***TP Nº9: Recursividad***

***Ejercicio 4***

Se puede apreciar fácilmente que al invocar a cualquiera de las dos funciones se entra en una recursión infinita.

***Ejercicio 5***

El error aquí es más sutil que en el ejercicio anterior. Si se invoca a la función esPar con un valor par o a la función esImpar con un valor impar, ambas funcionan en forma correcta. Por ejemplo:

esPar(4) --> esImpar(3) --> esPar(2) --> esImpar(1) --> 1

esImpar(1) retorna 1, lo que hace que esPar(2) también retorne 1, etc.

El problema surge cuando se invoca a esPar con un valor impar o esImpar con un valor par. Ejemplo:

esPar(3) --> esImpar(2) --> esPar(1) --> esImpar(0) --> esPar(-1)

el valor -1 es casteado a unsigned int para la función esPar, por lo que el valor que recibirá será un "entero muy grande" (en Linux será el 4294967295 ) y seguirá invocando recursivamente, apilando llamadas en el stack. El programa aborta en tiempo de ejecución por "stack overflow" o "segmentation fault".

***Ejercicio 9***

#define EPSILON 0.000000001

double

raizNR(double valorAnterior, int iter, double x)

{

*/\* Contemplamos el caso de querer calcular la raíz de cero \*/*

if (fabs(valorAnterior) < EPSILON)

return 0;

if (iter == 0)

return valorAnterior;

valorAnterior = (valorAnterior + (x / valorAnterior)) / 2;

return raizNR(valorAnterior, iter-1, x);

}

***Ejercicio 10***

void

invierte(const char \*origen, char \*destino)

{

if (\*origen)

{

\*(destino+strlen(origen)-1) = \*origen;

invierte(origen+1,destino);

}

}

Otra posibilidad, pero no respetaría el enunciado, es que invierte sea una función “wrapper” que calcule la longitud de origen e invoque a una función auxiliar que reciba ambos strings y la longitud.

Una tercera posibilidad es que la función sea de tipo entero y posponga la copia hasta encontrar el final, cuando la longitud de origen sea cero. A partir de ahí retorna siempre la longitud más uno y copia el carácter en la posición que le corresponde.

int invierte(const char \*origen, char \*destino)  
{

int n;  
    if (\*origen == 0) {  
         return 0;  
    }  
    n = invierte(origen +1, destino);  
    destino[n] = \*origen;  
    return n +1;  
}

***Ejercicio 12***

int

balance(const char \* texto)

{

if (\*texto == 0)

return 0;

switch (\*texto)

{

case '(': return -1 + balance (texto+1);

case ')': return 1 + balance (texto+1);

default: return balance (texto+1);

}

}

¿Es correcta la siguiente solución?

int

balance(const char \* texto)

{

if (\*texto == 0)

return 0;

switch (\*texto)

{

case '(': return -1 + balance (texto++);

case ')': return 1 + balance (texto++);

default: return balance (texto++);

}

}

Otra forma de escribir el mismo algoritmo.

int

balance(const char \* texto)

{

if (\*texto == 0)

return 0;

int count=0;

switch (\*texto)

{

case '(': count = -1; break;

case ')': count = 1; break;

}

return count + balance (texto+1);

}

***Ejercicio 15***

int

orden(const int v[])

{

*// No es un caso base, es validación. Sería más eficiente hacerlo*

*// en una función “wrapper” pero no cumpliría con la consigna*

if(v[0] == -1 || v[1] == -1)

return 0;

*// Este es el caso base. Cuando hay dos elementos determino si*

*// están ordenados o no*

if (v[2] == -1 ) {

if(v[0] < v[1])

return 1;

if(v[0] > v[1])

return -1;

return 0; *// son iguales, no está ordenado*

}

int ordenResto = orden(v+1);

*// Si el resto del vector no está ordenado…*

if ( ordenResto == 0 )

return 0;

*// Si el resto está ordenado ascendente, el elemento actual*

*// debe ser menor al primer elemento del resto del vector*

if ( ordenResto == 1 )

return v[0] < v[1];

*// El resto del vector está ordenado descendentemente*

if ( v[0] > v[1])

return -1;

return 0;

}

¿Se recorre sí o sí todo el vector?

En caso afirmativo: ¿se podría modificar para que en ciertos casos detecte que no está ordenado y evitar recorrerlo hasta el final? **Nota:** respetar el algoritmo recursivo.