Unidad 2
Recursividad y Análisis de Algoritmos

2.1 Definición
2.2 Procedimientos Recursivos
2.3 Ejemplos de Casos Recursivos
2.4 Análisis de Algoritmos.
- 2.4.1 Complejidad en el tiempo.
- 2.4.2 Complejidad en el Espacio.
- 2.4.3 Eficiencia de Algoritmos.

2.1 Definición de Recursividad

Antes de hablar de recursividad, es importante revisar algunos ejemplos de programación modular en los que se hacen llamadas a ejecución a un método desde el interior de otro método.

- Las llamadas a ejecución de un subprograma desde otro ayudan a clarificar el código y contribuyen a mayores niveles de abstracción.

2.1 Definición de Recursividad

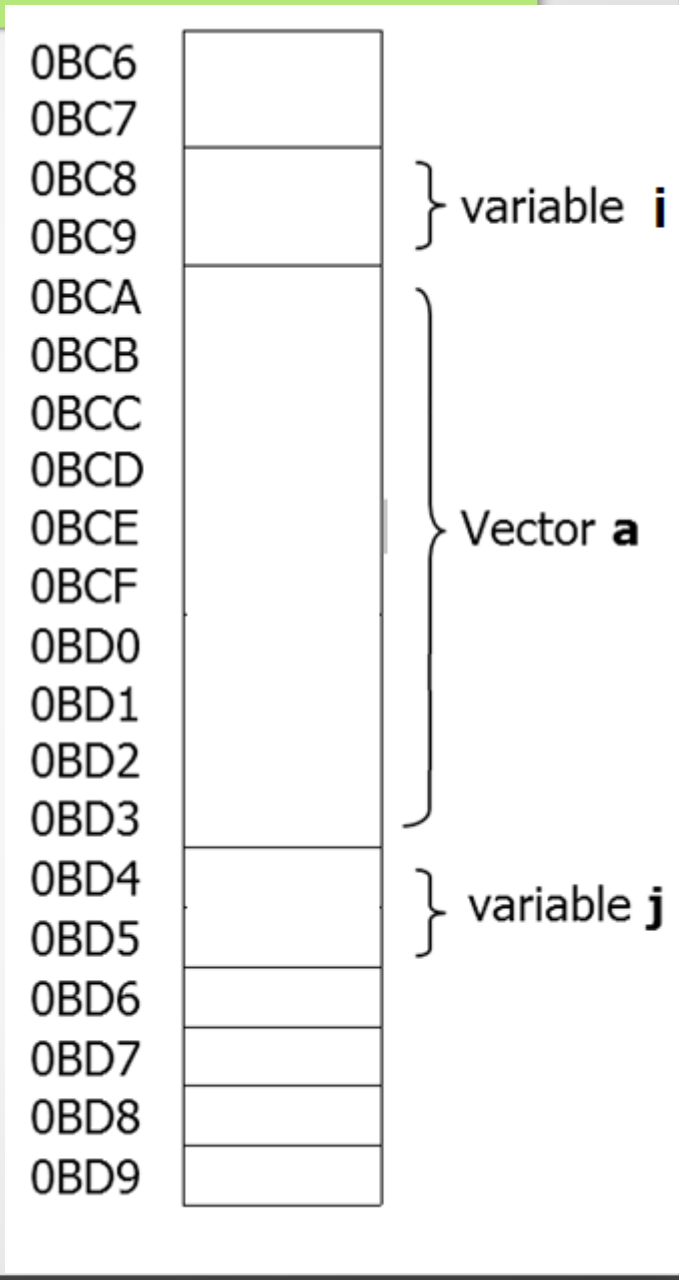
Ejemplos de programación modular:

2.1 Definición de Recursividad

En la página siguiente se demuestra el problema que se produce cuando se intenta guardar un dato fuera del tamaño máximo de un vector.

El pequeño programa de la página siguiente hace patente la necesidad de impedir de alguna forma se intente guardar datos fuera del tamaño de un vector.

```
#include <iostream>
using namespace std;
int main()
{
    int i;
    int a[5];
    int j;
    i = 10;
    j = 20;
    a[6] = 50; // las posiciones válidas son 0..4
    cout << "Instrucciones ejecutadas:\n";
    cout << "i=10; j=20; a[6]=50;\n";
    cout << "Valores conservados:\n";
    cout << "i=" << i << " " << "j=" << j << endl;
    return 0;
}
```



2.1 Definición de Recursividad

Primer ejemplo:

Se modificó la Clase ***Vec*** que escribimos anteriormente para que impida añadir datos al vector cuando se ha alcanzado su capacidad máxima.

Enseguida se muestran el código de la Clase Vec (con un nuevo método llamado ***EstaLLeno***) y el programa de aplicación.

Aplicación Lecturas de presión

Observe que incluimos otro archivo de cabecera

```
#include <iostream>
#include "ClaseVec3.h"
using namespace std;
int main() {
        Vec Lecturas;
        int nuevaLectura;
        cout<<"Escribe la lectura de compresion (0->200): ";cin >> nuevaLectura;
        while (nuevaLectura!=0) {
                Lecturas.Anadir(nuevaLectura);
                cout<<"Escribe la lectura de compresion (0->200): ";cin >>
                nuevaLectura;
        }
        cout << "Primer Valor registrado : " << Lecturas.Traer(0) << endl;
        cout << "Ultimo Valor registrado : " << Lecturas.Traer(Lecturas.Tamanio()-1)
                 << endl;
        cout << "Numero de Datos         : " << Lecturas.Tamanio() << endl;
        float suma=0;
        for(int i=0;i<Lecturas.Tamanio();i++)
                suma += Lecturas.Traer(i);
        float promedio=suma/Lecturas.Tamanio();
        cout << "Promedio                : " << promedio << endl;
        int mayores=0;
        for(int i=0;i<Lecturas.Tamanio();i++)
                if (Lecturas.Traer(i)>promedio)
                        mayores++;
        cout << "Numero de Datos mayores al Promedio: " << mayores << endl;
        return 0;
}
```

```
#define MaxVector 10
class Vec{
      private:
        int contador;
        int Datos[MaxVector];
      public:
        Vec(); void Anadir(int); void Insertar(int,int);
        void Eliminar(int); int Traer(int); int Tamanio(); bool EstaLLeno();
    };
Vec::Vec(){
   contador=0;
};
void Vec::Anadir(int nuevoDato){
    if (not this->EstaLLeno()) {
            Datos[contador]=nuevoDato;
            contador++;
    }
}

void Vec::Insertar(int nuevoDato,int posicion){
    if (not this->EstaLLeno()) {
            for (int i=contador-1;i>=posicion;i--)
                Datos[i+1] = Datos[i];
            Datos[posicion]=nuevoDato;
            contador++;
    }
}
void Vec::Eliminar(int posicion){
    for (int i=posicion+1;i<=contador-1;i++)
            Datos[i-1] = Datos[i];
    contador--;
}
int Vec::Traer(int posicion){
    return Datos[posicion];
}
int Vec::Tamanio(){
    return contador;
}
bool Vec::EstaLLeno() {
    bool result=false;
    if (contador==MaxVector) {
            std::cout << "*No hay espacio para un nuevo valor*\n";
            result=true;
    }
    return result;
}
```

Se definió el vector de un tamaño pequeño para observar fácilmente el comportamiento cuando ya no haya espacio

Archivo de cabecera ClaseVec3.h

2.1 Definición de Recursividad

Segundo ejemplo:

Programa en el que se observan llamadas a ejecución *en cascada* de varios procedimientos para mostrar que al llamar a ejecución a un procedimiento desde el interior de otro, el primero continuará hasta que el segundo termine, pero si un segundo llama a un tercero, el segundo continuará hasta que el tercero termine y así sucesivamente.

```
#include <iostream>
using namespace std;

void ImprimeD() {
       printf("D");
       printf("4");  };

void ImprimeCD() {
       printf("C");
       ImprimeD();
       printf("3");  };

void ImprimeBCD() {
       printf("B");
       ImprimeCD();
       printf("2");  };

void ImprimeABCD() {
       printf("A");
       ImprimeBCD();
       printf("1");  };

int main() {
     system("cls");
     ImprimeABCD();
}
```

2.1 Definición de Recursividad

Definición y ejemplos de Recursividad:

2.1 Definición de Recursividad

La Recursividad es una técnica de programación muy poderosa usada ampliamente para solucionar problemas.

Se logra mediante la definición del **problema en términos de una forma más simple del problema mismo**.

2.1 Definición de Recursividad

Antepasado de una persona

Definición Iterativa:

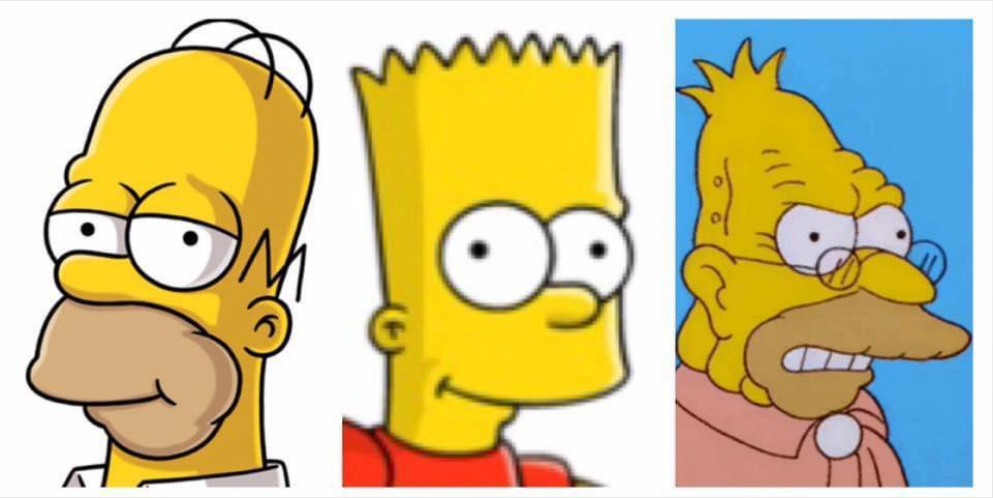
Antepasado de una persona es su padre, abuelo, bisabuelo, tatarabuelo, tátara-tatarabuelo, y así sucesivamente hasta el primer humano *(observe que aquí hay un ciclo explícito).*

Definición Recursiva:

Antepasado de una persona es su padre o el antepasado de su padre.

Ejemplo: Antepasados de Bart Simpson
Definición recursiva:
Antepasado = Padre o antepasado del padre.

1. Homero:
   - Es padre de Bart por lo tanto es su antepasado.
2. Abuelo Simpson:
   - Es el padre de Homero por lo tanto es su antepasado. De acuerdo a la definición recursiva, al ser antepasado de Homero también lo es de Bart.
   - Y también de acuerdo a la definición recursiva, el padre de Abuelo Simpson será antepasado de Homero y en consecuencia también de Bart.
   - Y así hasta el principio.



2.1 Definición de Recursividad

Números Naturales

aquí hay ciclo implícito

Definición Iterativa:

Un número natural es un entero entre **cero** e **infinito**.

0 1 2 3 4 5 6 7 8 9 10 11 12 ...

Definición Recursiva:

Un número natural **es cero** o el **siguiente entero** a un **número natural**.

2.1 Definición de Recursividad

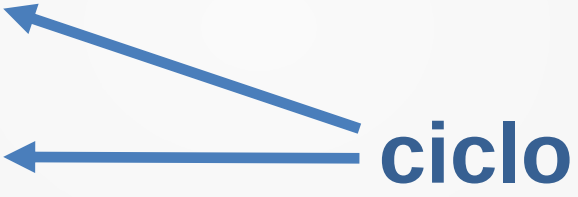
Factorial de n

Definición Iterativa:

El factorial de n es el producto de todos números naturales mayores a 0 y menores o iguales a n. Además, el factorial de 0 es 1.

Ejemplo para ***n=5***

5! = 1 x 2 x 3 x 4 x 5



Definición Recursiva:

$$n! = n \cdot (n-1)! \ si \ n > 0;$$

$$n! = 1 \ si \ n = 0;$$

Ejemplo:

$$5! = 5 * 4!$$

2.2 Procedimientos Recursivos.

Los procedimientos recursivos deben contar con tres propiedades:

1.- Definición recursiva.

El concepto que se define es utilizado en la propia definición.

2.- Al menos un Caso Base.

Un escenario final que detiene la recursividad produciendo un resultado. En algunos problemas, el caso base es el escenario inicial y es el punto de partida para la recursividad.

3.- Definición con reducción hacia el Caso Base.

Una regla o conjunto de reglas que dirige cualquier caso hacia el Caso Base. En algunos problemas esas reglas alejan los casos generales del caso base.

2.2 Procedimientos Recursivos.

```
#include <iostream>
using namespace std;
double factorial(char x);
int main()
{
    printf("El factorial de 0 es %.0f\n", factorial(0));
    printf("El factorial de 5 es %.0f\n", factorial(5));
    printf("El factorial de 7 es %.0f\n", factorial(7));
    char n;
    printf("A que número deseas obtener el factorial? ");
    scanf("%d",&n);
    printf("El factorial de %d es %.0f\n", n, factorial(n));
    system("pause");
    return 0;
}
```

```
double factorial(char x)
{
      if (x==0)
         return 1;
      else
         return x*factorial(x-1);
}
```

Ejercicio:

Ejecute el programa anterior en modo depuración e inspeccione los valores que va tomando la variable x durante las sucesivas ejecuciones de ***factorial*()**.

Si por alguna razón no puede hacerlo mediante depuración, puede hacer las siguientes modificaciones a la función factorial().

```
double factorial(char x)
{
 double temp;
 if (x==0)
   return 1;
 else {
   cout << "x -> " << (int)x << endl;system("pause");
   temp = x*factorial(x-1);
   cout << "x <- " << (int)x << endl;system("pause");
   return temp;
   }
}
```

2.2 Procedimientos Recursivos.

Ejercicio:

Reemplace el código de la función factorial() por un programa iterativo y verifique si el funcionamiento es el mismo.

```
double factorial(char x)
{
     ---------------------------
     ---------------------------
     ---------------------------
}
```

2.2 Procedimientos Recursivos.

Serie de Fibonacci:

Cada elemento es la suma de los dos elementos anteriores de la misma serie. **Cero** y **uno** son los primeros 2 elementos.

Posición Ordinal	1	2	3	4	5	6	7	8	9	10	11	12	
Serie	0	1	1	2	3	5	8	13	21	34	55	89	….

1.- Definición recursiva. $Fibonacci_6 = Fibonacci_5 + Fibonacci_4$

2.- Casos Base. $Fibonacci_1 = 0$; $Fibonacci_2 = 1$

3.- Reducción hacia el Caso Base.

$$Fibonacci_i = Fibonacci_{i-1} + Fibonacci_{i-2}$$

2.2 Procedimientos Recursivos.

```
#include <iostream>
using namespace std;
double fibo(int);
int main()
{
    int n;
    cout << "Elemento Ordinal de la serie de Fibonacci? 1->N ";cin >> n;
    cout << "Valor: " << fibo(n) << endl;
    return 0;
}
```

Ejecute el programa para diversos valores (entre ellos 50 por ejemplo) para observar el tiempo de ejecución

```
double fibo(int n){
   if (n==1)
      return 0;
   if (n==2)
      return 1;
   else
      return fibo(n-1)+fibo(n-2);
}
```

2 2 1 3 1 3

2.3 Ejemplos de Casos Recursivos.

Los programas recursivos tienden a ser menos eficientes que los iterativos, pero ganan en claridad y simplicidad de código.

Cuando se diseñan soluciones y se desea usar la recursividad, hay que cuidar ciertos casos como el de la serie de Fibonacci

```
double fibo(int posicion)
{
if ((posicion==1)or(posicion==2))
    return posicion-1;
else
    return fibo(posicion-1)+fibo(posicion-2);
}
```

2.3 Ejemplos de Casos Recursivos.

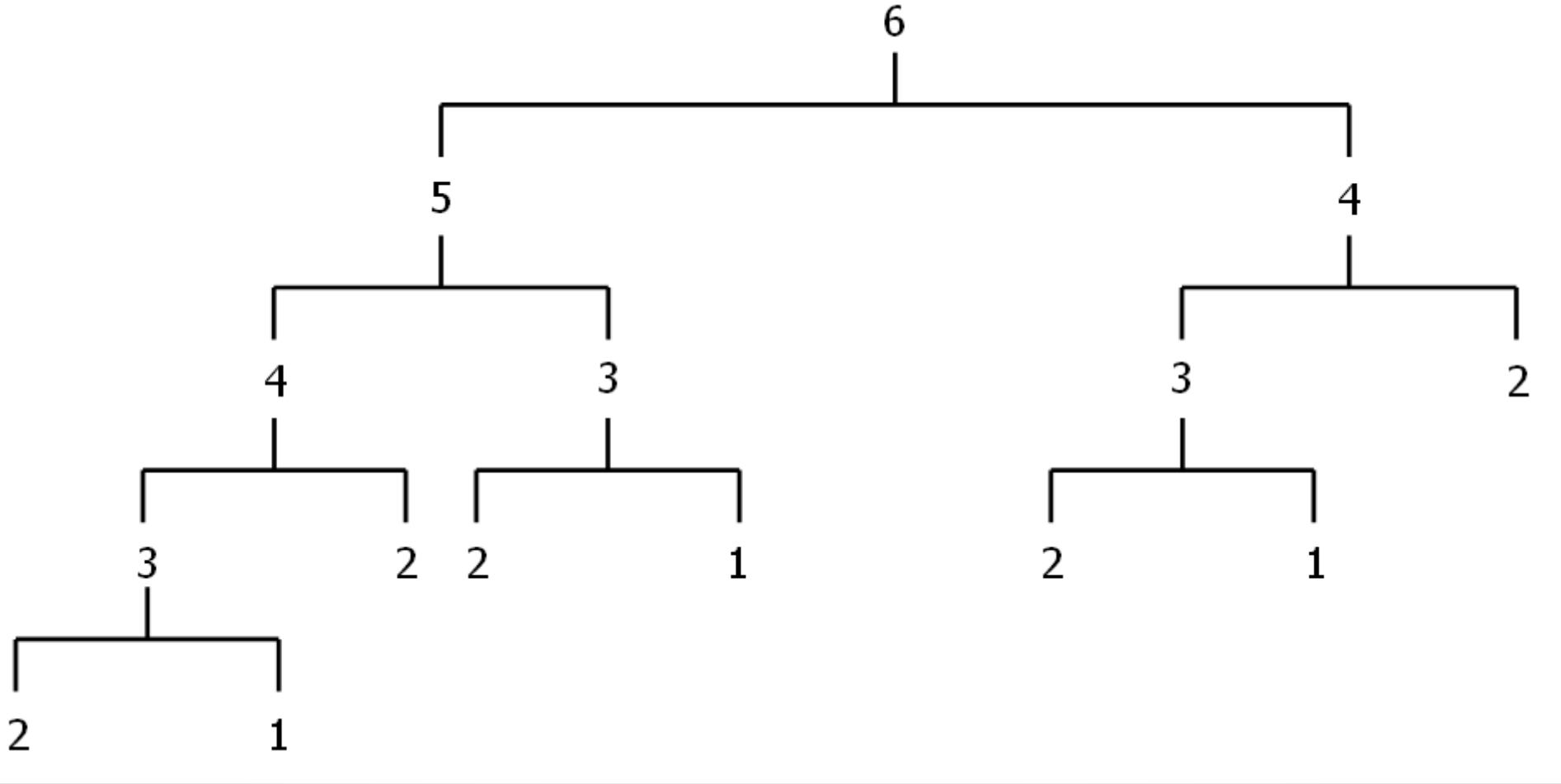
Este procedimiento cumple con las reglas para un procedimiento recursivo, pero al llamarse en 2 ocasiones a ejecución, se vuelve sumamente ineficiente para tamaños de problema grandes

```
double fibo(int posicion)
{
if ((posicion==1)or(posicion==2))
   return posicion-1;
else
   return fibo(posicion-1)+fibo(posicion-2);
}
```

2 2 1 3 3

2.3 Ejemplos de Casos Recursivos.

La siguiente figura muestra cuantas veces se llama a ejecución la función **fibo(6)**



2.3 Ejemplos de Casos Recursivos.

```
#include <iostream>
#include <cmath>
using namespace std;
double fibo(int);
double cuenta=0;
int main()
{
    int pos;
    cout << "Elemento Ordinal de la serie de Fibonacci? 1->N ";cin >> pos;
    cout << "Valor: " << fibo(pos) << endl;
    cout << "Numero de Ejecuciones de fibo() " << cuenta << endl;
    cout << "Numero de Niveles equivalente " << log2(cuenta) << endl;
    return 0;
}

double fibo(int posicion)
{
cuenta++;
if ((posicion==1)or(posicion==2))
   return posicion-1;
else
   return fibo(posicion-1)+fibo(posicion-2);
}
```

Añada a este programa recursivo de Fibonacci las instrucciones resaltadas

Ejercicio

Reemplace la función fibo() con el código iterativo de abajo y verifique el funcionamiento para valores grandes de **n**.

```
double fibo(int posicion){
      double result;
      double ant1=0;
      double ant2=1;
      if ((posicion==1)or(posicion==2))
                result=posicion-1;
      else
                for(int i=3;i<=posicion;i++) {
            result=ant1+ant2;
            ant1=ant2;
            ant2=result;
            }
      return result;
}
```

2.3 Ejemplos de Casos Recursivos.

La forma de recursividad usada para resolver Fibonacci es llamada **Recursividad Múltiple** porque cada llamada a la función causa más de una llamada adicional.

Evite la Recursividad Múltiple ya que tiende a ser muy deficiente.

Se comprueba calculando un elemento de la serie de Fibonacci mayor a 50 por ejemplo. El programa iterativo es eficiente a diferencia del recursivo, por la recursividad múltiple.

La **Recursividad Lineal** es eficiente en cuanto al tiempo de ejecución porque cada llamada solo causa una más, como en la función **factorial()**.

2.3 Ejemplos de Casos Recursivos.

Hay otra clasificación que es no es tan importante desde el punto de vista de tiempo de ejecución:

Directa (también llamada Simple).

Es la que corresponde a nuestros ejemplos. Una función se llama a ejecución a sí misma.

Indirecta (también llamada Compleja o Mutua).

Una función se llama a ejecución a través de una o varias:
A→B→A o A→B→C→A.
Esta forma de recursividad da poca claridad en el código.

2.3 Ejemplos de Casos Recursivos.

```
// Ejemplo Recursividad Indirecta
#include <iostream>
using namespace std;

double factorial(char);
double ReduccionCasoBase(char);

int main(){
    printf("El factorial de 0 es %.0f\n", factorial(0));
    printf("El factorial de 5 es %.0f\n", factorial(5));
    printf("El factorial de 7 es %.0f\n", factorial(7));
    char n;
    printf("A que numero deseas obtener el factorial? ");
    scanf("%d",&n);
    printf("El factorial de %d es %.0f\n", n, factorial(n));
return 0;
}
double factorial(char x){
     if (x==0)
        return 1;
     else
        return ReduccionCasoBase(x);
}
double ReduccionCasoBase(char y){
         return y*factorial(y-1);
}
```

2.3 Ejemplos de Casos Recursivos.

- Hay que evitar el error de **recursividad infinita** pero **no se recomienda ninguna verificación en el programa recursivo**.
- Asegúrese, en el programa de aplicación desde donde llama a la función, que los parámetros estén dentro de los límites previstos.
- En los ejemplos que hicimos debe impedirse ejecutar **factorial** para números negativos. La verificación no debe hacerse en la función porque tendría que regresarse un valor y no hay resultado posible para factorial(-3) por ejemplo.
- Enseguida se muestra un ejemplo.

2.3 Ejemplos de Casos Recursivos.

```
// Como evitar la Recursividad Infinita
#include <iostream>
using namespace std;
double factorial(char x);

int main(){
    char n;
    do{
       printf("A que numero deseas obtener el factorial? ");
       scanf("%d",&n);
    }while (n<0);
    printf("El factorial de %d es %.0f\n", n, factorial(n));
    system("pause");
    return 0;
}

double factorial(char x){
     if (x==0)
        return 1;
     else
        return x*factorial(x-1);
}
```

2.3 Ejemplos de Casos Recursivos.

En las siguientes diapositivas finales, se muestran un par de otros ejemplos sencillos de recursividad lineal.

Sumatoria de los números naturales de 1 a N

```
#include <iostream>
using namespace std;
int sumatoria(int);
int main() {
    int n;
    cout << "Sumatoria hasta que numero natural? ";
    cin >> n;
    cout << "Sumatoria: " << sumatoria(n) << endl;
    return 0;
}

int sumatoria(int x){
     if (x==0)
        return 0;
     else
        return x+sumatoria(x-1);
}
```

Con este sencillo programa recursivo podemos comprobar la formula **n*(n+1)/2**

Sumatoria de los dígitos de un número entero

```
#include <iostream>
using namespace std;

int SumaDigitos(int);

int main() {
    int x;
    do{
        cout << "Escribe un numero entero (cero para terminar): ";
        cin >> x;
        cout << "La suma de sus digitos es: " << SumaDigitos(x) << endl;
    }
    while (x>0);
    return 0;
}

int SumaDigitos(int n) {
    if (n==0)
        return 0;
    else
        return n%10 + SumaDigitos(n/10);
}
```

Ejemplo
Entrada: 7692
Resultado: 24 (7+6+9+2)