Ontiveros Lara Claudia Sarahi. 17212167

Recursividad

Técnica de programación muy potente que puede ser usada en lugar de la iteración. Una función recursiva es una función que se llama a sí misma, ya sea directa o indirecta a través de otra función; es una alternativa diferente para implementar estructuras de repetición (ciclos). Los módulos se hacen llamadas recursivas.
Se puede usar en toda situación en la cual la solución pueda ser expresada como una secuencia de movimientos, pasos o transformaciones gobernadas por un conjunto de reglas no ambiguas.

¿En qué consiste la recursividad?

- En el cuerpo de sentencias del subalgoritmo se invoca al propio subalgoritmo para resolver “una versión más pequeña” del problema original.
- Habrá un caso (o varios) tan simple que pueda resolverse directamente sin necesidad de hacer otra llamada recursiva.

Ámbito de Aplicación:

- General
- Problemas cuya solución se puede hallar solucionando el mismo problema pero con un caso de menor tamaño.

Razones de uso:

- Problemas “casi” irresolubles con las estructuras iterativas.
- Soluciones elegantes.
- Soluciones más simples.

Condición necesaria: asignación dinámica de memoria.

Características.

Consta de una parte recursiva, otra iterativa o no recursiva y una condición de terminación. La parte recursiva y la condición de terminación siempre existen. En cambio la parte no recursiva puede coincidir con la condición de terminación.
Algo muy importante a tener en cuenta cuando usemos la recursividad es que es necesario asegurarnos que llega un momento en que no hacemos más llamadas recursivas. Si no se cumple esta condición el programa no parará nunca.

Componentes:

- Caso base. Es el resultado más simple, lo que conoce la función.
- Paso de recursión. Problema poco menos complejo que el original. También puede incluir la palabra reservada return.

Ventaja.

La principal ventaja es la simplicidad de comprensión y su gran potencia, favoreciendo la resolución de problemas de manera natural, sencilla y elegante; y facilidad para comprobar y convencerse de que la solución del problema es correcta.

Factible de utilizar recursividad:

- Para simplificar el código.
- Cuando la estructura de datos es recursiva

Inconveniente.

El principal inconveniente es la ineficiencia tanto en tiempo como en memoria, dado que para permitir su uso es necesario transformar el programa recursivo en otro iterativo, que utiliza bucles y pilas para almacenar las variables.

No factible utilizar recursividad:

- Cuando los métodos usen arreglos largos.
- Cuando el método cambia de manera impredecible de campos.
- Cuando las iteraciones sean la mejor opción.

Ejemplos:

1)

La suma de los elementos del arreglo:

```
int suma(int *vector, int fin)
{
int result;
if(fin==0)
result=vector[0];
else
result=vector[fin]+suma(vector,fin-1);
return result;
}
```

2)

Programar un algoritmo recursivo que permita sumar los dígitos de un número.Ejemplo: Entrada:123 Resultado:6

Solución:

```
int sumar_dig (int n)
{
if (n == 0) {      //caso base
return n;
```

```
}
else {
return sumar_dig (n / 10) + (n % 10);
}
}
```

3)

Programar un algoritmo recursivo que determine si un número es positivo/negativo.
Solución:

```
public boolean positivo(int n){
   if(n<0) return true;
   else return negativo(n);
  }
  public boolean negativo(int n){
   if(n>0) return false;
   else return  positivo(n);
  }
```

4)
Implementación en Pascal:

```
function fibonacci (n:integer): integer;
begin
if n = 0 then
fibonacci := 0
else
if n = 1 then fibonacci:= 1
else fibonacci := fibonacci (n-1)+fibonacci (n-2)
end; {fibonacci}
```

5)
Programar un algoritmo recursivo que calcule el factorial de un número.
Solución:

```
int factorial(int n)
{
if(n==0)
{
return 1; //Caso Base
}
else {
return n * factorial(n-1);  //Fórmula Recursiva
}
}
```

6)
Programar un algoritmo recursivo que calcule un número de la serie fibonacci.
Solución:

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

7)
Programar un algoritmo recursivo que permita invertir un número.Ejemplo: Entrada:123 Salida:321
Solución:

```
int invertir (int n)
{
 if (n < 10) {         //caso base
  return n;
 }
 else {
  return (n % 10) + invertir (n / 10) * 10;
 }
}
```

Referencias:

Tenenbaum Aaron. (1983). Estructuras de datos en Pascal. Prentice Hall.

Joyanes Aguilar, L. (1988). Fundamentos de programación. (Ed. rev.). Madrid, España: McGraw-Hill.

Joyanes Aguilar, Luis (1996) Fundamentos de programación, Algoritmos y Estructura de datos. McGraw-Hill, México.

Deitel & Deitel (2001) C++ Como programar en C/C++. Prentice Hall

Kerrighan y Ritchie “El lenguaje de programación”. Prentice Hall

Gottfried, Byron (1999) “Programación en C” McGrawHill, México.

Dale/Orshalick. (1986). Pascal. Ed McGraw Hill.

Nell Dale, Susan C. Lilly. (1989). Pascal y Estructuras de Datos. McGraw Hill.