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[6];
    int j;
    i = 10;
    j = 20;
    a[6] = 50; // las posiciones válidas son 0..5
    cout << "i = " << i << endl;
    cout << "j = " << j << endl;
    system("pause");
    return 0;
}
```

2.1 Definición de Recursividad

Primer ejemplo:

Se modificó el TDA VectorDinamico que hicimos anteriormente para que impida añadir datos al vector cuando se ha alcanzado su capacidad máxima.

Enseguida se muestra el código de la clase VectorDinamico en la que se incluye el uso de un método llamado EstaLLeno() que se llama a ejecución desde los métodos *anadir* e *insertar*.

2.1 Definición de Recursividad

```
#define MaxVector 25 // El tamaño mínimo funcional es 25 por una falla
en DevC++
class VectorDinamico{
      private:
               int numdatos;
               unsigned char vector[MaxVector];
      public:
               VectorDinamico();
               void Anadir(unsigned char);
               void Insertar(unsigned char,int);
               void Eliminar(int);
               unsigned char Get(int);
               void ModifTamanio(int);
               int  Tamanio();
               bool EstaLLeno();
    };

VectorDinamico::VectorDinamico() {
       numdatos=0;
};
```

2.1 Definición de Recursividad

```
void VectorDinamico::Anadir(unsigned char dato){
	if (not this->EstaLLeno()) {
		vector[numdatos]=dato;
		numdatos++;
	}
}

void VectorDinamico::Insertar(unsigned char dato,int pos) {
	if (not this->EstaLLeno()) {
		for (int i=numdatos-1;i>=pos;i--)
			vector[i+1] = vector[i];
		vector[pos]=dato;
		numdatos++;
	}
}

void VectorDinamico::Eliminar(int pos) {
	for (int i=pos+1;i<=numdatos-1;i++)
		vector[i-1] = vector[i];
	numdatos--;
}
```

2.1 Definición de Recursividad

```
unsigned char VectorDinamico::Get(int pos) {
	return vector[pos];
}

int VectorDinamico::Tamanio() {
	return numdatos;
}

void VectorDinamico::ModifTamanio(int nuevoTam) {
	numdatos = nuevoTam;
}

bool VectorDinamico::EstaLLeno() {
	bool result=false;
	if (numdatos==MaxVector) {
		std::cout << “*No hay espacio para un nuevo valor*\n";
		result=true;
	}
	return result;
}
```

2.1 Definición de Recursividad

Segundo ejemplo:

Programa en el que se observan llamadas a ejecución *en cascada* de varios procedimientos para ejemplificar que cuando desde el interior de un procedimiento se llama a ejecución a otro, el primero continuará hasta que el segundo termine.

```
#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:

Su padre, abuelo, bisabuelo, tatarabuelo, tátara-tatarabuelo, y así sucesivamente hasta el primer humano *(observe que aquí hay un ciclo explícito).*

Definición Recursiva:

Es su padre o el antepasado de su padre.

2.1 Definición de Recursividad

Dado el caso de:

- Carlos I, Carlos II, Carlos III, Carlos IV, Carlos V.

Antepasado de una persona es su padre o el *(padre o antepasado)* de su padre … o
Antepasado de una persona es su padre o el padre o *(padre o antepasado)* de su padre.

Carlos I es antepasado de Carlos V porque
Carlos IV es antepasado (padre) de Carlos V,
Carlos III es antepasado (padre) de Carlos IV,
Carlos II es antepasado (padre) de Carlos III y
Carlos I es antepasado (padre) de Carlos II

2.1 Definición de Recursividad

Números Naturales

Definición Iterativa:

Los enteros entre **cero** e **infinito**.

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

Definición Recursiva:

Cero o el **siguiente entero** a un **número natural**.

2.1 Definición de Recursividad

Factorial de n.

Definición Iterativa:

El producto de todos los enteros positivos menores o iguales a n. Como excepción, el factorial de 0 es 1.
Para n=5, 5! = 1 x 2 x 3 x 4 x 5

Definición Recursiva:

n por el factorial del número natural anterior. El factorial de **cero** es **1**.
Para n=5, 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 no usa recursividad para producir un resultado.

3.- Definición con reducción hacia el Caso Base.

Una regla o conjunto de reglas que reduce cualquier caso hacia el 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 de Fibonacci. **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. $Fibonacci_1 = 0$; $Fibonacci_2 = 1$

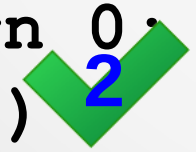
3.- Reducción hacia el Caso Base.

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

2.2 Procedimientos Recursivos.

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

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



2.2 Procedimientos Recursivos.

Ejercicio:

Reemplace el código de la función fibo() por un programa iterativo y verifique si el funcionamiento es el mismo para valores de **n** un poco mayores.

```
int fibo(int n)
{
     ---------------------------
     ---------------------------
     ---------------------------
}
```

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

```
int 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.

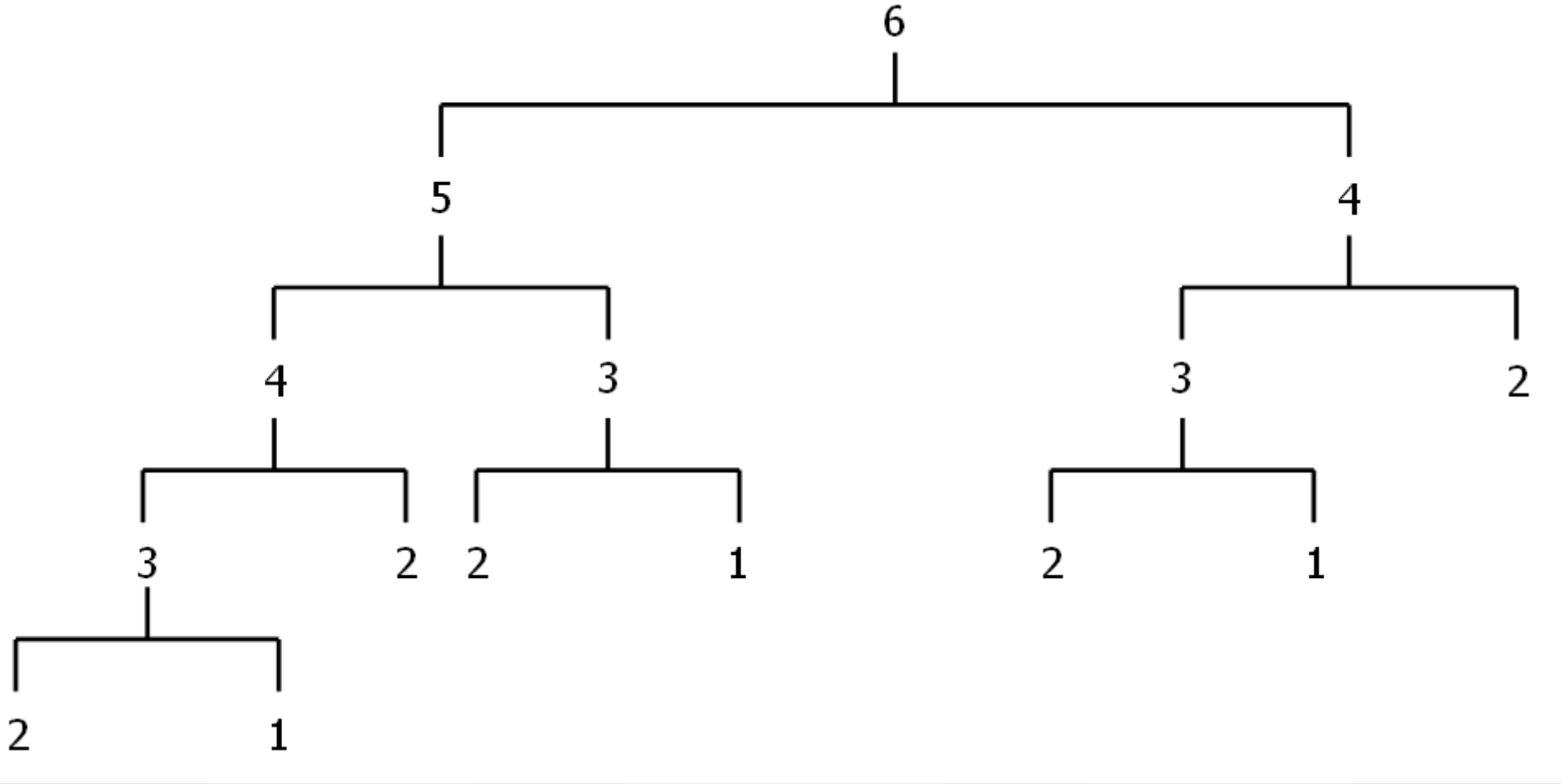
Este procedimiento cumple con las reglas para un procedimiento recursivo, pero al llamarse en 2 ocasiones a ejecución, se vuelve sumamente ineficiente para tamaños de problema grandes

```
int 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.

Añada al programa recursivo de Fibonacci el código resaltado

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

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

}
```

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 20 o 30 por ejemplo. Escriba el programa iterativo para comprobarlo.

La **Recursividad Lineal** no es tan deficiente como la múltiple, ya que se le llama de esa manera porque cada llamada solo causa una llamada más. Por ejemplo **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 (tambien llamada Compleja o Mutua).

Una función se llama a ejecución a través de otra, en forma directa o indirecta. Es decir, A→ B→A o A→B→C→A causa una llamada más.
Esta causa 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.

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.

Escriba un programa recursivo para encontrar la sumatoria de los números naturales de 1 hasta N.

Definición del problema:

La sumatoria de los números naturales hasta N es igual a N más la sumatoria de los números naturales hasta N-1