Funciones recursivas

He sido estudiante de programaciÃ³n y entiendo la complejidad que entraÃ±a diseÃ±ar y tratar de entender las funciones recursivas, por ello voy a realizar un compendio de algunas funciones, que utilice o diseÃ±Ã© durante mis aÃ±os de estudiante de programaciÃ³n, para ello os pongo una pequeÃ±a definiciÃ³n y/o explicaciÃ³n de lo que definen los libros e internet como funciÃ³n recursiva y seguidamente algunas funciones recursivas a modo de ejemplo:

“Tipo de funciÃ³n de programaciÃ³n mÃ¡s compleja. Una funciÃ³n comÃºn es llamada desde otra funciÃ³n; pero es posible crear funciones que puedan llamarse a sÃ− mismas, las funciones recursivas.
Una funciÃ³n puede ser recursiva tanto de forma directa (si es llamada a sÃ− misma) o de forma indirecta (si llama a una funciÃ³n que luego la llama).
Existen algunos problemas que pueden ser resueltos de forma mÃ¡s eficiente (o su resoluciÃ³n puede ser mÃ¡s naturalmente pensada) utilizando funciones recursivas.
Una funciÃ³n recursiva puede dar origen a un tÃ−pico problema en programaciÃ³n, la recursiÃ³n infinita ,que es cuando una funciÃ³n se llama a sÃ− misma infinitas veces. Esto detiene el normal funcionamiento de un programa.
Para que esto no suceda una funciÃ³n recursiva debe ser muy bien pensada. Principalmente una funciÃ³n recursiva debe saber resolver el caso mÃ¡s simple, llamado caso base. Si la funciÃ³n es llamada con el caso base, inmediatamente retorna el resultado (no necesita volver a llamarse a sÃ− misma para poder resolverlo).
Si la funciÃ³n es llamada con un caso mÃ¡s complejo, las sucesivas llamadas a sÃ− mismas irÃ¡n virtualmente descomponiendo ese caso hasta llegar a un caso base, para luego determinar el resultado final de la funciÃ³n”

- FunciÃ³n que calcula el factorial de un nÃºmero (en Java)

```
static int factorial(int nÃºmero){
if ( nÃºmero <= 1 ) {
return 1;
} else {
return nÃºmero*factorial(nÃºmero-1);
}
}
```

- FunciÃ³n que calcula la serie de Fibonacci (en C)

```
int fib(int n){
Â  if ((n == 0) || (n == 1)) return 1;
Â  else return fib(n-1) + fib(n-2);
}
```

- AlgorÃ−tmo que realiza la curva de Koch

```
Iniciar_Koch( Punto_Inicial, Longitud, NÃºmero_Iteraciones )
1. Declarar Cursor como un punto y un Ã¡ngulo
2. Inicializar Cursor:
3. Cursor.punto <- Punto_Inicial
4. Cursor.angulo <- 0
```

5. Dibujar_Koch(Cursor, Longitud, NÃºmero_Iteraciones)
6. Terminar

Dibujar_Koch(Cursor, Dist, N)
1. Si N = 0 entonces, // A(vanzar)
2. Cursor.punto <- Avanzar(Cursor, Dist)
3. Terminar
4. Si no, entonces, // Recursividad: A->AIADDAIA
5. Dibujar_Koch(Cursor, Dist/3, N-1) // A
6. Cursor.angulo <- Cursor.angulo + Ï“/3 // I
7. Dibujar_Koch(Cursor, Dist/3, N-1) // A
8. Cursor.angulo <- Cursor.angulo - Ï“*2/3 // DD
9. Dibujar_Koch(Cursor, Dist/3, N-1) // A
10. Cursor.angulo <- Cursor.angulo + Ï“/3 // I
11. Dibujar_Koch(Cursor, Dist/3, N-1) // A
12. Terminar

- FunciÃ³n que calcula el nÃºmero de unos necesarios para representar un nÃºmero en base 2 (en C)

```
int recursivaUnos (int numero){
int res=0;
if (numero/2 == 0){
if (n % 2 == 1) res++;
} else if (n/2 >0){
if (n % 2 == 1) res++; }
recursivaUnos(n/2);
}
return res;
}
```

- FunciÃ³n de Ackerman

```
int Ackermann(int p, int q)

{

if(p==0) return q+1;

if(q==0) return Ackermann(p-1,1);

return Ackermann(p-1,Ackermann(p,q-1));

}
```

- Funciones recursivas, una en C y otra en Pascal, que calculan el nÃºmero de combinaciones de m objetos en n.

```
int comb(int n, int m)
{
Â  if ((n == 0) || (n == m)) return 1;
Â  else return comb(n-1,m-1) + comb(n-1,m);
```

}

```
function comb(n, m : integer) : integer
begin
Â  if (n = 0) or (n = m) then comb := 1
Â  else comb := comb(n-1,m-1) + comb(n-1,m)
end
```

16-2-2007