Departamento
de Ingeniería de la
Información y las
Comunicaciones
UMU

Tecnología de la Programación

Título de Grado en Ingeniería Informática

Curso 2023/24

Grupo 4

Tema 4. Recursividad

Dept. Ingeniería de la Información y las Comunicaciones

Universidad de Murcia

Tema 4. Recursividad

4.1 Definición de recursividad (1/5)

Se dice que un ***proceso es recursivo*** si se especifica basándose en su propia definición. Un ***objeto es recursivo*** si forma parte de sí mismo.

La recursividad puede aparecer en la vida real en forma de imágenes, frases hechas, acrónimos, etc. Por ejemplo, el acrónimo informático *GNU* significa "*GNU's Not Unix*".

La ***recursividad*** o ***recursión*** tiene gran relevancia en los campos de las **matemáticas** y de **las ciencias de la computación**.

"El poder de la recursión evidentemente se fundamenta en la posibilidad de definir un conjunto infinito de objetos con una declaración finita. Igualmente, un número infinito de operaciones computacionales puede describirse con un programa recursivo finito, incluso en el caso de que este programa no contenga repeticiones explícitas." (Wirth,Niklaus, Algorithms + Data Structures = Programs. Prentice-Hall. 1976).

4.1 Definición de recursividad (2/5)

Recursividad en Matemáticas

La recursividad en matemáticas puede usarse para la **definición de conjuntos**, **funciones**, **constantes**, **relaciones de recurrencia**, etc.

Ejemplos:

Conjunto de los *números naturales*:

- 1 es un número natural.
- Si n es un número natural, entonces $n+1$ es un número natural.

Función *factorial*, *n!* (para números enteros no negativos):

$$n! = \begin{cases} 1 & \text{si } n = 0 \\ n \cdot (n-1)! & \text{si } n > 0 \end{cases}$$

***Razón áurea* mediante fracciones continuas:**

$$\varphi = 1 + \frac{1}{\varphi} = 1 + \cfrac{1}{1 + \cfrac{1}{1 + \cfrac{1}{\cdots}}}$$

4.1 Definición de recursividad (3/5)

Recursividad en Ciencias de la Computación

En ciencias de la computación la recursividad es una forma de ***resolución de problemas***. Para resolver un problema éste se divide en problemas derivados de menor tamaño del mismo tipo, lo que constituye la ***técnica de diseño de algoritmos*** denominada ***divide y vencerás***. La recursividad es también parte esencial de la técnica de diseño de algoritmos de ***programación dinámica***, así como del ***backtracking*** (***vuelta atrás***).

El ***diseño recursivo*** está basado en el concepto matemático de ***inducción***: a un problema que puede ser definido en función de su tamaño *N*, que puede ser dividido en instancias más pequeñas (menor que *N*) del mismo problema, y que se conoce la solución explícita para las instancias más simples, denominadas casos base, se le puede aplicar inducción sobre las instancias más pequeñas y suponer que estas quedan resueltas, y aplicar finalmente una composición de las soluciones de los subproblemas para obtener la solución del problema original.

4.1 Definición de recursividad (4/5)

Recursividad en Ciencias de la Computación

En general, un ***algoritmo recursivo*** P puede expresarse como una composición $\wp$ de instrucciones básicas S (que no contienen a P) y el propio P :

$$P \equiv \wp[S,P]$$

Los algoritmos recursivos son especialmente apropiados cuando el problema que resuelven o los datos que manejan están definidos en términos recursivos.

Un ejemplo típico de algoritmo recursivo basado en la técnica divide y vencerás es el algoritmo de ***ordenación por mezcla*** (***mergesort***). Dado un vector v de N elementos, éste se divide en dos vectores de $N/2$ elementos. Por inducción, se puede aplicar la ordenación de estos dos subproblemas. Una vez ordenados ambos vectores, simplemente hay que aplicar una composición de éstos basada en la mezcla de sus elementos. El caso base es ordenar un vector de 1 ó 0 elementos que está ya ordenado y por tanto no hay que realizar ningún proceso sobre él.

4.1 Definición de recursividad (5/5)

En ***Programación de Ordenadores***, una **función** (o **procedimiento**) **es recursiva** si se llama a sí misma. Una función recursiva está bien construida si:

1. Contiene, al menos, un ***caso base***, que no contiene ninguna llamada recursiva.

2. Los valores de los parámetros de la función en las sucesivas llamadas recursivas, están más cerca del caso base que el valor con el que se llamó inicialmente la función, garantizando que el caso base se alcanza. Al caso que contiene la llamada recursiva se le llama ***caso general*** o ***de recursión*** (pueden ser varios).

Otra aplicación importante de la recursividad en programación es la posibilidad de definir ***estructuras de datos recursivas*** (Temas 2 y 5) las cuales pueden crecer de forma dinámica hasta un tamaño teórico infinito en función de los requerimientos de la aplicación.

4.2 Ejemplos de Recursividad en *C* (1/11)

Multiplicación: $n \cdot m$ **(** $m \geq 0$ **)**

Caso Base ($m = 0$): $n \cdot 0 = 0$

Caso General ($m > 0$): $n \cdot m = n + n \cdot (m - 1)$

```
int multiplica(int n, int m)
{
   int salida;
   if (m==0) salida = 0;
   else salida=n + multiplica(n,m-1);
   return salida;
}
```

```
// Versión iterativa
int multiplica(int n, int m)
{
   int salida = 0;
   while(m>0) { salida+=n; m--; }
   return salida;
}
```

Esquemas recursivos más compactos en *C*:

```
int multiplica(int n, int m)
{
   if (m==0) return 0;
   return n + multiplica(n,m-1);
}
```

```
int multiplica(int n, int m)
{
   return m==0?0:n+multiplica(n,m-1);
}
```

4.2 Ejemplos de Recursividad en *C* (2/11)

División entera: n / m **(** $n \geq 0$ **y** $m > 0$ **)**

Caso Base ($n < m$): $n / m = 0$
Caso General ($n \geq m$): $n / m = 1 + (n - m) / m$

```
int divide(int n, int m)
{
     if (n<m) return 0;
     return 1 + divide(n-m,m);
}
```

```
// Versión iterativa
int divide(int n, int m)
{
     int salida = 0;
     while(n>=m) { salida++; n-=m;}
     return salida;
}
```

4.2 Ejemplos de Recursividad en *C* (3/11)

Módulo: $n \,\%\, m$ **(** $n \geq 0$ **y** $m > 0$ **)**

Caso Base ($n < m$): $n \,\%\, m = n$

Caso General ($n \geq m$): $n \,\%\, m = (n - m) \,\%\, m$

```
int modulo(int n, int m)
{
     if (n<m) return n;
     return modulo(n-m,m);
}
```

```
// Versión iterativa
int modulo(int n, int m)
{
     while(n>=m) n-=m;
     return n;
}
```

4.2 Ejemplos de Recursividad en *C* (4/11)

Potencia: n^m **(** **(** $m > 0$ **)** **o** **(** **(** $m = 0$ **)** **y** **(** $n \neq 0$ **)** **)** **)**

Caso Base ($m = 0$): $n^0 = 1$

Caso General ($m > 0$): $n^m = n \cdot n^{m-1}$

```
int potencia(int n, int m)
{
     if (m==0) return 1;
     return n * potencia(n,m-1);
}
```

```
// Versión iterativa
int potencia(int n, int m)
{
     int salida = 1;
     while(m>0){ salida*=n; m--; }
     return salida;
}
```

4.2 Ejemplos de Recursividad en *C* (5/11)

Factorial: $n!$ **(** $n \geq 0$ **)**

Caso Base ($n = 0$): $0! = 1$

Caso General ($n > 0$): $n! = n \cdot (n-1)!$

```
int factorial(int n)
{
     if (n==0) return 1;
     return n * factorial(n-1);
}
```

```
// Versión iterativa
int factorial(int n)
{
     int salida = 1;
     while(n>0){ salida*=n; n--; }
     return salida;
}
```

4.2 Ejemplos de Recursividad en *C* (6/11)

Fibonacci: ***fibonacci(n)*** **($n \geq 0$)**

Caso Base ($n \leq 1$): $fibonacci(n) = n$
Caso General ($n > 1$): $fibonacci(n) = fibonacci(n-2) + fibonacci(n-1)$

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

```
// Versión iterativa
int fibonacci(int n)
{
     int i = 1; int j = 0;
     for(int k=0; k<n; k++) { int aux = i+j; i = j; j = aux; }
     return j;
}
```

4.2 Ejemplos de Recursividad en *C* (7/11)

El conjunto de ***cadenas de caracteres*** puede definirse de forma recursiva de la siguiente forma:

- La cadena vacía es una cadena de caracteres.
- Si *s* es una cadena de caracteres, entonces *s* + *c* es una cadena de caracteres, donde + es la operación de concatenación por la derecha, y *c* es un carácter.

A partir de esta definición recursiva, pueden definirse funciones recursivas sobre cadenas de caracteres.

4.2 Ejemplos de Recursividad en *C* (8/11)

Función que comprueba si un carácter está contenido en una cadena de caracteres:

```
int buscar(char * s, char c) // Versión recursiva
{
     if (s[0]==0) return 0;
     if (s[0]==c) return 1;
     return buscar(s+1,c);
}
```

```
int buscar(char * s, char c) // Versión recursiva más comprimida
{
     if (s[0]==0) return 0;
     return (s[0]==c)||buscar(s+1,c);
}
```

```
int buscar(char * s, char c) // Versión iterativa
{
     int i=0;
     while((s[i]!=0)&&(s[i]!=c)) i++;
     return (s[i]!=0);
}
```

4.2 Ejemplos de Recursividad en *C* (9/11)

Función que compara dos cadenas de caracteres:

```
int iguales(char * s1, char *s2) // Versión recursiva
{
     if ((s1[0]==0)&&(s2[0]==0)) return 1;
     return (s1[0]==s2[0]) && iguales(s1+1,s2+1);
}
```

```
int iguales(char * s1, char *s2) // Versión iterativa
{
     int i=0;
     while ((s1[i]!=0) && (s2[i]!=0) && (s1[i]==s2[i]))
          i++;
     return (s1[i]==s2[i]);
}
```

4.2 Ejemplos de Recursividad en *C* (10/11)

Función que invierte una cadena de caracteres, versión recursiva:

```
void invertir(char * s)
{
      int longitud=0;
      while (s[longitud]!=0) longitud++;
      invertirAux(s,0,longitud-1);
}
```

```
void invertirAux(char * s, int i, int j) // Función auxiliar recursiva
{
      if (i<j)
      {
            char aux = s[i];
            s[i] = s[j];
            s[j] = aux;
            invertirAux(s,i+1,j-1);
      }
}
```

4.2 Ejemplos de Recursividad en *C* (11/11)

Función que invierte una cadena de caracteres, versión iterativa:

```
void invertir(char * s)
{
    int longitud=0;
    while (s[longitud]!=0) longitud++;
    for(int i=0,j=longitud-1;i<j;i++,j--)
    {
        char aux = s[i];
        s[i] = s[j];
        s[j] = aux;
    }
}
```

4.3 Tipos de Recursividad (1/3)

Existen varios tipos de recursividad dependiendo de cómo se realice la llamada recursiva:

- ***Recursividad directa* o *simple*:** La función contiene una llamada explícita a sí misma, es decir, A invoca a A.
- ***Recursividad múltiple*:** Existe más de una llamada recursiva en el cuerpo de la función.
- ***Recursividad indirecta* o *cruzada*:** La función se invoca a sí misma de forma indirecta, es decir, A invoca a B y B invoca a A.
- ***Recursividad de cola* o de *extremo final*:** La llamada recursiva es la última instrucción que ejecuta la función.
- ***Recursividad anidada*:** En alguno de los argumentos de la llamada recursiva hay una nueva llamada recursiva.

4.3 Tipos de Recursividad (2/3)

La función *modulo* es un ejemplo de recursividad directa y de extremo final. La función *fibonacci* es un ejemplo de recursividad múltiple.

Ejemplo de recursividad indirecta o cruzada:

Par: ***par(n)*** **(** $n \geq 0$ **)**

Caso Base ($n = 0$): *par(n) = cierto*

Caso General ($n > 0$): *par(n) = impar(n-1)*

```
int par(int n)
{
    if (n==0) return 1;
    return impar(n-1);
}
```

Impar: ***impar(n)*** **(** $n \geq 0$ **)**

Caso Base ($n = 0$): *impar(n) = falso*

Caso General ($n > 0$): *impar(n) = par(n-1)*

```
int impar(int n)
{
    if (n==0) return 0;
    return par(n-1);
}
```

4.3 Tipos de Recursividad (3/3)

Ejemplo de recursividad anidada:

Ackermann: ***ack(m,n)*** **(** $n \geq 0$ **y** $m \geq 0$ **)**

Caso Base ($m = 0$): $ack(m,n) = n+1$
Caso General ($m > 0$ y $n = 0$): $ack(m,n) = ack(m-1,1)$
($m > 0$ y $n > 0$): $ack(m,n) = ack(m-1, ack(m,n-1))$

```
int ackermann(int m, int n)
{
     if (m==0) return n+1;
     if (n==0) return ackermann(m-1,1);
     return ackermann(m-1,ackermann(m,n-1));
}
```

4.4 La pila de llamadas (1/2)

Una ***pila de llamadas*** (***call stack***) es una estructura dinámica de datos que almacena información sobre las subrutinas activas en un programa. Este tipo de pila es conocida también como pila de ejecución, pila de control, pila de función o pila de tiempo de ejecución.

Cada llamada produce un **registro de activación** (***strack frame***) en la pila, en donde se guardan:

- Variables locales.
- Argumentos.
- Dirección de retorno.
- Valor devuelto.
- Dirección del frame anterior.
- Valores de registros modificados por la función.

Las llamadas recursivas se gestionan exactamente igual que el resto de llamadas.

A continuación mostramos, para el ejemplo de la función factorial, la evolución de la pila de llamadas del sistema.

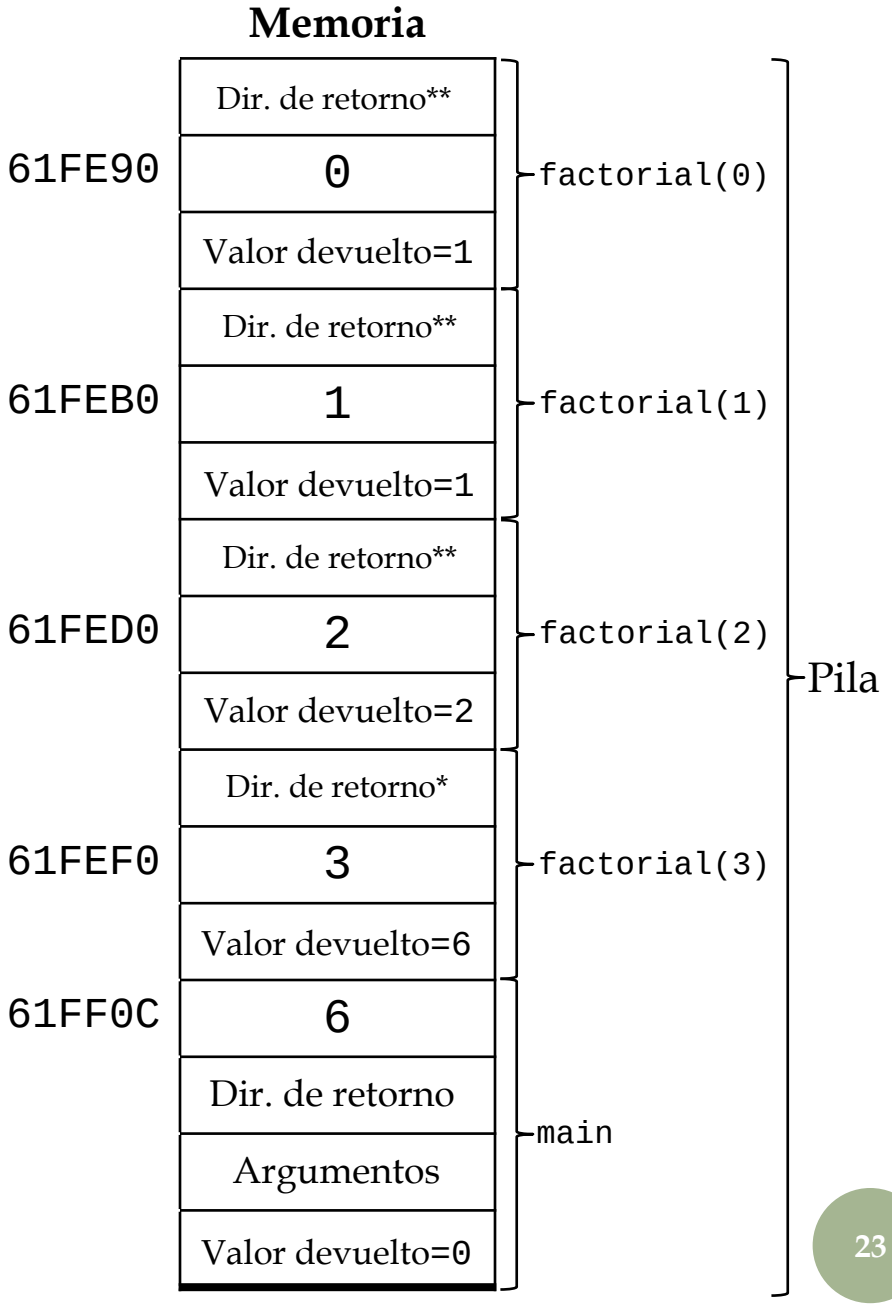
4.4 La pila de llamadas (2/2)

Tabla de símbolos

Identificador	Tipo	Dirección
f	int	61FF0C
n	int	61FEF0

```
int factorial(int n)
{
    if (n==0) return 1;
    return n*factorial(n-1); // **
}
int main()
{
    int f = factorial(3);    // *
    printf("%d",f);
    return 0;
}
```

Nota: Las direcciones son reales obtenidas con un Procesador Intel(R) Core(TM) i5-4460 CPU@ 3.20GHz 8,00 GB, GNU GCC compiler. No se han visualizado los valores de los registros modificados por las funciones ni las direcciones al frame anterior. Las direcciones en la tabla de símbolos son absolutas (no relativas al comienzo del bloque).



4.5 El árbol de recursión (1/2)

El ***árbol de recursión*** permite obtener una visión gráfica global de la ***complejidad de un algoritmo recursivo***, así como de la forma en la que se producen las llamadas recursivas.

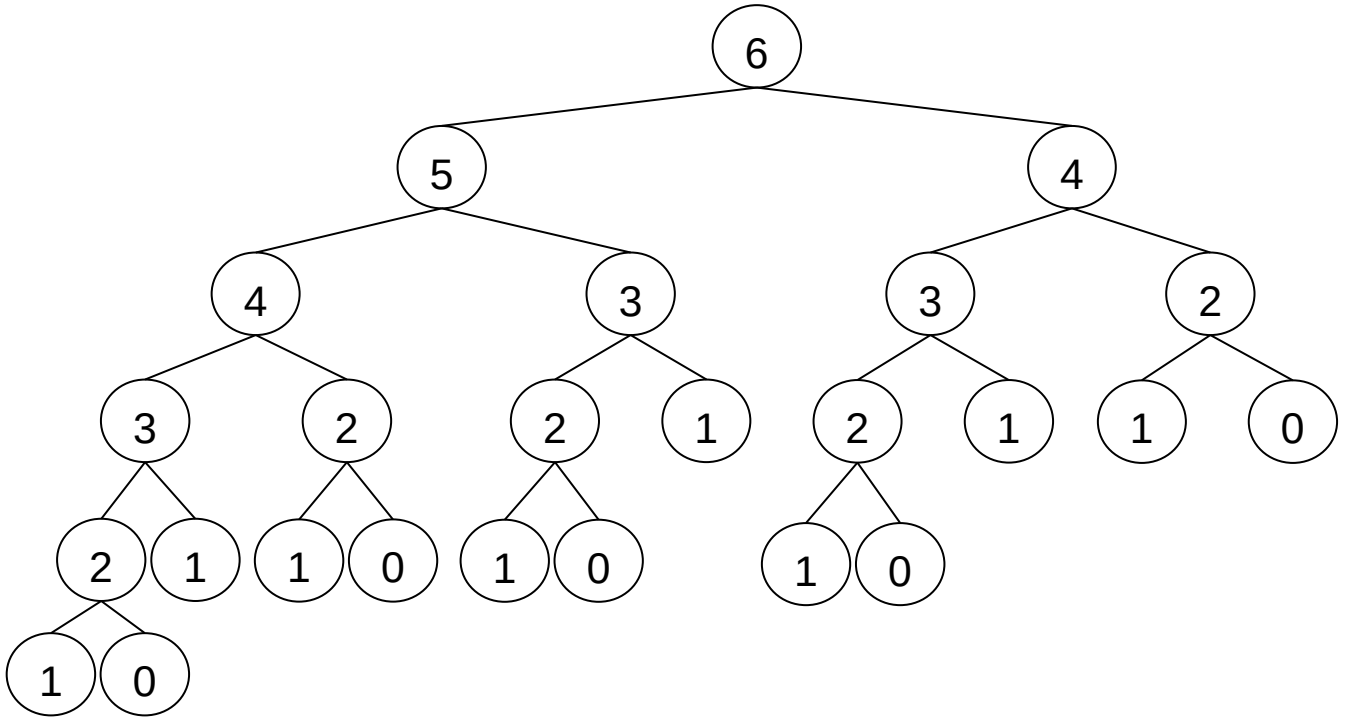
Esta información puede resultar de gran utilidad a la hora de decidir si el enfoque recursivo realizado para el problema es apropiado o por el contrario no resulta adecuado y es mejor buscar otro enfoque (por ejemplo, iterativo).

En un árbol de recursión **cada nodo representa una subllamada recursiva**, incluyendo en el contenido del nodo el valor de los parámetros de llamada.

Los nodos hoja en el árbol representan los casos base de la recursión.

4.5 El árbol de recursión (2/2)

Ejemplo (Árbol de recursión de la función *fibonacci(6)*):



4.6 Tiempo de ejecución de los algoritmos recursivos (1/2)

El mecanismo de ***expansión de recurrencias*** para calcular el ***tiempo de ejecución T(n)*** de un algoritmo recursivo con entrada ***n*** es el siguiente:

- Se obtiene la ***ecuación de recurrencia*** que expresa el tiempo de ejecución del caso base (suponemos $n = 1$) y del caso general ($n > 1$).

- Se sustituye $T(n)$ por su definición recurrente. Al aparecer entonces $T(m)$ con $m < n$ en la derecha, se vuelve a sustituir por su definición recurrente en la derecha, hasta que, por ***inducción en i***, se obtiene la ***Forma Cerrada*** para $T(n)$.

- La expansión terminará cuando se alcance $T(1)$ en el lado derecho de la forma cerrada. Como $T(1)$ siempre es constante, se tiene una fórmula para $T(n)$ en función de n y de algunas constantes.

4.6 Tiempo de ejecución de los algoritmos recursivos (2/2)

Ejemplo (Factorial):

Ecuación de recurrencia:

$T(0) = c_1$
$T(n) = T(n-1) + c_2$, *si* $n>0$

Expansión de recurrencias:

$$T(n) = T(n-1)+c_2 = [T(n-2)+c_2]+c_2 = T(n-2)+2c_2 = [T(n-3)+c_2]+2c_2 = T(n-3)+3c_2$$

Forma cerrada para T(n) (por inducción en i):

$$T(n) = T(n-i) + i\ c_2$$

La expansión terminará cuando se alcance $T(0)$ en el lado derecho de la forma cerrada, es decir, cuando $n-i=0$, por tanto $i=n$. Sustituyendo en la forma cerrada i por n se obtiene:

$$T(n) = T(0) + n\ c_2 = c_1 + n\ c_2$$

Esto demuestra que *T(n)* es *O(n)*.

4.7 Recursión vs iteración

La recursión implica múltiples llamadas y el uso de la pila interna para almacenar, en cada llamada recursiva, los parámetros de llamada, variables locales y dirección de retorno, lo que hace que en general sea más ineficiente que el diseño iterativo.

Siempre es posible sustituir un algoritmo recursivo por un algoritmo iterativo que utilice una pila.

Casos en los que no se debe utilizar la recursión:

1. Recursividad de cola.
2. Árbol de recursión lineal (ejemplo, factorial).
3. Árbol de recursión ramificado, pero con nodos repetidos (ejemplo Fibonacci).

Conclusión: La recursión se deberá utilizar cuando, además de facilitar el diseño del algoritmo, genere un árbol de recursión ramificado y con nodos no repetidos.

4.8 Ejercicios resueltos (1/9)

1) *Torres de Hanoi*: El juego, en su forma más tradicional, consiste en tres postes verticales. En uno de los postes se apila un número indeterminado de discos perforados por su centro. Los discos se apilan sobre uno de los postes en tamaño decreciente de abajo a arriba. No hay dos discos iguales, y todos ellos están apilados de mayor a menor radio, desde la base del poste hacia arriba, en uno de los postes, quedando los otros dos postes vacíos. El juego consiste en pasar todos los discos desde el poste ocupado (es decir, el que posee la torre) a uno de los otros postes vacíos, usándose el tercer poste como auxiliar. Para realizar este objetivo, es necesario seguir tres simples reglas:

   1. Solo se puede mover un disco cada vez.
   2. Un disco de mayor tamaño no puede estar sobre uno más pequeño que él mismo.
   3. Solo se puede desplazar el disco que se encuentre arriba en cada poste.

   Escribir una función recursiva en *C* que resuelva el juego de las torres de Hanoi, imprimiendo en pantalla todos los movimientos de los discos a realizar, usando tres postes (`origen`, `destino` y `auxiliar`) y `n` discos.

```
void Hanoi(char origen, char destino, char auxiliar, int n)
{
...
}
```

4.8 Ejercicios resueltos (2/9)

```
#include  <stdio.h>
void Hanoi(char * origen, char * destino, char * auxiliar, int n)
{
    if (n>0)
    {
        Hanoi(origen,auxiliar,destino,n-1);
        printf("Mover disco de %s a %s\n",origen,destino);
        Hanoi(auxiliar,destino,origen,n-1);
    }
}
int main()
{
    int n;
    printf("Introduce n%cmero de discos: ",163);
    scanf("%d",&n);
    Hanoi("origen","destino","auxiliar",n);
    return 0;
}
```

4.8 Ejercicios resueltos (3/9)

```
Introduce número de discos: 3
Mover disco de origen a destino
Mover disco de origen a auxiliar
Mover disco de destino a auxiliar
Mover disco de origen a destino
Mover disco de auxiliar a origen
Mover disco de auxiliar a destino
Mover disco de origen a destino

Process returned 0 (0x0)   execution time : 3.716 s
Press any key to continue.
```

4.8 EJERCICIOS RESUELTOS (4/9)

2) Implementar recursivamente en *C* el método de ordenación *MergeSort* para estructuras estructuras enlazadas **sin** nodo de encabezamiento.

```
struct Nodo
{
     int elem;
     struct Nodo * sig;
};
typedef struct Nodo * NodoPtr;

void MergeSort(NodoPtr * l, int n)
{
...
}
```

Nota: No usamos nodo de encabezamiento por que complica en exceso el tratamiento recursivo. Esto implica un paso por referencia de `l`. En cualquier caso, siempre es posible añadir el nodo de encabezamiento, si este requiere, después de hacer `MergeSort`. Se pasa también la longitud n de la lista por razones de eficiencia.

4.8 Ejercicios resueltos (5/9)

```
#include  <stdio.h>
#include  <stdlib.h>
struct Nodo
{
     int elem;
     struct Nodo * sig;
};
typedef struct Nodo * NodoPtr;

void divide(NodoPtr l, NodoPtr * lizq, NodoPtr * lder, int n)
{
     *lizq=l;
     for (int i=1; i<n/2; i++)
          l=l->sig;
     *lder=l->sig;
     l->sig=NULL;
}
```

4.8 Ejercicios resueltos (6/9)

```
NodoPtr Merge(NodoPtr lizq, NodoPtr lder)
{
      NodoPtr inicio, actual, posterior;
      if (lizq->elem<lder->elem)
      {
            inicio=lizq; actual=lizq; posterior=lder;
      }
      else
      {
            inicio=lder; actual=lder; posterior=lizq;
      }
      while (actual->sig!=NULL)
            if (actual->sig->elem>posterior->elem)
            {
                  NodoPtr aux=actual->sig;
                  actual->sig=posterior;
                  actual=posterior;
                  posterior=aux;
            }
            else
                  actual=actual->sig;
      actual->sig=posterior;
      return inicio;
}
```

4.8 Ejercicios resueltos (7/9)

```
void MergeSort(NodoPtr * l, int n) // Paso por referencia de un apuntador
{
     if (n>1)
     {
          NodoPtr lizq, lder;
          divide(*l,&lizq,&lder,n);
          MergeSort(&lizq,n/2);
          MergeSort(&lder,n-n/2);
          *l=Merge(lizq,lder);
     }
}
```

4.8 Ejercicios resueltos (8/9)

```
int main()
{
    int n;
    printf("Introduce n%cmero de datos: ",163);
    scanf("%d",&n);
    NodoPtr l=NULL;
    if (n>0)
    {
        int dato;
        printf("Introduce dato %d: ",1);
        scanf("%d",&dato);
        NodoPtr aux=malloc(sizeof(int));
        aux->elem = dato;
        l=aux;
        for (int i=2; i<=n; i++)
        {
            printf("Introduce dato %d: ",i);
            scanf("%d",&dato);
            aux->sig=malloc(sizeof(int));
            aux->sig->elem = dato;
            aux=aux->sig;
        }
        aux->sig=NULL;
    }
    imprimir(l);
    MergeSort(&l,n);
    imprimir(l);
    return 0;
}
```

4.8 Ejercicios resueltos (9/9)

```
Introduce número de datos: 8
Introduce dato 1: 7
Introduce dato 2: 5
Introduce dato 3: 3
Introduce dato 4: 1
Introduce dato 5: 8
Introduce dato 6: 6
Introduce dato 7: 4
Introduce dato 8: 2
< 7  5  3  1  8  6  4  2 >
< 1  2  3  4  5  6  7  8 >

Process returned 0 (0x0)   execution time : 35.933 s
Press any key to continue.
```

4.9 Ejercicios propuestos (1/3)

1) Escribe una función recursiva que, dada una cadena, muestre por pantalla una secuencia de cadenas de longitud creciente empezando por la última letra y progresando hasta ser igual a la cadena original. Por ejemplo, para la cadena `"Hola"` se debe mostrar:

```
a
la
ola
Hola
```

2) Escribe una función recursiva que, dada una cadena, muestre por pantalla una secuencia de cadenas de longitud decreciente en la que cada una tenga una letra menos que la anterior. Por ejemplo, para la cadena `"Hola"` se debe mostrar:

```
Hola
ola
la
a
```

4.9 Ejercicios propuestos (2/3)

3) Escribe una función recursiva que, dado un número entero mayor que cero, calcule la suma de todos los enteros positivos menores o igual que él.

```
int sumatorio(int n)
{
...
}
```

4) Escribe una función recursiva que sume los dígitos de un número entero mayor que cero pasado como parámetro.

```
int suma_digitos(int n)
{
...
}
```

4.9 Ejercicios propuestos (3/3)

5) Escribe una función recursiva que devuelva 1 si la cadena pasada como parámetro es un palíndromo y 0 en caso contrario. Un palíndromo es una frase que se lee igual de izquierda a derecha que de derecha a izquierda, por ejemplo, la palabra `"reconocer"`.

```
int palindromo( char * cadena )
{
...
}
```

6) Escribe una función recursiva que sustituya, en una cadena pasada como parámetro, todas las apariciones del carácter `a` por el carácter `b`.

```
void sustituye( char * cadena, char a, char b )
{
...
}
```