Universidad Autónoma del Estado de México

Centro Universitario UAEM Texcoco

Departamento de Ciencias Aplicadas.

Ingeniería en Computación.

Programación avanzada.

Unidad de competencia I: “Paradigmas de programación modular y recursiva”

Presenta:

M. en C. C. J. Jair Vázquez Palma.

Programación avanzada

Objetivos de la Unidad de la Unidad de Aprendizaje

- Analizar y diseñar sistemas de información.
- Utilizar eficazmente los lenguajes de programación.
- Responder eficazmente a nuevas situaciones informáticas.
- Analizar soluciones del entorno y problemas propios de ser tratados mediante sistemas computacionales
- Aplicar los conocimientos en la práctica
- Desarrollar la habilidad análisis y síntesis de información

Unidad I: Paradigmas de programación modular y recursiva.

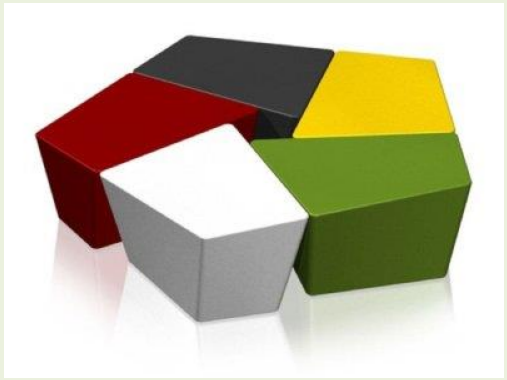
Contenido:

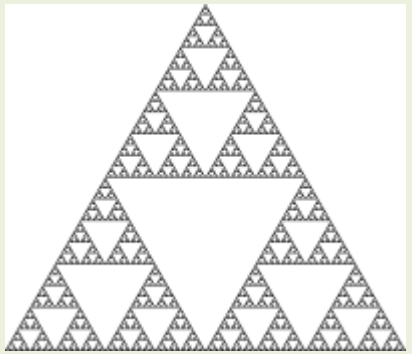
- Programación modular.
- Programación recursiva.
    - Tipos de recursividad.

Objetivos de la Unidad de Competencia I

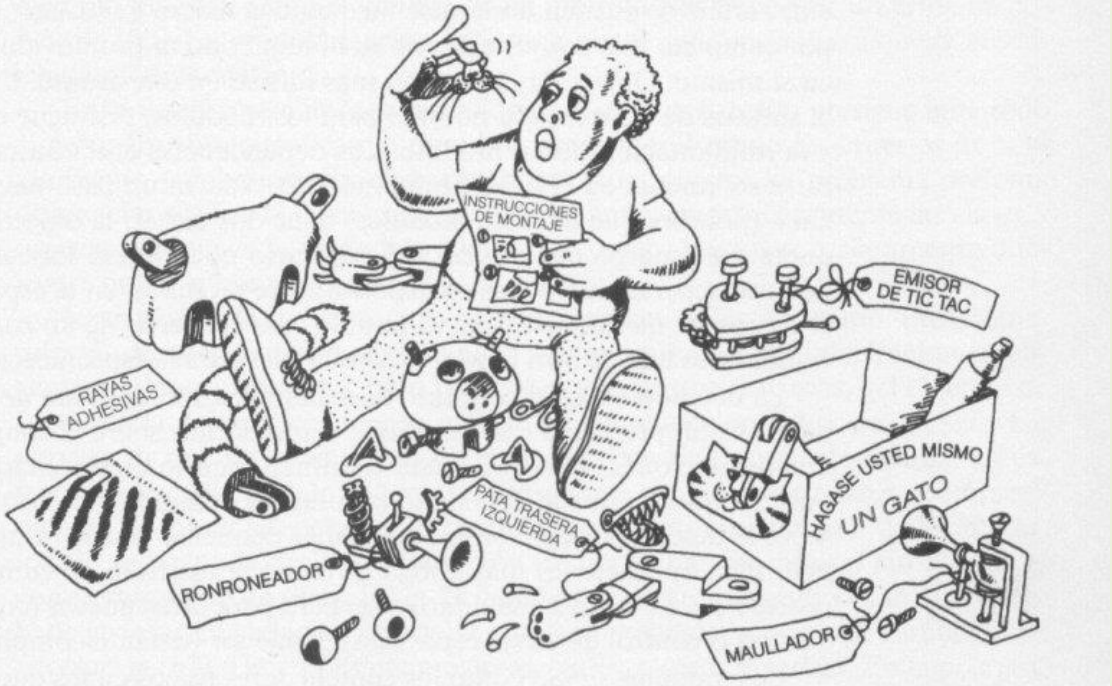
- Desarrollar algoritmos bajo los paradigmas de programación modular, programación recursiva.
- Capacidad de abstracción en la implementación de programas bajo un determinado lenguaje de programación.
- Manejo de los distintos tipo de recursividad.

PARADIGMA DE PROGRAMACION MODULAR Y RECURSIVA





MODULARIDAD.



La modularidad empaqueta las abstracciones en unidades discretas.

MODULARIDAD

“La modularidad es la descomposición de un sistema en un conjunto de módulos cohesivos y débilmente acoplados.”

- La descomposición de un sistema en componentes individuales ayuda a manejar la complejidad.

- Sin embargo, una descomposición desordenada puede producir un efecto contrario que se puede contrarrestar reagrupando los componentes en módulos o paquetes**.**

MODULARIDAD

- Cada módulo debe contener componentes con características afines, de tal manera que faciliten la producción de la arquitectura física de un sistema.

- En el diseño orientado a objetos, la modularización consiste en decidir dónde colocar físicamente las clases y objetos a partir de la estructura lógica del diseño.

- La estructura de cada módulo debe ser lo bastante simple como para ser comprendida en su totalidad, haciendo posible cambiar su implementación sin saber nada de la implementación de los demás módulos y sin afectar el comportamiento de éstos.

MÉTODOS.

Los métodos permiten especificar el comportamiento de una clase. La implementación sencilla de un método consta de dos partes, una *declaración* y un *cuerpo*.

```
modificadorDeAcceso tipoRetorno nombreMetodo( [listaDeArgumentos] )
{
        cuerpoMetodo
}
```

Programas y Sentencias

- Un programa de computadora es un sistema con componentes e interrelaciones. Intentaremos determinar cuáles son dichos componentes y como se relacionan.
- En primer lugar definiremos los conceptos de programa y programa de computadora.
- Un *programa* puede ser definido como "una secuencia precisa y ordenada de instrucciones y grupos de instrucciones, las cuales, en su total, definen, describen, o caracterizan la realización de alguna tarea".
- Un *programa de computadora* es simplemente un programa el cual, posiblemente a través de una transformación, indica a la computadora como realizar una tarea.

Programas y Sentencias

- En el nivel más elemental, observamos que un programa de computadora está compuesto de sentencias o instrucciones. Estas instrucciones están ordenadas en una secuencia. Podemos por lo tanto identificar a las instrucciones como componentes y a la secuencia como una relación. Esta es la visión clásica o "algorítmica" de un programa.
- De esta visión tenemos como consecuencia, que el esfuerzo en el desarrollo de un programa se enfatiza en encontrar un método de solución y su transcripción sentencia a sentencia.
- Para los propósitos de nuestro estudio, consideraremos que una sentencia es una línea de código que el programador escribe.

La Lingüística de la Modularidad.

Un módulo es una secuencia lexicográficamente contigua de sentencias, encerrada entre elementos de frontera, y que poseen un identificador del conjunto de dichas sentencias.

- Dicho de otra manera, un módulo es un grupo de sentencias contiguas que poseen un identificador simple por el cual son referenciadas.

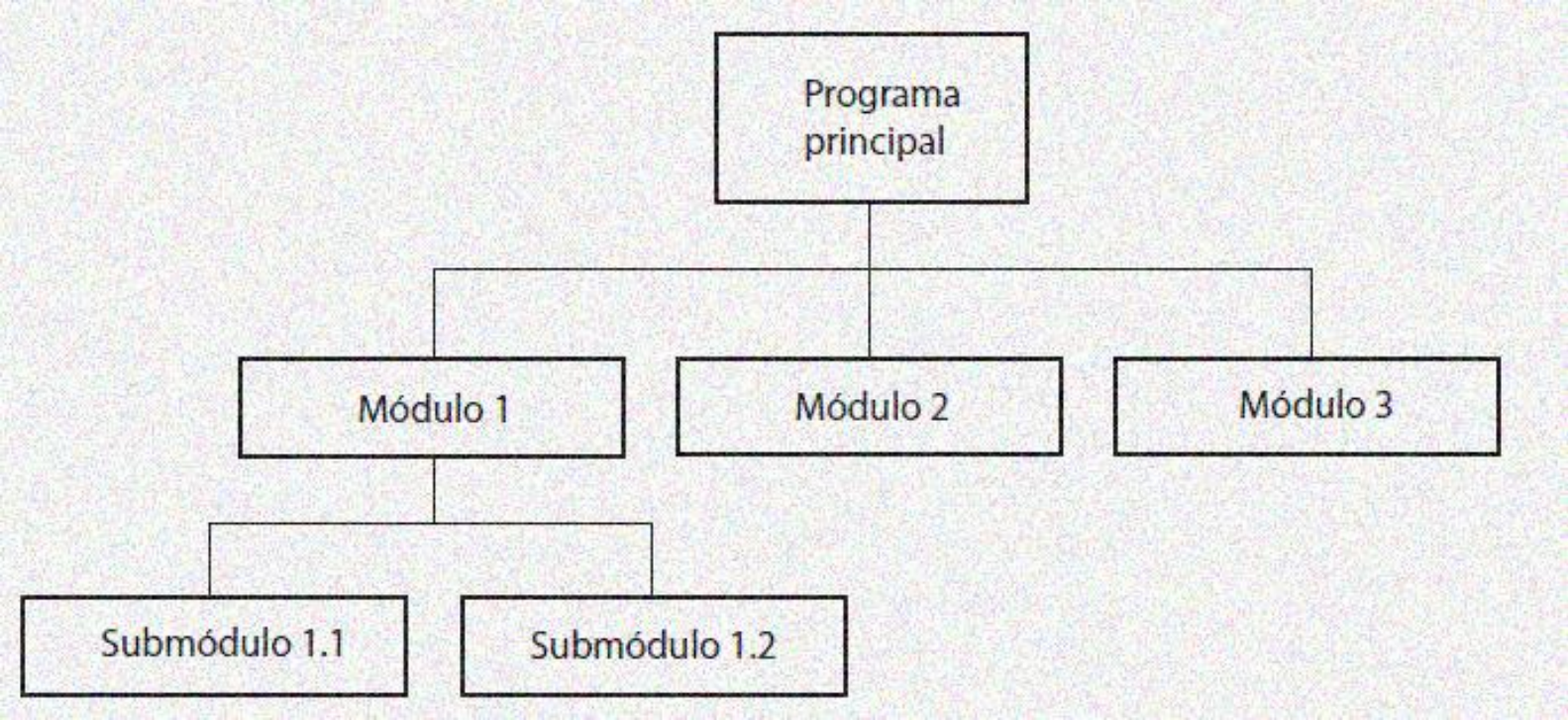
La Lingüística de la Modularidad.

- Esta definición es general y dentro de la misma podemos encontrar implementaciones particulares de lenguajes específicos como ser: "párrafos", "secciones", y "subprogramas" de COBOL, "funciones" de C, "procedimientos" de Pascal, etc.
- Un lenguaje de programación incluye un determinado tipo de módulo, solo si implementa construcciones lingüísticas específicas que realizan las características de definición y activación de dichos módulos.

Las ventajas de la programación modular son las siguientes:

1. Facilita el diseño descendente.
2. Se simplifica un algoritmo complejo.
3. Cada módulo se puede elaborar de manera independiente, lo que permite trabajar simultáneamente a varios programadores y con ello disminuir el tiempo de elaboración del algoritmo.
4. La depuración se lleva a cabo en cada módulo.
5. El mantenimiento es más sencillo.
6. Creación de bibliotecas con módulos específicos (se pueden utilizar en otros programas).

Los módulos o subprogramas los llamaremos *funciones*, ya que todas las características descritas son propias del lenguaje C



Programa principal o función *main()*

- El papel más importante del *programa principal (main())* es coordinar a las otras funciones mediante llamadas o invocaciones, es la primer función llamada por el compilador.

Función

- Es un subprograma que realiza una tarea específica que puede o no recibir valores (parámetros).
- En C podemos devolver cualquier tipo de datos escalares (puntero, tipo numérico y el tipo carácter o en su caso regresar un valor nulo que llamaremos *nada* o *ninguno*). Asimismo, no se pueden devolver arreglos ni estructuras.

Ámbito de las variables.

- ***Variable local.*** Variable declarada en una determina función, sólo se encuentra disponible durante el funcionamiento de la misma, es decir está en memoria cuando dicha función está activa.

- ***Variable global.*** Variable declarada fuera de cualquier función y que puede ser utilizada por las funciones que se encuentran después de dicha declaración; ***por lo tanto si la declaramos junto a las librerías, la podrá utilizar todo el programa. Esta características es propia del lenguaje C.***

Llamada o invocación de una función.

- Para llamar una función es necesario escribir el nombre o identificador de la función, con paréntesis vacíos si hablamos de funciones sin paso de parámetros y con información en caso de ser funciones con paso de parámetros.

Ubicación de una función en un programa.

- Las funciones pueden ir antes o después del programa principal (función principal). Una función puede que regrese o no regrese nada (*void*).
- Existen dos tipos de funciones: Sin paso de parámetros y con paso de parámetros.

Programación estructurada y modular.

```c
#include <stdio.h>
main()
{
    int matriza[3][2];
    int i, j;

    for(i=0;i<3;i++)
    {
        for(j=0;j<2;j++)
        {
            printf("Ingrese el elemento %d - %d:",i,j);
            scanf("%d",&matriza[i][j]);
        }
            printf("\n");
    }
    for(i=0;i<3;i++)
    {
        for(j=0;j<2;j++)
        {
            printf("%d\t",matriza[i][j]);
        }
            printf("\n");
    }
    getch();
}
```

```c
#include<stdio.h>
#include<conio.h>
void llenar_matriz(int x[][20], int tam)
{
   int i,j;
   for (i=0; i<tam; i++)
      for (j=0; j<tam; j++)
        scanf ("%d",&x[i][j]);
}
void imprimir_matriz(int x[][20],int tam)

{
   int i,j;
   for (i=0;i<tam; i++)
   {
      for (j=0; j<tam; j++)
          printf("%d ",x[i][j]);
   printf("\n");
   }
}
```

Paso de parámetros en una función.

- En C todos los parámetros se pasan *“por valor”*, es decir, en cada llamada a la función se genera una *copia* de los valores de los parámetros *actuales*, que se almacenan en variables temporales en la pila mientras dure la ejecución de la función.

- Parámetro por “referencia” su *dirección*, lo cual se realiza mediante un puntero que señale a esa dirección; a estos parámetros los llamamos *por variable* o *por referencia*; en este tipo de parámetros los datos salen modificados.

El concepto de Cajas Negras

Una caja negra es un sistema (o un componente) con entradas conocidas, salidas conocidas, y generalmente transformaciones conocidas, pero del cual no se conoce el contenido en su interior.

En la vida diaria existe innumerable cantidad de ejemplos de uso cotidiano: una radio, un televisor, un automóvil, son cajas negras que usamos a diario sin conocer (en general) como funciona en su interior. Solo conocemos como controlarlos (entradas) y las respuestas que podemos obtener de los artefactos (salidas).

Principio de abstracción.

Manejo de la complejidad

- En principio diremos que escribir un programa "grande" generalmente lleva más tiempo que escribir uno "pequeño". Esto es verdadero si medimos "grande" y "pequeño" en unidades apropiadas.
- Claramente el número de instrucciones de un programa no es una medida de complejidad ya que existen instrucciones más complejas que otras, y algoritmos más complejos que otros. En realidad lo que diremos es que *es más difícil resolver un problema difícil!* , e intentaremos realizar un análisis sobre como disminuir la complejidad de un determinado problema.

Acoplamiento

- Muchos aspectos de la modularización pueden ser comprendidos solo si se examinan módulos en relación con otros.
- En principio veremos el concepto de ***independencia***. Diremos que dos módulos son totalmente independientes si ambos pueden funcionar completamente sin la presencia del otro. Esto implica que no existen interconexiones entre los módulos, y que se tiene un valor cero en la escala de "dependencia".
- El concepto de independencia funcional es una derivación directa del de modularidad y de los conceptos de abstracción y ocultamiento de la información.

Cohesión

La cohesión es la medida cualitativa de cuan estrechamente relacionados están los elementos internos de un módulo.

- Este concepto representa la técnica principal que posee un diseñador para mantener su diseño lo más semánticamente próximo al problema real, o dominio de problema.
- Claramente los conceptos de cohesión y acoplamiento están íntimamente relacionados. Un mayor grado de cohesión implica un menor de acoplamiento. Maximizar el nivel de cohesión intramódular en todo el sistema resulta en una minimización del acoplamiento intermódular.

La obligación del diseñador es conocer los efectos producidos por la variación en la cohesión, especialmente en términos de modularidad, en orden de realizar soluciones de *compromiso* beneficiando un aspecto en contra de otro.

En conclusión se puede decir:

UNA BUENA MODULARIDAD SE OBTIENE SI EXISTE UNA ALTA COHESIÓN Y UN BAJO ACOPLAMIENTO

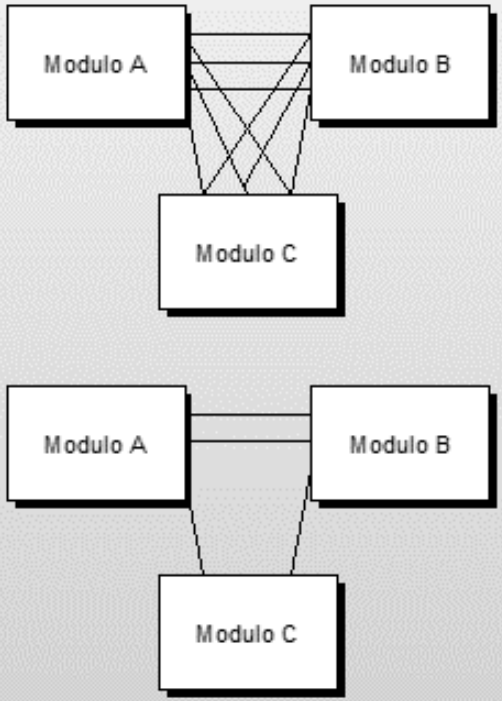
Cohesión y acoplamiento.

Acoplamiento mínimo:

- Numero de conexiones sea mínimo
- Flujo de información sea mínimo
- Comunicaciones explícitas

Cohesión interna:

- Cohesión de datos
- Cohesión funcional



RECURSIVIDAD

"Para entender la recursividad hay que primero entender la recursividad".

La recursividad (recursión) es aquella propiedad que posee un método por la cual puede llamarse a sí mismo.

- Puede utilizar la recursividad como una alternativa a la iteración, una solución recursiva es, normalmente, menos eficiente en términos de tiempo de computadora que una solución iterativa, debido a las operaciones auxiliares que llevan consigo las invocaciones suplementarias a los métodos.

- La recursión es una herramienta poderosa e importante en la resolución de problemas y en la programación.

Permite definir un objeto(problemas, estructura de datos) en términos de si mismo.

- Casos típicos de estructuras de datos definidas de manera recursiva son los arboles y las listas ligadas.

- Un subprograma que se llama así mismo se dice que es recursivo.

Cairo-Guardatti. "Estructura de datos"

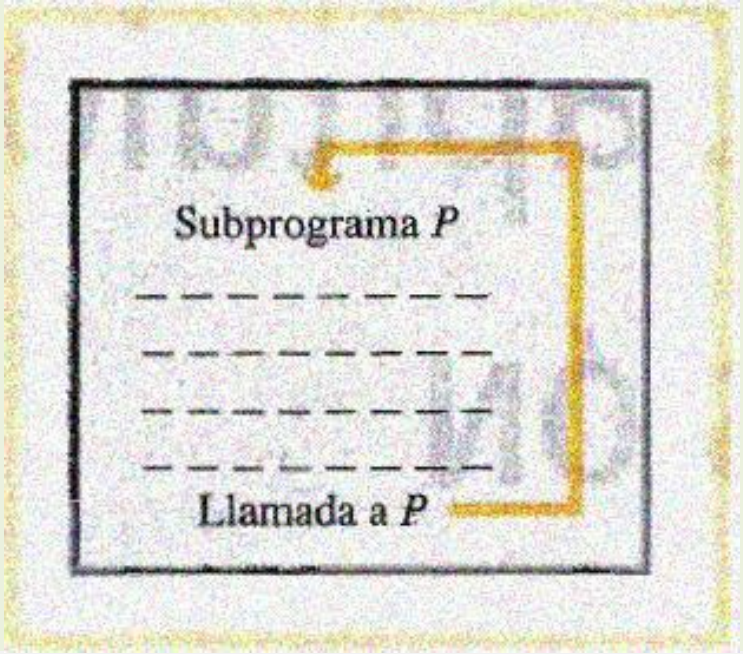
Se dice que un proceso es recursivo si forma parte de si mismo, o sea que se define en función de si mismo.

- La recursión es un proceso extremadamente potente, pero consume muchos recursos, razón por la cual es importante analizar detenidamente para saber cuando y como aplicarla.

Ceballos Fco. "Java 2, curso de programación"

La recursión en los subprogramas puede darse de dos maneras diferentes:

a) Directa: El subprograma se llama directamente así mismo. Por ejemplo:



Recursión Directa.

Solución Iterativa → Si X >= 0 XI = 1 * 2 * 3 * 4 * * X

Solución Recursiva →
Si X = 0 XI = 1
Si X > 0 XI = X * (X-1)I

Solución Iterativa

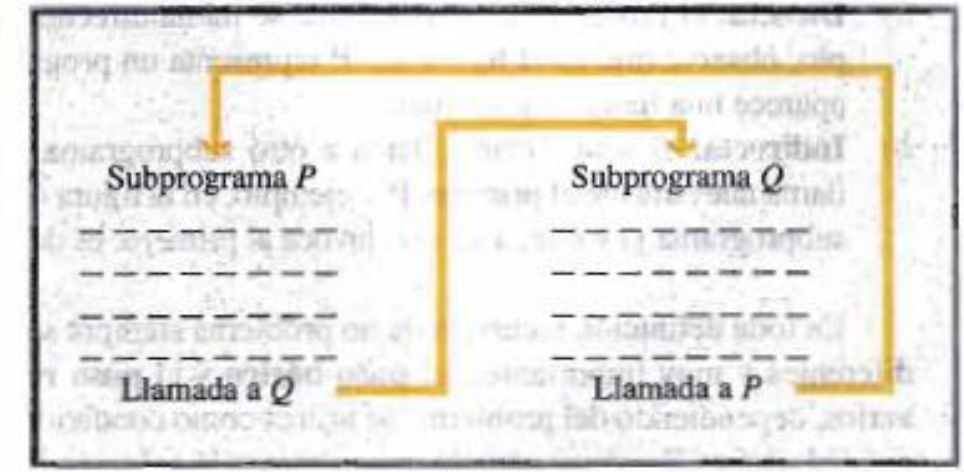
```
public static int factorial(int x)
{
        int fact = 1;
        for (int i=1; i<= x; i++)
        {
            fact = fact * i;
        }
        return  fact;
}
```

Solución Recursiva

```
public static int factorial(int x)
{
        if (x == 0)
    return 1;
        else
            return  (x * factorial(x – 1));
}
```

Recursión.

b) Indirecta: El subprograma llama a otro subprograma, y éste, en algún momento, llama nuevamente al primero. Por ejemplo:



Recursión indirecta.

```
public static boolean impar (int numero){
     if (numero==0)
      return false;
     else
      return par(numero-1);
 }

 public static boolean par (int numero){
     if (numero==0)
      return true;
     else
      return impar(numero-1);
 }
```

Recursividad.

- En toda definición recursiva de un problema se debe establecer un estado básico, es decir un estado en el cual la solución no se presente de manera recursiva si no directamente.

- La entrada (datos) del problema debe ir acercándose al estado básico.

Recursividad.

Según el número de llamadas recursivas generadas en tiempo de ejecución:

- Función recursiva lineal o simple: Se genera una única llamada interna.
- Función recursiva no lineal o múltiple: Se generan dos o más llamadas internas.

Recursividad.

Según el punto donde se realice la llamada recursiva, la función recursiva puede ser:

- Final: (Tail recursión): La llamada recursiva es la última instrucción que se produce dentro de la función.

- No final: (Nontail recursive Function): Se realiza alguna operación al volver de la llamada recursiva.

Diseño de funciones recursivas.

- El problema original se puede transformar en otro problema similar más simple
- Tenemos alguna manera directa de solucionar “problemas triviales”

Para que el módulo recursivo sea correcto se debe realizar:

- Un análisis por casos del problema: Existe al menos una condición de terminación en la cual no es necesario una llamada recursiva. Son los casos triviales que se solucionan en forma directa.

Si n=0 o n=1, el factorial es 1

Diseño de funciones recursivas.

Para que el módulo recursivo sea correcto se debe realizar:

- Convergencia de la llamada recursiva: Cada llamada recursiva se realiza con un dato más pequeño, de forma que se llegue a la condición de terminación.

  Factorial (n)=n*Factorial (n-1)

- Si las llamadas recursivas funcionan bien, el módulo completo funciona bien: principio de inducción.

  Factorial(0)=1

  Factorial(1)=1

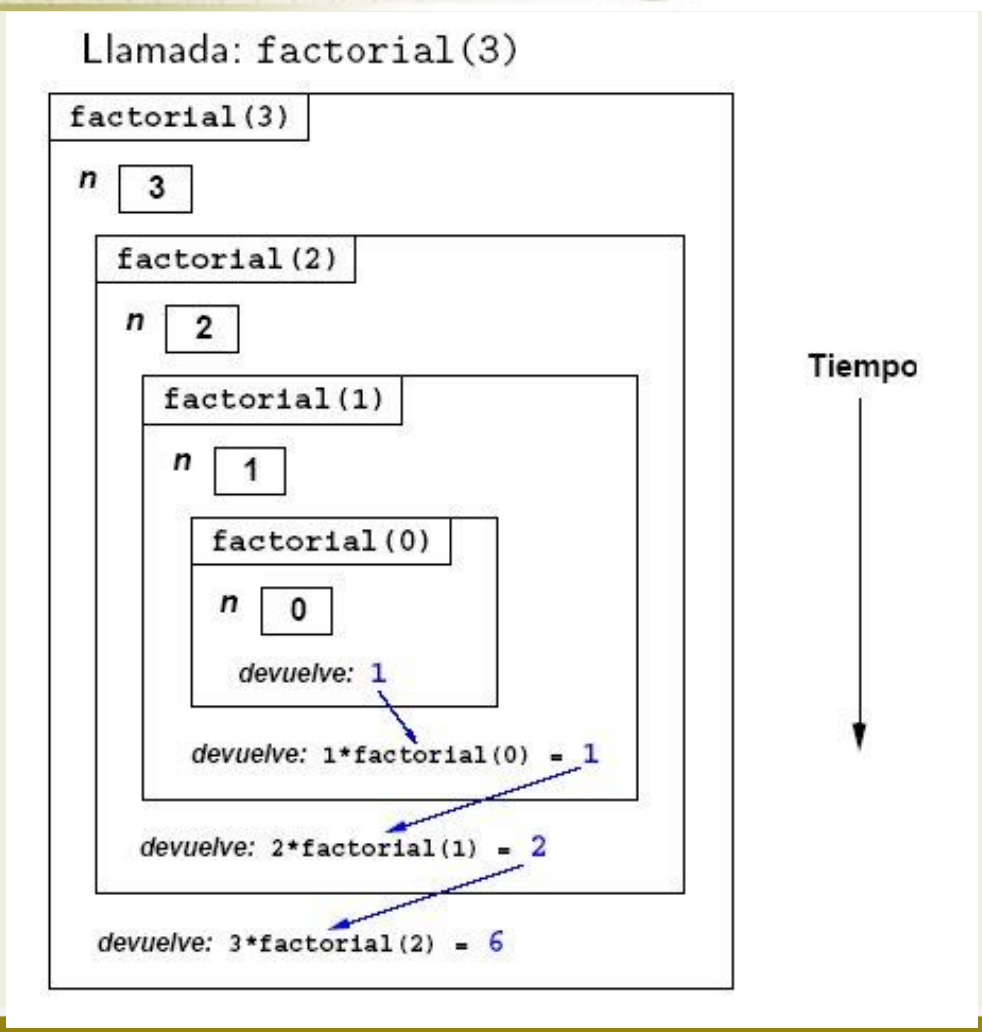
  Para n>1, si suponemos correcto el cálculo del factorial de (n-1),

  Factorial (n)=n*Factorial (n-1)

Diseño de funciones recursivas.

Gráficamente, la función factorial quedaría →

- Un requisito importante para que sea correcto un algoritmo recursivo es que no genere una secuencia infinita de llamadas así mismo.



Ventajas e inconvenientes de la recursividad.

Una función recursiva debe ser una manera natural, sencilla, comprensible y elegante del problema. Por ejemplo, dado un número entero no negativo, escribir su codificación en binario.

```
void binario(int n){
        if(n<2)
        printf("%d", n);
        else{
                binario(n/2);
                printf ("%d",n%2);
        }
}
```

Ventajas e inconvenientes de la recursividad.

- Otro elemento a tomar en cuenta es la facilidad para comprobar y verificar que la solución es correcta (inducción matemática).

- En general, las soluciones recursivas son más ineficientes en tiempo y espacio que las versiones iterativas, debido a las llamadas a subprogramas, la creación de variables dinámicas en la pila recursiva y la duplicación de variables. Otra desventaja es que en algunas soluciones recursivas repiten cálculos en forma innecesaria.

Ejemplo de recursividad.

Por ejemplo, el cálculo del n-ésimo término de la sucesión de Fibonacci.

```
int fibo(int n){
      if(n<2)
            return 1;
      return fibo(n-1) + fibo(n-2);
}
```

Recursividad.

En general, cualquier función recursiva se puede transformar en una función iterativa.

- Ventaja de la función iterativa: Más eficiente en tiempo y espacio.
- Desventaja de la función iterativa: en algunos casos, muy complicada; además, suelen necesitarse estructuras de datos auxiliares.

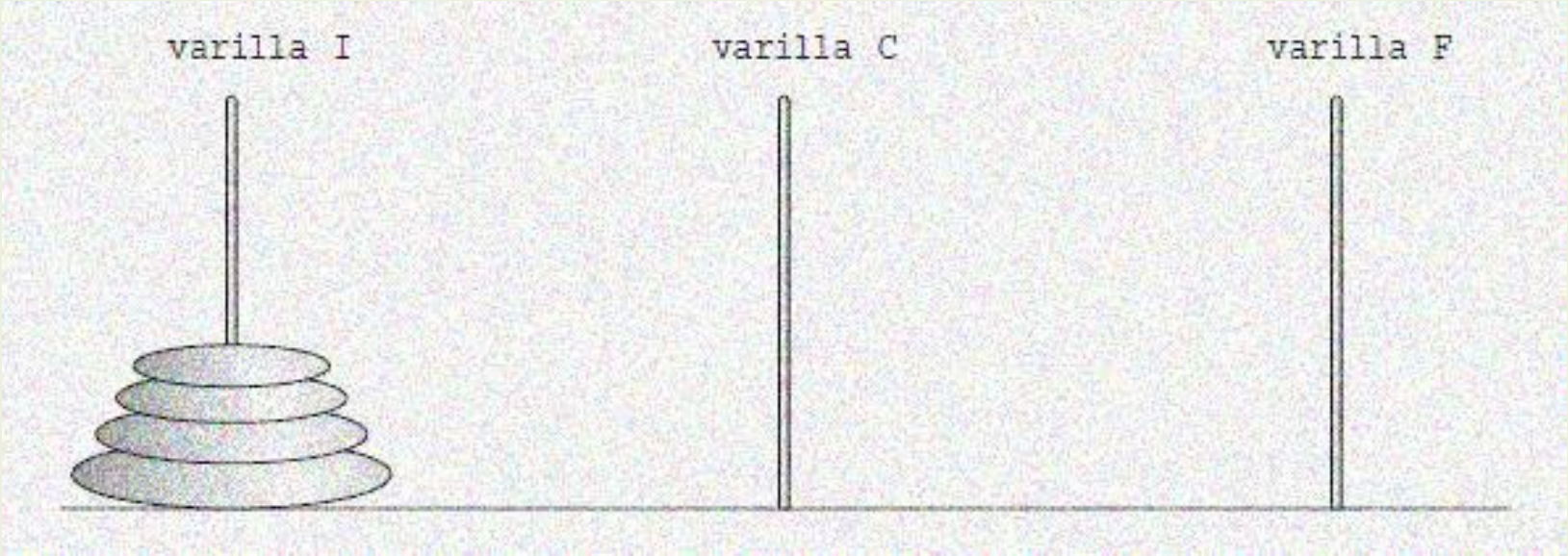
La recursión se puede simular con el uso de pilas para transformar un programa recursivo en iterativo. Las pilas se usan para almacenar los valores de los parámetros del subprograma, los valores de las variables locales, y los resultados de la función.

Recursividad, torres de hanoi.

El problema en cuestión supone la existencia de 3 varillas o postes en los que se alojaban discos, cada disco es ligeramente inferior en diámetro al que está justo debajo de él, y pretende determinar los movimientos necesarios para trasladar los discos de una varilla a otra cumpliendo las siguientes reglas:

- En cada movimiento sólo puede intervenir un disco.
- Nunca puede quedar un disco sobre otro de menor tamaño.

Recursividad, torres de hanoi.



Los cuatro discos situados en la varilla I se desean trasladar a la varilla F conservando la condición de que cada disco sea ligeramente inferior en diámetro al que tiene situado debajo de él.

Diseño del algoritmo de torres de hanoi.

El algoritmo se escribe generalizando para *n* discos y tres varillas. La función de Hanoi declara las varillas o postes como objetos cadena. En la lista de parámetros, el orden de las variables o varillas es:

```
varinicial varcentral varfinal
```

lo que implica que se están moviendo discos desde la varilla inicial a la final utilizando la varilla central como auxiliar para almacenar los discos. Si *n = 1* se tiene la condición de parada, ya que se puede manejar moviendo el único disco desde la varilla inicial a la varilla final.

Algoritmo de torres de hanoi.

```
1. Si n es 1
      1.1 Mover el disco 1 de varinicial a varfinal
2. Si_no
      1.2 Mover n – 1 discos desde varinicial hasta la
      varilla auxiliar utilizando varfinal
      1.3 Mover el disco n desde varinicial a varfinal
      1.4 Mover n – 1 discos desde la varilla auxiliar
      o central a varfinal utilizando la varilla
      inicial.
```

Es decir, si n es 1, se alcanza la condición de salida o terminación del algoritmo. Si n es mayor que 1, las etapas recursivas 1.2, 1.3 y 1.4 son tres subproblemas más pequeños, que aproximan a la condición de salida.

Implementación de las Torres de Hanoi.

- La implementación del algoritmo se apoya en los nombres de las tres varillas ‘A’, ‘B’ y ‘C’, que se pasan como parámetros al método ***hanoi( ).***

- El método tiene un cuarto parámetro, que es el número de discos ***n,*** que intervienen. Se obtiene un listado de movimientos que transferirán los ***n*** discos desde la varilla inicial ‘A’, a la varilla final, ‘C’.

Algoritmo de torres de hanoi.

```
Algoritmo Torres de Hanoi
rutina Principal
inicio
        Anillos ← Pide la cantidad de anillos a mover
        Mueve(Anillos, origen, intermedio, destino)
fin

Rutina Mueve (a, o, i, d)
inicio
        si a = 1
                entonces Muestra o " mover a " d
                en otro caso
                        Mueve(a-1, o, d, i)
                        Muestra o " mover a " d
                        Mueve(a-1, i, o, d)
Fin Si
Fin
```

Programa de torres de hanoi en C++.

```
// función recursiva Torres de Hanoi
void Hanoi(char varinicial, char varfinal, char varcentral, int n){
if ( n == 1)
cout << " Mover disco 1 de varilla " << varinicial << " a varilla "
<< varfinal << endl;
Else {
Hanoi(varinicial, varcentral, varfinal, (n - 1);
cout << " Mover disco " << n << " desde varilla " << varinicial <<
" a varilla " << varfinal << endl;
Hanoi(varcentral, varfinal, varinicial, n - 1);
}
}
```

Programa de torres de hanoi en Java.

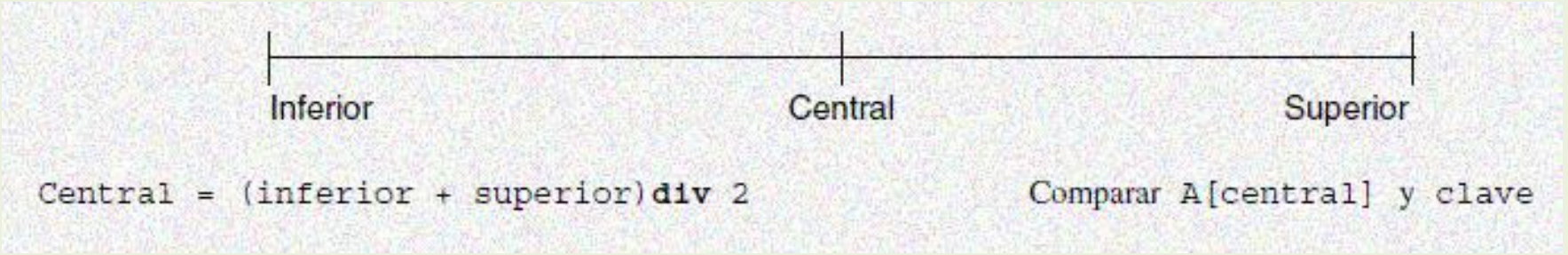
```
static void hanoi(char varinicial, char varcentral, char varfinal, int n)
        {
           if ( n == 1)
              System.out.println("Mover disco " + n + " desde varilla " +
                                      varinicial + " a varilla " + varfinal);
           else
           {
              hanoi(varinicial, varfinal, varcentral, n-1);
              System.out.println("Mover disco " + n + " desde varilla " +
                                     varinicial + " a varilla " + varfinal);

              hanoi(varcentral, varinicial, varfinal, n - 1);
           }
        }
```

Búsqueda binaria recursiva.

- La búsqueda binaria es el método de búsqueda de una clave especificada dentro de una lista o array ordenado de *n* elementos que realizaba una exploración de la lista hasta que se encontraba o no la coincidencia con la clave especificada.

- El algoritmo de búsqueda binaria se puede describir recursivamente. Supóngase que se tiene una lista ordenada ***A*** con un límite inferior y un límite superior. Dada una clave (valor buscado) se comienza la búsqueda en la posición central de la lista (índice central).

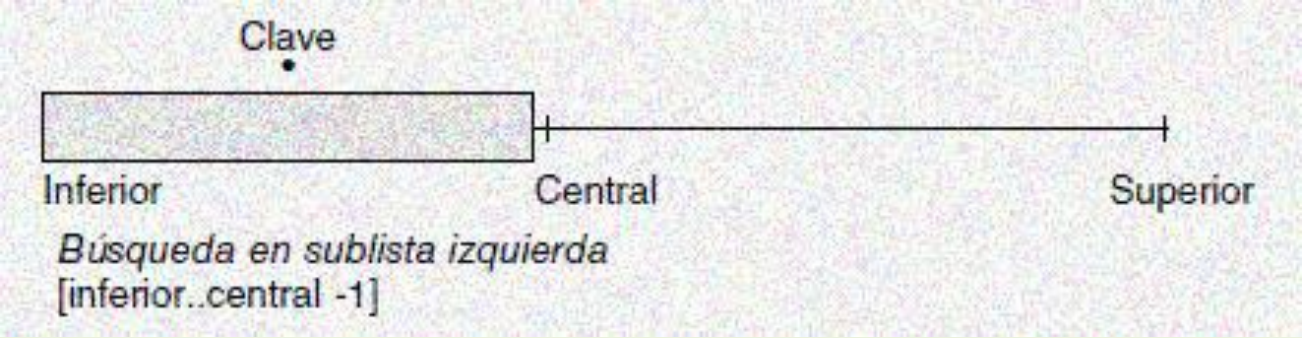
Búsqueda binaria recursiva.



Si se produce coincidencia (se encuentra la clave), se tiene la condición de terminación que permite detener la búsqueda y devolver el índice central. Si no se produce la coincidencia (no se encuentra la clave), dado que la lista está ordenada, se centra la búsqueda en la "sublista inferior" (a la izquierda de la posición central) o en la "sublista derecha" (a la derecha de la posición central).

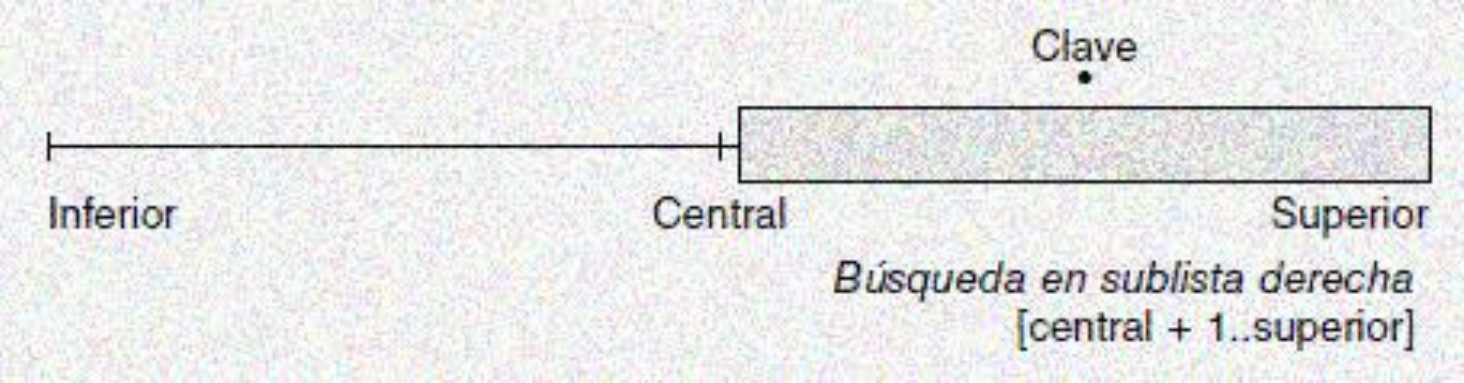
Algoritmo búsqueda binaria recursiva.

1. Si clave < A[central], el valor buscado sólo puede estar en la mitad izquierda de la lista con elementos en el rango inferior a central – 1.



Algoritmo búsqueda binaria recursiva.

2. Si clave > A[central], el valor buscado sólo puede entrar en la mitad derecha de la lista con elementos en el rango de índices, Central + 1 a Superior.



Algoritmo búsqueda binaria recursiva.

3. El proceso recursivo continúa la búsqueda en sublistas más y más pequeñas. La búsqueda termina o con éxito (*aparece la clave buscada*) o sin éxito (*no aparece la clave buscada*), situación que ocurrirá cuando el límite superior de la lista sea más pequeño que el límite inferior.

La condición Inferior > Superior será la condición de salida o terminación y el algoritmo devuelve el índice – 1.

Algoritmo búsqueda binaria recursiva.

```
BusquedaBR(int a[], int clave, int inferior, int superior) //
BR, binaria recursiva
        int central;
        si inferior > superior
                devolver -1 // valor no encontrado
        si no
                central ← (inferior + superior)/2
                si a[central] = clave
                devolver central
                        si a[central] < clave
                        devolver
                        BusquedaBR(a,clave,central+1,superior)
                        si no a[central] > clave
                        devolver
                        BusquedaBR(a,clave,inferior,central-1)
```

Programa de búsqueda binaria recursiva.

```
int binaria(int a[], int clave, int inferior, int superior){
	int central;
	if (inferior>superior)
	return -1;
	else{
		central = (inferior+superior)/2;
		if (a[central]==clave)
		return central;
		else if (a[central]<clave)
		return binaria(a,clave,central+1,superior);
		else
		return binaria(a,clave,inferior,central-1);
	}
}
```

BIBLIOGRAFÍA

- Cairó, Osvaldo y Guardati, Silvia. (2002). Estructuras de datos (2a. Edición). McGraw-Hill.

- Drozdeck, Adam. (2007). Estructuras de datos y algoritmos en Java (2ª Edición). Thomson.

- Joyanes Aguilar L., Fundamentos de programación, algoritmos, estructuras de datos y objetos, 4ed. McGrawHill.

- Fuente de internet consultada: http://www.ugr.es/~eaznar/fibo.htm