecución de este programa:

```
par(5) -> return(impar(4))    return(0)  (no es par)
impar(4) -> return(par(3))    return(0)
par(3) -> return(impar(2))    return(0)
impar(2) -> return(par(1))    return(0)
par(1) -> return(impar(0))    return(0)
```

impar(0) -> return(0)

3 Ejercicios

1. Construir una función recursiva que calcule la suma de los n primeros números naturales.
2. Construir una función recursiva que imprima la lista de números naturales comprendidos entre dos valores a y d dados por el usuario.
3. Escribir una función recursiva que devuelva la cantidad de dígitos de un número entero.
4. Escribir una función recursiva que calcule x^y mediante multiplicaciones sucesivas, siendo x e y dos números enteros.
5. Escribir una función recursiva que calcule x*y mediante sumas sucesivas, siendo x e y dos números enteros.
6. Calcula mediante un diseño recursivo el valor máximo de un vector de componentes numéricas.
7. Construir una función recursiva que cuente el número de secuencias de dos 1 seguidos que hay en una cadena de caracteres que represente un número binario.
8. Escribir la función recursiva que recibiendo como parámetros una cadena de dígitos hexadecimales y su longitud devuelva el valor decimal que representa dicha cadena.
9. Modificar el programa anterior para que la secuencia de 1 sea de longitud m.
10. Calcular C (n,k) siendo:

    C (n,0)= C (n,n)=1 si n>=0

    C (n,k)= C (n-1,k) + C (n-1,k-1) si n>k>0

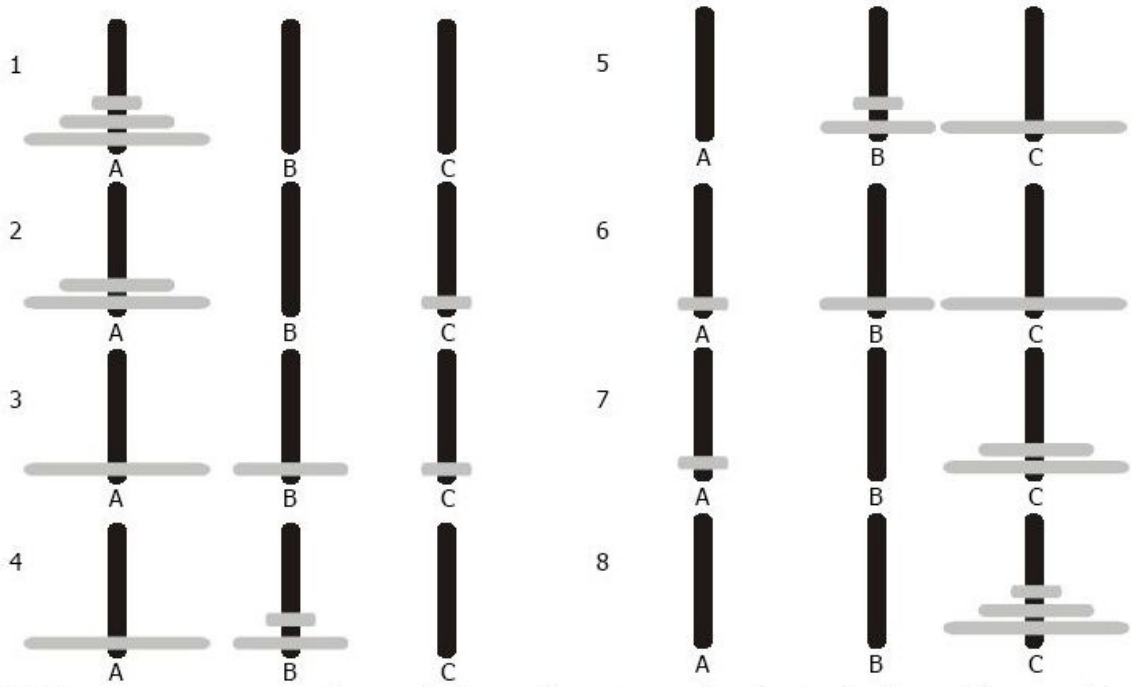
4 Apéndice: las torres de Hanoi

Las torres de Hanoi son un conocido juego de niños que se juega con tres pivotes y un cierto número de discos de diferentes tamaños. Los discos tienen un agujero en el centro y se pueden apilar en cualquiera de los pivotes. Inicialmente, los discos se amontonan en el pivote de la izquierda por tamaño decreciente como se muestra en la figura.

El objeto del juego es llevar los discos del pivote más a la izquierda al de más a la derecha. No se puede colocar nunca un disco sobre otro más pequeño y sólo se puede mover un disco a la vez. Cada disco debe encontrarse siempre en uno de los pivotes.

El problema de las torres de Hanoi puede resolverse fácilmente usando recursividad. La estrategia consiste en considerar uno de los pivotes como el origen y otro como destino. El problema de mover n discos al pivote derecho se puede formular recursivamente como sigue:

11. mover los n-1 discos superiores del pivote izquierdo al del centro empleando como sería el pivote de la derecha.
12. mover el n-ésimo disco (el más grande) al pivote de la derecha.
13. mover los n-1 discos del pivote del centro al de la derecha.



Este problema tiene una complejidad computacional exponencial: el número mínimo de movimientos para mover n discos es 2^n-1.

4.1 Leyenda sobre las torres de Hanoi

Cuenta la leyenda que Dios al crear el mundo, colocó tres varillas de diamante con 64 discos en la primera. También creó un monasterio con monjes, los cuales tienen la tarea de resolver esta Torre de Hanoi divina. El día que estos monjes consigan terminar el juego, el mundo acabará en un gran estruendo.

El mínimo número de movimientos que se necesita para resolver este problema es de 2^{64}-1. Si los monjes hicieran un movimiento por segundo, las 64 torres estarían en la tercera varilla en poco menos de 585 mil millones de años. Como comparación para ver la magnitud de esta cifra, la Tierra tiene unos 5 mil millones de años, y el Universo entre 15 y 20 mil millones de años de antiguedad, solo una pequeña fracción de esa cifra.

Ejercicio: Realice una implementación en C se permita resolver el problema de las torres de Hanoi.