CAPITULO 5
Algoritmos recursivos

Objetivos

Una vez que haya leído y estudiado este capítulo, usted podrá:

- Conocer cómo funciona la recursividad.
- Distinguir entre recursividad y recursividad indirecta.
- Resolver problemas numéricos sencillos aplicando métodos recursivos básicos.
- Aplicar la técnica algorítmica *divide y vence* para la resolución de problemas.
- Determinar la complejidad de algoritmos recursivos mediante inducción matemática.
- Conocer la técnica algorítmica de resolución de problemas *vuelta atrás: backtracking*.
- Aplicar la técnica de *backtracking* al problema de la *selección óptima.*

Contenido

Conceptos clave

- *Backtracking*.
- Búsqueda exhaustiva.
- Caso base.
- Complejidad.
- *Divide y vence*.
- Inducción.
- Iteración versus recursión.
- Mejor selección.
- Recursividad.
- Torres de Hanoi.

Para profundizar (página web: www.mhe.es/joyanes)

- Ordenación por mezclas: *mergesort*.
- Resolución de problemas con algoritmos de vuelta atrás.

INTRODUCCIÓN

La *recursividad (recursión)* es aquella propiedad que posee un método por la cual puede llamarse a sí mismo. Aunque se 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; sin embargo, en muchas circunstancias, el uso de la recursión permite a los programadores especificar soluciones naturales, sencillas, que serían, en caso contrario, difíciles de resolver. Por esta causa, la recursión es una herramienta poderosa e importante en la resolución de problemas y en la programación. Diversas técnicas algorítmicas utilizan la recursión, como los algoritmos *divide y vence* y los algoritmos de *vuelta atrás*.

5.1. LA NATURALEZA DE LA RECURSIVIDAD

Los programas examinados hasta ahora, generalmente estructurados, se componen de una serie de métodos que se llaman de modo disciplinado. En algunos problemas es útil disponer de métodos que se llamen a sí mismos. Un método *recursivo* es aquel que se llama a sí mismo, bien directamente o bien indirectamente, a través de otro método. La recursividad es un tópico importante examinado frecuentemente en cursos que estudian la resolución de algoritmos y en cursos relativos a estructuras de datos.

En este libro se dará una importancia especial a las ideas conceptuales que soportan la recursividad. En matemáticas existen numerosas funciones que tienen carácter recursivo; de igual modo, numerosas circunstancias y situaciones de la vida ordinaria tienen carácter recursivo. Piense, por ejemplo, en la búsqueda de "Sierra de Lupiana" en páginas web, puede ocurrir que aparezcan direcciones (enlaces) que lleven a otras páginas y éstas, a su vez, a otras nuevas y así hasta completar todo lo relativo a la búsqueda inicial.

Un método que tiene sentencias entre las que se encuentra al menos una que llama al propio método se dice que es *recursivo.* Así, supongamos que se dispone de dos métodos `metodo1` y `metodos2`. La organización de una aplicación no recursiva adoptaría una forma similar a ésta:

```
metodo1 (...)
{
     ...
}

metodo2 (...)
{
     ...
     metodo1(); //llamada al metodo1
     ...
}
```

Con una organización recursiva, se tendría esta situación:

```
metodo1(...)
{
     ...
     metodo1();
     ...
}
```

Ejemplo 5.1

Algoritmo recursivo de la función matemática que suma los n primero números enteros positivos.

Como punto de partida, se puede afirmar que para `n = 1` se tiene que la suma `S(1) = 1`. Para `n = 2` se puede escribir `S(2) = S(1) + 2;` en general, y aplicando la inducción matemática, se tiene:

```
S(n) = S(n-1) + n
```

Se está definiendo la función suma `S()` respecto de sí misma, eso sí, siempre para un caso más pequeño. `S(2)` respecto a `S(1)`, `S(3)` respecto a `S(2)` y, en general `S(n)` respecto a `S(n-1)`.

El algoritmo que determina la suma de modo *recursivo* ha de tener presente una condición de salida o una condición de parada. Así, en el caso del cálculo de `S(6)`; la definición es `S(6) = = 6 + S(5)`, a su vez `S(5)` es `5 + S(4)`, este proceso continúa hasta `S(1) = 1` por definición. En matemáticas la definición de una función en términos de sí misma se denomina definición **inductiva** y, de forma natural, conduce a una implementación recursiva. El **caso base** `S(1) = 1` es esencial dado que se detiene, potencialmente, una cadena de llamadas recursivas. Este caso base o condición de salida debe fijarse en cada solución recursiva. La implementación del algoritmo es:

```
long sumaNenteros (int n)
{
   if (n == 1)
      return 1;
   else
      return n + sumaNenteros(n - 1);
}
```

Ejemplo 5.2

Definir la naturaleza recursiva de la ***serie de Fibonacci:*** *0, 1, 1, 2, 3, 5, 8 ,13, 21 ...*

Se observa en esta serie que comienza con 0 y 1, y tiene la propiedad de que cada elemento es la suma de los dos elementos anteriores, por ejemplo:

```
0 + 1 = 1
1 + 1 = 2
2 + 1 = 3
3 + 2 = 5
5 + 3 = 8
...
```

Entonces se puede establecer que :

```
fibonacci(0) = 0
fibonacci(1) = 1
...
fibonacci(n) = fibonacci(n - 1) + fibonacci(n - 2)
```

y la definición recursiva será :

```
fibonacci(n) = n                                        si n = 0 o n = 1
fibonacci(n) = fibonacci(n - 1) + fibonacci(n - 2) si n > = 2
```

Obsérvese que la definición recursiva de los números de Fibonacci es diferente de las definiciones recursivas del factorial de un número y del producto de dos números. Así:

```
fibonacci(6) = fibonacci(5) + fibonacci(4)
```

o, lo que es igual, `fibonacci(6)` ha de aplicarse en modo recursivo dos veces, y así sucesivamente.

El algoritmo iterativo equivalente es:

```
if (n == 0 || n == 1)
   return n;
fibinf = 0;
fibsup = 1;
for (i = 2; i <= n; i++)
{
   int x;
   x = fibinf;
   fibinf = fibsup;
   fibsup = x + fibinf;
}
return (fibsup);
```

El tiempo de ejecución del algoritmo crece linealmente con `n` ya que el bucle es el término dominante. Se puede afirmar que `t(n)` es `O(n)`.

El algoritmo recursivo es:

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

En cuanto al tiempo de ejecución del algoritmo recursivo, ya no es tan elemental establecer una cota superior. Observe que, por ejemplo, para calcular `fibonacci(6)` se calcula, recursívamente, `fibonacci(5)` y, cuando termine éste, `fibonacci(4)`. A su vez, el cálculo de `fibonacci(5)` supone calcular `fibonacci(4)` y `fibonacci(3)`; se está repitiendo el cálculo de `fibonacci(4)`, es una pérdida de tiempo. Por inducción matemática se puede demostrar que el número de llamadas recursivas crece exponencialmente, *t(n)* es $O(2^n)$.

> ***A tener en cuenta***
>
> La formulación recursiva de una función matemática puede ser muy ineficiente sobre todo si se repiten cálculos realizados anteriormente. En estos casos el algoritmo iterativo, aunque no sea tan evidente, es notablemente más eficiente.

5.2. MÉTODOS RECURSIVOS

Un método **recursivo** es un método que se invoca a sí mismo de forma directa o indirecta. En **recursión directa,** el código del método `f()` contiene una sentencia que invoca a `f()`, mientras

que en **recursión indirecta,** el método `f()` invoca a un método `g()` que a su vez invoca al método `p()`, y así sucesivamente hasta que se invoca de nuevo al método `f()`.

Un requisito para que un algoritmo recursivo sea correcto es que no genere una secuencia infinita de llamadas sobre sí mismo. Cualquier algoritmo que genere una secuencia de este tipo no puede terminar nunca. En consecuencia la definición recursiva debe incluir una *condición de salida*, que se denomina **componente base,** en el que *f(n)* se defina directamente (es decir, no recursivamente) para uno o más valores de *n*.

En definitiva, debe existir una *"forma de salir"* de la secuencia de llamadas recursivas. Así, en el algoritmo que calcula la suma de los n primeros enteros:

$$S(n) = \begin{cases} 1 & n = 1 \\ n + S(n-1) & n > 1 \end{cases}$$

la condición de salida o componente base es *S (n) = 1* para *n = 1.*

En el caso del algoritmo recursivo de la serie de Fibonacci:

$F_0 = 0, \quad F_1 = 1; \; F_n = F_{n-1} + F_{n-2} \qquad$ *para* $n > 1$

$F_0 = 0$ y $F_1 = 1$ constituyen el componente base o condiciones de salida, y $F_n = F_{n-1} + F_{n-2}$ es el componente recursivo. Un método recursivo correcto debe incluir un componente base o condición de salida ya que, en caso contrario, se produce una recursión infinita.

A tener en cuenta

Un método es recursivo si se llama a sí mismo, directamente o bien indirectamente a través de otro método `g()`. Es necesario contemplar un caso base que determina la salida de las llamadas recursivas.

Ejercicio 5.1

Escribir un método recursivo que calcule el factorial de un número n y un programa que pida un número entero y escriba su factorial..

La *componente base* del método recursivo que calcula el factorial es que n = 0 o incluso n = 1, ya que en ambos caso el factorial es 1. El problema se resuelve recordando la definición expuesta anteriormente del factorial:

```
n! = 1              si n = 0   o n = 1    (componente base)
n! = n(n - 1)       si n > 1
```

En la implementación no se realiza tratamiento de error, que puede darse en el caso de calcular el factorial de un número negativo.

```
import java.io.*;

public class Factorial
{
```

```
   public static void main(String[] ar)throws IOException
   {
      int n;
      BufferedReader entrada = new BufferedReader(
                        new InputStreamReader(System.in));
      do {
         System.out.print("Introduzca número n: ");
         n = Integer.parseInt(entrada.readLine());
      }while (n < 0);
      System.out.println("\n \t" + n + "!= " + factorial(n));
   }
   static long factorial (int n)
   {
      if (n <= 1)
         return 1;
      else
      {
         long resultado = n * factorial(n - 1);
         return resultado;
      }
   }
}
```

5.2.1. Recursividad indirecta: métodos mutuamente recursivos

La recursividad indirecta se produce cuando un método llama a otro, que eventualmente terminará llamando de nuevo al primer método.

Ejercicio 5.2

Mostrar por pantalla el alfabeto, utilizando recursión indirecta.

El método `main()` llama a **`metodoA()`** con el argumento '`Z`' (la última letra del alfabeto). Este examina su parámetro *c,* si *c* está en orden alfabético después que '`A`', llama a **`metodoB(),`** que inmediatamente invoca a **`metodoA()`** pasándole un parámetro predecesor de *c.* Esta acción hace que **`metodoA()`** vuelva a examinar c, y nuevamente llame a **`metodoB()`**. Las llamadas continúan hasta que *c* sea igual a '`A`'. En este momento, la recursión termina ejecutando **`System.out.print()`** veintiséis veces y visualiza el alfabeto, carácter a carácter.

```
public class Alfabeto
{
   public static void main(String [] a)
   {
      System.out.println();
      metodoA('Z');
      System.out.println();
   }
   static void metodoA(char c)
   {
      if (c > 'A')
```

```
        metodoB(c);
        System.out.print(c);
    }
    static void metodoB(char c)
    {
        metodoA(--c);
    }
}
```

5.2.2. Condición de terminación de la recursión

Cuando se implementa un método recursivo, es preciso considerar una condición de terminación ya que, en caso contrario, continuaría indefinidamente llamándose a sí mismo y llegaría un momento en que la pila que registra las llamadas se desbordaría. En consecuencia, en cualquier método recursivo se necesita establecer la *condición de parada* de las llamadas recursivas y evitar indefinidas llamadas. Por ejemplo, en el caso del método `factorial()` definido anteriormente, la condición de parada ocurre cuando `n` es `1` o `0`, ya que en ambos casos el factorial es `1`. Es importante que cada llamada suponga un acercamiento a la condición de parada, porque en el método factorial cada llamada supone un decrecimiento del entero `n` lo que supone estar más cerca de la condición `n == 1`.

En el Ejercicio 5.2 se muestra una recursión mutua entre `metodoA()` y `metodoB()`, la condición de parada es que `c == 'A'`, y cada llamada mutua supone un acercamiento a la letra `'A'`.

> ***A tener en cuenta***
>
> En un algoritmo recursivo, se entiende por caso base el que se resuelve sin recursión, directamente con unas pocas sentencias elementales. El caso base se ejecuta cuando se alcanza la condición de parada de llamadas recursivas. Para que funcione la recursión el progreso de las llamadas debe tender a la condición de parada.

5.3. RECURSIÓN *VERSUS* ITERACIÓN

Se han estudiado varios métodos que se pueden implementar fácilmente, bien de modo recursivo, bien de modo iterativo. En esta sección se comparan los dos enfoques y se examinan las razones por las que el programador puede elegir un enfoque u otro según la situación específica.

Tanto la iteración como la recursión se basan en una estructura de control: *la iteración utiliza una estructura repetitiva y la recursión utiliza una estructura de selección*. Tanto la iteración como la recursión implican repetición: la iteración utiliza explícitamente una estructura repetitiva mientras que la recursión consigue la repetición mediante llamadas repetidas al método. La iteración y la recursión implican cada una un test de terminación *(condición de parada)*. La iteración termina cuando la condición del bucle no se cumple, mientras que la recursión termina cuando se reconoce un caso base o se alcanza la condición de parada.

La recursión tiene muchas desventajas. Se invoca repetidamente al mecanismo de llamadas a métodos y, en consecuencia, se necesita un tiempo suplementario para realizar cada llamada.

Esta característica puede resultar cara en tiempo de procesador y espacio de memoria. Cada llamada recursiva produce una nueva creación y copia de las variables de la función, esto consume más memoria e incrementa el tiempo de ejecución. Por el contrario, la iteración se produce dentro de un método, de modo que las operaciones suplementarias en la llamada al método y en la asignación de memoria adicional son omitidas.

Entonces, ¿cuáles son las razones para elegir la recursión? La razón fundamental es que existen numerosos problemas complejos que poseen naturaleza recursiva y, en consecuencia, son más fáciles de implementar con algoritmos de este tipo. Sin embargo, en condiciones críticas de tiempo y de memoria; es decir, cuando el consumo de tiempo y memoria sean decisivos o concluyentes para la resolución del problema, la solución a elegir debe ser, normalmente, la iterativa.

A tener en cuenta

Cualquier problema que se pueda resolver recursivamente tiene, al menos, una solución iterativa utilizando una pila. Un enfoque recursivo se elige, normalmente, con preferencia a un enfoque iterativo cuando resulta más natural para la resolución del problema y produce un programa más fácil de comprender y de depurar. Otra razón para elegir una solución recursiva es que una solución iterativa puede no ser clara ni evidente.

Consejo de programación

Se ha de evitar utilizar recursividad en situaciones de rendimiento crítico o exigencia de altas prestaciones en tiempo y en memoria, ya que las llamadas recursivas emplean tiempo y consumen memoria adicional. No es conveniente el uso de una llamada recursiva para sustituir un simple bucle.

Ejemplo 5.3

Dado un número natural n, obtener la suma de los dígitos de que consta. Presentar un algoritmo recursivo y otro iterativo.

El ejemplo ofrece una muestra clara de comparación entre la resolución de modo iterativo y de modo recursivo. Se asume que el número es natural y que, por tanto, no tiene signo. La suma de los dígitos se puede expresar:

```
suma = suma(n/10) + modulo (n,10)   para n > 9
suma = n                            para n ≤ 9, caso base
```

Para, por ejemplo, `n = 259`:

```
suma = suma(259/10)   +  modulo (259,10)      → 2 + 5 + 9 = 16
         ↓
suma = suma(25/10)    +  modulo (25,10)       → 2 + 5    ↑
         ↓
suma = suma(2/10)     +  modulo (2,10)        → 2 ↑
```

El caso base, el que se resuelve directamente, es `n ≤ 9` y, a su vez, es la condición de parada.

Solución recursiva

```
int sumaRecursiva(int n)
{
   if (n <= 9)
      return n;
   else
      return sumaRecursiva(n/10) + n%10;
}
```

Solución iterativa

La solución iterativa se construye con un bucle *mientras,* repitiendo la acumulación del resto de dividir `n` por `10` y actualizando `n` en el cociente. La condición de salida del bucle es que `n` sea menor o igual que `9`.

```
int sumaIterativa(int n)
{
   int suma = 0;
   while (n > 9)
   {
      suma += n%10;
      n /= 10;
   }
   return (suma+n);
}
```

5.3.1. Directrices en la toma de decisión iteración/recursión

1. Considérese una solución recursiva sólo cuando una solución iterativa *sencilla* no sea posible.
2. Utilícese una solución recursiva sólo cuando la ejecución y eficiencia de la memoria de la solución esté dentro de límites aceptables, considerando las limitaciones del sistema.
3. Si son posibles las dos soluciones, iterativa y recursiva, la solución recursiva siempre requerirá más tiempo y espacio debido a las llamadas adicionales a los métodos.
4. En ciertos problemas, la recursión conduce a soluciones que son mucho más fáciles de leer y de comprender que su alternativa iterativa. En estos casos, los beneficios obtenidos con la claridad de la solución suelen compensar el coste extra (en tiempo y memoria) de la ejecución de un programa recursivo.

Consejo de programación

Un método recursivo que tiene la llamada recursiva como última sentencia *(recursión final)* puede transformarse fácilmente en iterativa reemplazando la llamada mediante un bucle condicional que chequee el caso base.

5.3.2. Recursión infinita

La iteración y la recursión pueden producirse infinitamente. Un bucle infinito ocurre si la prueba o test de continuación de bucle nunca se vuelve falsa; una recursión infinita ocurre si la etapa de recursión no reduce el problema en cada ocasión, de modo que converja sobre el caso base o condición de salida.

La **recursión infinita** significa que cada llamada recursiva produce otra llamada recursiva y ésta, a su vez, otra llamada recursiva, y así para siempre. En la práctica, dicho método se ejecutará hasta que la computadora agote la memoria disponible y se produzca una terminación anormal del programa.

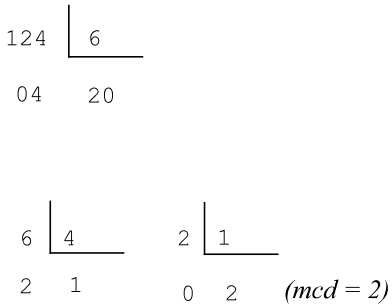
El flujo de control de un método recursivo requiere tres condiciones para una terminación normal:

- Un test para detener (o continuar) la recursión *(condición de salida o caso base).*
- Una llamada recursiva (para continuar la recursión).
- Un caso final para terminar la recursión.

Ejemplo 5.4

Deducir cual es la condición de salida del método `mcd()`*, que calcula el mayor denominador común de dos números enteros,* `b1` *y* `b2` *(el* ***mcd,*** *máximo común divisor, es el entero mayor que divide a ambos números).*

Supongamos dos número, `6` y `124`; el procedimiento clásico para hallar el *mcd* es la obtención de divisiones sucesivas entre ambos números (`124` entre `6`); si el resto no es `0`, se divide el número menor (`6`, en el ejemplo) por el resto (`4`, en el ejemplo), y así sucesivamente hasta que el resto sea `0`.



	20	1	2
124	6	4	2
4	2	0	

mcd = 2

En el caso de `124` y `6`, donde el mcd es `2`, la condición de salida es que el resto sea cero. El algoritmo del *mcd* entre dos números *m* y *n* es:

- `mcd(m,n)` *es* `n` *si* `n <= m` *y* `n` *divide a* `m`
- `mcd(m,n)` *es* `mcd(n, m)` *si* `m < n`
- `mcd(m,n)` *es* `mcd(n, resto de m divido por n)` *en caso contrario.*

El método recursivo:

```
static int mcd(int m, int n)
{
   if (n <= m && m % n == 0)
      return n;
   else if (m < n)
      return mcd(n, m);
   else
      return mcd(n, m % n);
}
```

5.4. ALGORITMOS *DIVIDE Y VENCERÁS*

Una de las técnicas más importantes para la resolución de muchos problemas de computadora es la denominada *divide y vencerás*. El diseño de algoritmos basados en esta técnica consiste en transformar (dividir) un problema de tamaño *n* en problemas más pequeños, de tamaño menor que *n,* pero similares al problema original, de modo que resolviendo los subproblemas y combinando las soluciones se pueda construir fácilmente una solución del problema completo *(vencerás)*.

Normalmente, el proceso de división de un problema en otros de tamaño menor va a dar lugar a que se llegue al *caso base,* cuya solución es inmediata. A partir de la obtención de la solución del problema para el caso base, se combinan soluciones que amplían el tamaño del problema resuelto, hasta que el problema original queda también resuelto.

Por ejemplo, se plantea el problema de dibujar un segmento que está conectado por los puntos en el plano `(x1,y1)` y `(x2,y2)`. El problema puede descomponerse así: determinar el punto medio del segmento, dibujar dicho punto y *dibujar los dos segmentos mitad obtenidos al dividir el segmento original por el punto mitad*. El tamaño del problema se ha reducido a la mitad, el hecho de dibujar un segmento se ha transformado en dibujar dos segmentos con un tamaño de justamente la mitad. Sobre cada segmento mitad se vuelve aplicar el mismo procedimiento, de tal forma que llega un momento en que, a base de dividir el segmento, se alcanza uno de longitud cercana a cero, se ha llegado al caso base, y se dibuja un punto. Cada tarea realiza las mismas acciones, por lo que se puede plantear con llamadas recursivas al proceso de dibujar el segmento cada vez con un tamaño menor, exactamente la mitad.

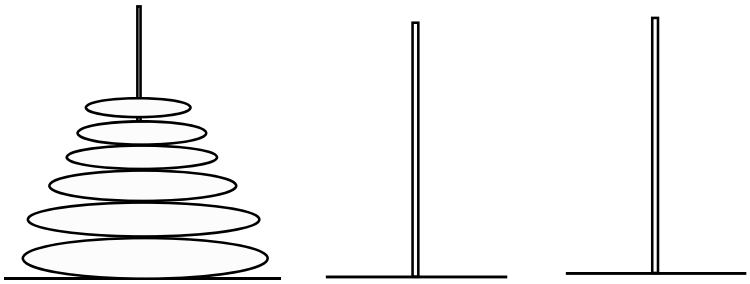
Un algoritmo *divide y vencerás* se define de manera recursiva, de tal modo que se llama a sí mismo sobre un conjunto menor de elementos. Normalmente, se implementan con dos llamadas recursivas, cada una con un tamaño menor. Se alcanza el *caso base* cuando el problema se resuelve directamente.

> ***Norma***
>
> Un algoritmo *divide y vencerás* consta de dos partes. La primera, *divide recursivamente* el problema original en subproblemas cada vez mas pequeños. La segunda, soluciona *(vencerás)* el problema dando respuesta a los subproblemas. Desde el caso base se empieza a combinar soluciones de subproblemas hasta que queda resuelto el problema completo.

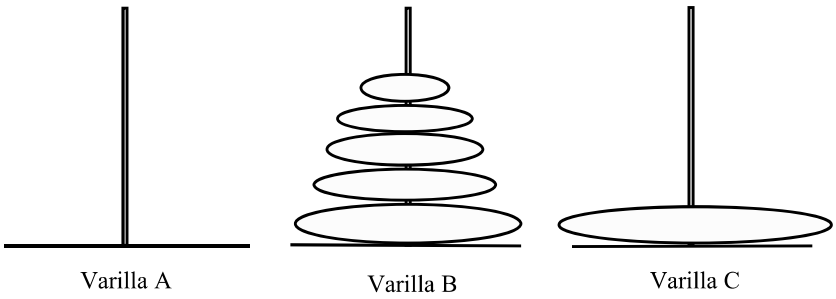
Problemas clásicos resueltos mediante recursividad son las Torres de Hanoi, el método de búsqueda binaria, la ordenación rápida, la ordenación por mezclas, etc.

5.4.1. Torres de Hanoi

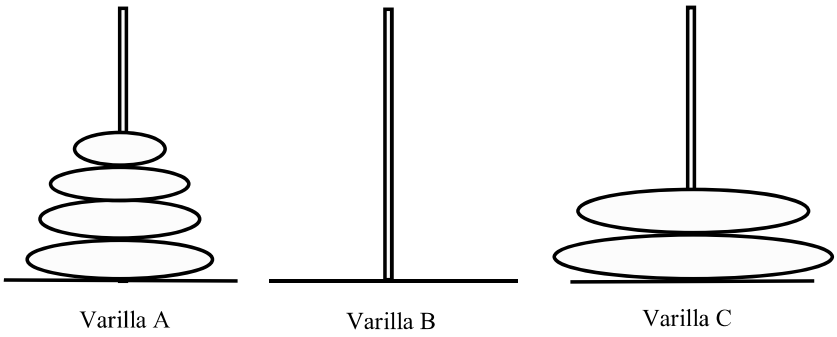
Este juego (un algoritmo clásico) tiene sus orígenes en la cultura oriental y en una leyenda sobre el Templo de Brahma, cuya estructura simulaba una plataforma metálica con tres varillas y discos en su interior. El problema en cuestión suponía la existencia de 3 varillas (A, B y C) o postes en los que se alojaban discos (*n* discos) que se podían trasladar de una varilla a otra libremente, pero con una condición: cada disco era ligeramente inferior en diámetro al que estaba justo debajo de él.



Se ilustra el problema con tres varillas con seis discos en la varilla A, y se desea trasladar a la varilla C conservando la condición de que cada disco sea ligeramente inferior en diámetro al que tiene situado debajo de él. Por ejemplo, se pueden cambiar cinco discos de golpe de la varilla A a la varilla B, y el disco más grande a la varilla C. Ya ha habido una transformación del problema en otro de menor tamaño, se ha divido el problema original.



Ahora el problema se centra en pasar los cinco discos de la varilla B a la varilla C. Se utiliza un método similar al anterior, pasar los cuatro discos superiores de la varilla B a la varilla A y, a continuación, se pasa el disco de mayor tamaño de la varilla B a la varilla C, y así sucesivamente. El proceso continúa del mismo modo, siempre dividiendo el problema en dos de menor tamaño, hasta que finalmente se queda un disco en la varilla B, que es *el caso base* y, a su vez, la condición de parada.



La solución del problema es claramente recursiva. Además, con las dos partes mencionadas anteriormente: división recursiva y solución a partir del caso base.

Diseño del algoritmo

El juego consta de tres varillas (alambres) denominadas *varilla inicial, varilla central* y *varilla final.*

En la varilla inicial se sitúan `n` discos que se apilan en orden creciente de tamaño con el mayor en la parte inferior. El objetivo del juego es mover los `n` discos desde la varilla inicial a la varilla final. Los discos se mueven uno a uno con la condición de que un disco mayor nunca puede ser situado encima de un disco más pequeño. El método `hanoi()` declara las varillas o postes como datos de tipo `char`. En la lista de parámetros, el orden de las variables es:

```
varinicial,  varcentral,  varfinal
```

e 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 el caso base, se resuelve directamente moviendo el único disco desde la varilla inicial a la varilla final. El algoritmo es el siguiente:

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 `varcentral` utilizando `varfinal` como auxiliar.
   - **1.3** Mover el disco `n` desde `varinicial` a `varfinal`.
   - **1.4** Mover `n - 1` discos desde `varcentral` a `varfinal` utilizando como auxiliar `varinicial`.

Es decir, si `n` es 1, se alcanza *el caso base,* 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, uno de los cuales es la condición de salida.

Las figuras 5.1 a 5.6 muestran el algoritmo anterior:

Etapa 1: Mover *n-1* discos desde varilla inicial (A).

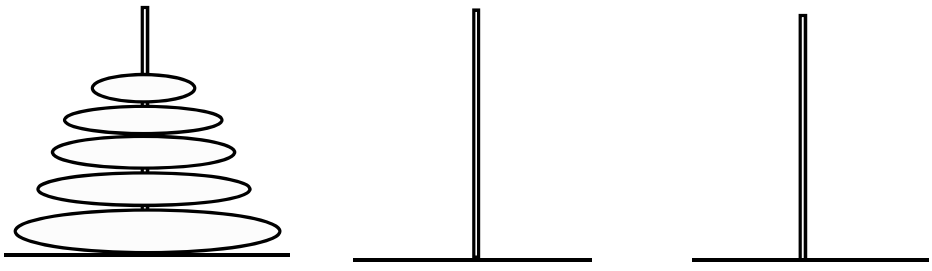


Figura 5.1 Situación inicial

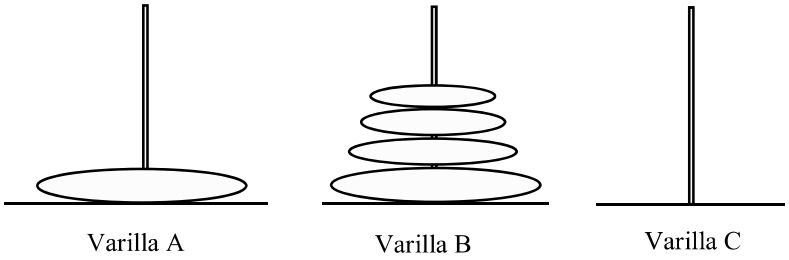


Figura 5.2 Después del movimiento

Etapa 2: Mover un disco desde A a C.

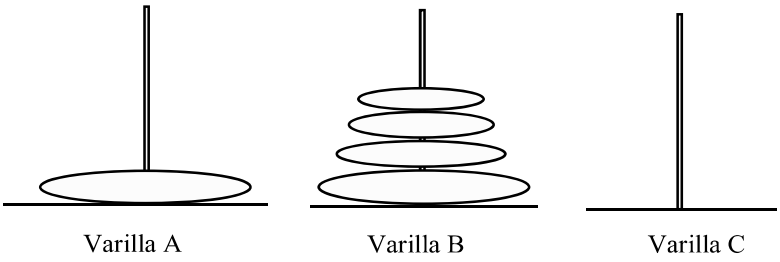


Figura 5.3 Situación de partida de etapa 2

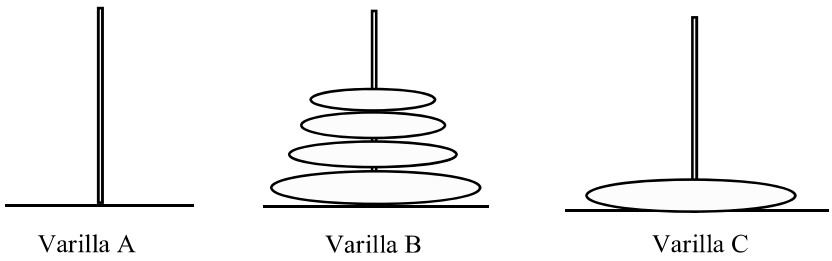


Figura 5.4 Después de la etapa 2

Etapa 3: Mover discos desde B a C.

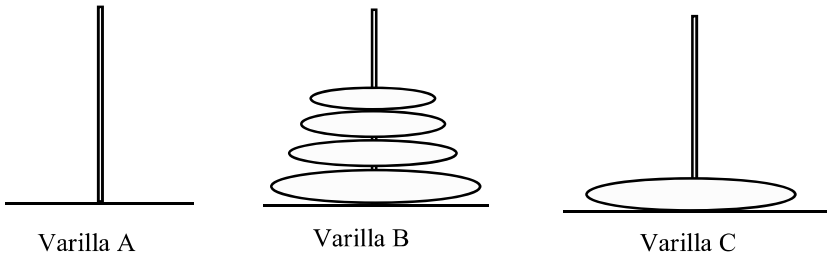


Figura 5.5 Antes de la etapa 3

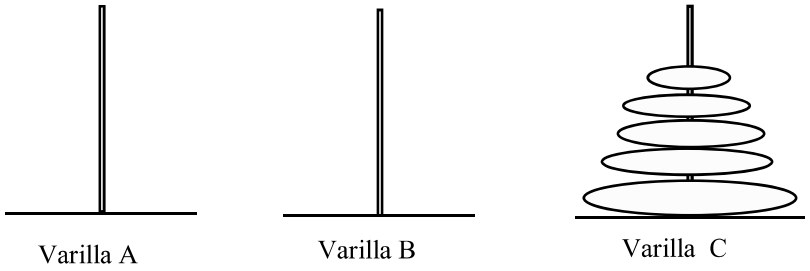


Figura 5.6 Después de la etapa 3

La primera etapa en el algoritmo mueve *n - 1* discos desde la varilla inicial a la varilla central utilizando la varilla final como almacenamiento temporal. Por consiguiente, el orden de parámetros en la llamada recursiva es *varinicial, varfinal* y varcentral.

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

La segunda etapa, simplemente mueve el disco mayor desde la varilla inicial a la varilla final:

```
System.out.println("Mover disco " + n + " desde varilla " + varinicial
                    + " a varilla " + varfinal);
```

La tercera etapa del algoritmo mueve *n - 1* discos desde la varilla central a la varilla final utilizando varinicial para almacenamiento temporal. Por consiguiente, el orden de parámetros en la llamada al método recursivo es: *varcentral, varinicial y varfinal.*

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

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 los movimientos que transferirán los `n` discos desde la varilla inicial, `'A'`, a la varilla final, `'C'`. La codificación es:

```
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);
   }
}
```

Análisis del algoritmo Torres de Hanoi

Fácilmente se puede encontrar el árbol de llamadas recursivas para `n = 3` discos, que en total realiza `7 = (2^3 - 1)` llamadas a `hanoi()` y escribe `7` movimientos de disco. El problema de `5` discos se resuelve con `31 = (2^5 - 1)` llamadas y `31` movimientos. Si se supone que `T(n)` es el número de movimientos para `n` discos, teniendo en cuenta que el método realiza dos llamadas con `n-1` discos y mueve el disco `n`, se tiene la recurrencia:

$$T(n) = \begin{cases} 1 & \text{si } n = 1 \\ 2T(n-1) + 1 & \text{si } n > 1 \end{cases}$$

Los sucesivos valores que toma `T`, según `n`: `1, 3, 7, 15, 31, 63 ... 2^n -1`.

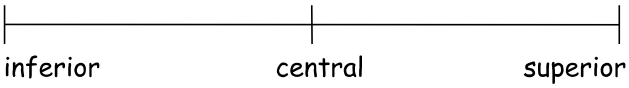
En general, el número de movimientos requeridos para resolver el problema de *n* discos es $2^n - 1$. Cada llamada al método requiere la asignación e inicialización de un área local de datos en la memoria; por ello, el tiempo de computadora se incrementa exponencialmente con el tamaño del problema.

> ***Nota de ejecución***
>
> La complejidad del algoritmo que resuelve el problema de las Torres de Hanoi es exponencial. Por consiguiente, a medida que crece n, aumenta exponencialmente el tiempo de ejecución de la función.

5.4.2. Búsqueda binaria

La búsqueda binaria es un método de localización de una clave especificada dentro de una lista o array ordenado de *n* elementos que realiza una exploración de la lista hasta que se encuentra o se decide que no se está en la lista. El algoritmo de búsqueda binaria se puede describir recursívamente aplicando la técnica *divide y vencerás*.

Se tiene una lista ordenada `a[]` con un límite inferior y un límite superior. Dada una clave (valor buscado), comienza la búsqueda en la posición central de la lista (índice central).

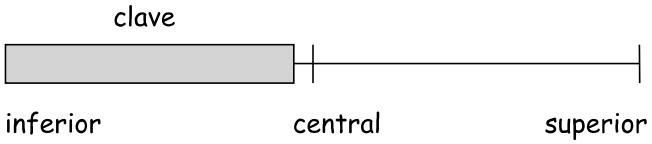


`central = (inferior + superior)/2` Comparar `a[central]` y `clave`

Si hay coincidencia (se encuentra la clave), se tiene la condición de terminación que permite detener la búsqueda y devolver el índice central. En caso contrario (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 superior" (a la derecha de la posición central). El problema de la búsqueda se ha dividido en la mitad, el tamaño de la lista donde buscar es la mitad del

tamaño anterior.El tamaño de la lista se reduce cada vez a la mitad, así hasta que se encuentre el elemento, o bien la lista resultante esté vacía.

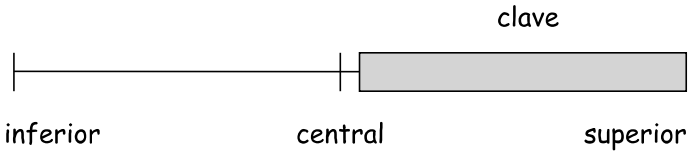
1. Si `clave < a[central]`, el valor buscado sólo puede estar en la mitad izquierda de la lista con elementos en el rango, `inferior` a `central - 1`.



Búsqueda en sublista izquierda

`[inferior .. central - 1]`

2. Si `clave > a[central]`, el valor buscado sólo puede estar en la mitad superior de la lista con elementos en el rango de índices, `central + 1` a `superior`.



3. La búsqueda continúa en sublistas más y más pequeñas, exactamente la mitad, con dos llamada recursivas: una se corresponde con la sublista inferior y la otra, con la sublista superior. El algoritmo termina 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` es la condición de salida o terminación sin éxito, y el algoritmo devuelve el índice -1.

Se codifica en un método `static` que formaría parte de una clase de utilidades.

```
static int busquedaBR(double a[], double clave,
                      int inferior, int superior)
{
   int central;
   if (inferior > superior) // no encontrado
      return -1;
   else
   {
      central = (inferior + superior)/2;
      if (a[central] == clave)
         return central;
      else if (a[central] < clave)
         return busquedaBR(a, clave, central+1, superior);
      else
         return busquedaBR(a, clave, inferior, central-1);
   }
}
```

Análisis del algoritmo

El peor de los casos que hay que contemplar en una búsqueda es que ésta no tenga éxito. El tiempo del algoritmo recursivo de búsqueda binaria depende del número de llamadas. Cada llamada reduce la lista a la mitad, y progresivamente se llega a que el tamaño de la lista es unitario; en la siguiente llamada, el tamaño es `0 (inferior > superior)` y termina el algoritmo. La secuencia siguiente describe los sucesivos tamaños:

$n/2, n/2^2, n/2^3, \ldots, n/2^t = 1$

tomando logaritmo, $t = \log n$. Por tanto, el número de llamadas es $\lfloor \log n \rfloor + 1$. Cada llamada es de complejidad constante, así que se puede afirmar que la complejidad, en término de notación O, es logarítmica $O(\log n)$.

5.5. *BACKTRACKING*, ALGORITMOS DE VUELTA ATRÁS

La resolución de algunos problemas exige probar sistemáticamente todas las posibilidades que pueden existir para encontrar una solución. Los algoritmos de *vuelta atrás* utilizan la recursividad para probar cada una de las posibilidades de encontrar la solución.

Este método de resolución de problemas recurre a realizar una búsqueda exhaustiva, sistemática, de una posible solución al problema planteado. Descompone el proceso de búsqueda o tanteo de una solución en tareas parciales, cada una de las cuales realiza las mismas acciones que la tarea anterior, por eso se expresa, frecuentemente, en forma recursiva.

Por ejemplo, el problema de determinar los sucesivos saltos que debe de hacer un caballo de ajedrez, desde una posición inicial cualquiera (con su forma típica de moverse) para que pase por todas las casillas de un tablero vacío; la tarea parcial es realizar un salto válido, en cuanto que dicho salto esté dentro de las coordenadas del tablero y no haya pasado ya por la casilla destino. En este problema, que se resolverá posteriormente, todas las posibilidades son los ocho posibles saltos que el caballo puede realizar desde una casilla dada.

El proceso general de los algoritmos de *vuelta atrás* se contempla como un método de prueba o búsqueda, que gradualmente construye tareas básicas y las inspecciona para determinar si conducen a la solución del problema. Si una tarea no conduce a la solución, prueba con otra tarea básica hasta agotar todas las posibilidades. Es una prueba sistemática hasta llegar a la solución, o bien determinar que no hay solución por haberse agotado todas las opciones que probar.

> ***Nota de ejecución***
>
> Una de las características principales de los algoritmos de *vuelta atrás* es la búsqueda exhaustiva, con todas las posibilidades, de soluciones parciales que conduzcan a la solución del problema. Otra característica es la *vuelta atrás*, en el sentido de que si una solución o tarea parcial no conduce a la solución global del problema se *vuelve atrás* para probar (ensayar) con otra de las posibilidades de solución parcial.

Modelo de los algoritmos de vuelta atrás

El esquema general de la técnica de resolución de problemas *backtracking*:

```
procedimiento ensayarSolucion
inicio
```

```
<inicializar cuenta de posibles selecciones>
repetir
   <tomar siguiente selección (tarea)>
   <determinar si selección es valida>
   si válido entonces
      <anotar selección>
      si <problema solucionado>  entonces
         exito = true
      si no
         ensayarSolución {llamada para realizar otra tarea}
            {vuelta atrás, se analiza si se ha alcanzado la solución del problema}
         si no exito entonces
            <borrar anotación> {el bucle se encarga de probar con otra selección}
         fin _ si
      fin _ si
   fin _ si
hasta (exito) o (<no más posibilidades>)
fin
```

Este esquema tendrá las variaciones necesarias para adaptarlo a la casuística del problema a resolver. A continuación, se aplica esta forma de resolución a diversos problemas, como el del *salto del caballo* y el de las *ocho reinas.*

5.5.1. Problema del Salto del caballo

En un tablero de ajedrez de `n x n` casillas, se tiene un caballo situado en la posición inicial de coordenadas (x_0, y_0). El problema consiste en encontrar, si existe, un circuito que permita al caballo pasar exactamente una vez por cada una de las casillas de tablero, teniendo en cuenta los movimientos (saltos) permitidos a un caballo de ajedrez.

Este es un ejemplo clásico de problema que se resuelve con el esquema del algoritmo de *vuelta atrás*. El problema consiste en buscar la secuencia de saltos que tiene que dar el caballo, partiendo de una casilla cualquiera, para pasar por cada una de las casillas del tablero. Se da por supuesto que el tablero está vacío, no hay figuras excepto el caballo. Lo primero que hay que tener en cuenta es que el caballo, desde una casilla, puede realizar hasta 8 movimientos, como muestra la Figura 5.7.

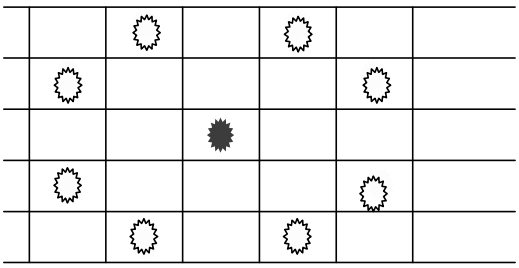


Figura 5.7 Los ocho posibles saltos del caballo

Por consiguiente, el número de *posibles selecciones* en este problema es ocho. La tarea básica, anteriormente se ha denominado *solución parcial*, en que se basa el problema consiste en que el caballo realice un nuevo movimiento entre los ocho posibles.

Los ocho posibles movimientos de caballo se obtienen sumando a su posición actual, `(x,y)`, unos desplazamientos relativos, que son:

```
d = {(2,1), (1,2), (-1,2), (-2,1), (-2,-1), (-1,-2), (1,-2), (2,-1)}
```

Por ejemplo, si el caballo se encuentra en la casilla `(3,5)`, los posibles movimientos que puede realizar:

```
{(5,6), (4,7), (2,7), (1,6), (1,4), (2,3), (4,3), (5,4)}
```

No siempre será posible realizar los ocho movimientos, se debe comprobar que la casilla destino esté dentro del tablero y también que el caballo no haya pasado previamente por ella. En caso de ser posible el movimiento, se anota, guardando el número del salto realizado.

```
tablero[x][y] = numeroSalto {número del Salto}
nuevaCoorX    = x + d[k][1]
nuevaCoorY    = y + d[k][2]
```

Las llamadas a la función que resuelve el problema transmiten las nuevas coordenadas y el nuevo salto a realizar son:

```
saltoCaballo(nuevaCoorX, nuevaCoorY, numeroSalto+1)
```

La condición que determina que el problema se ha resuelto está ligada con el objetivo que se persigue, y en este problema es que se haya pasado por las n^2 casillas; en definitiva, que el caballo haya realizado $n^2 - 1$ (63) saltos. En ese momento, se pone a `true` la variable `exito`.

¿Qué ocurre si se agotan los ocho posibles movimientos sin alcanzar la solución? Se vuelve al movimiento anterior, *vuelta atrás,* se borra la anotación para ensayar con el siguiente movimiento. Y si también se han agotado los movimientos, ocurre lo mismo: se vuelve al que fue su movimiento anterior para ensayar, si es posible, con el siguiente movimiento de los ocho posibles.

Representación del problema

De manera natural, el tablero se representa mediante una matriz de enteros para guardar el número de salto en el que pasa el caballo. En Java, los arrays siempre tienen como índice inferior 0, se ha preferido reservar una fila y una columna más para el tablero y así representarlo más fielmente.

```
final int N = 8;
final int n = (N+1);

int [][] tablero = new int[n][n];
```

Una posición del tablero puede contener:

`tablero[x,y]=`

`0`	por la casilla `(x,y)`	no pasó el caballo.
`i`	por la casilla `(x,y)`	pasó el caballo en el salto `i`.

Los desplazamientos relativos que determinan el siguiente salto del caballo se guardan en una matriz constante con los valores predeterminados.

Parámetros de la llamada recursiva

Los únicos parámetros que va a tener el método recursivo que resuelve el problema son los necesarios para que el caballo realice un nuevo movimiento, que son las coordenadas actuales y el número de salto del caballo. El *flag* que indica si se ha completado el problema será una variable de la clase en la que se implementa el problema.

Codificación del algoritmo *Salto del caballo*

En la clase *CaballoSaltador* se declaran los atributos que representan el tamaño y el tablero (matriz). También se define el atributo `exito`, de tipo lógico, que será puesto a *verdadero* si el método recursivo que resuelve el problema encuentra una solución. Además, la matriz `SALTO` guarda los ocho desplazamientos relativos a una posición dada del caballo.

El método interno `saltoDelCaballo()` realiza todo el trabajo mediante llamadas recursivas; el método *público* `resolverProblema()` llama a `saltoCaballo()` con las coordenadas iniciales que se establecen en el constructor. Éste valida que las coordenadas estén dentro del tablero, inicializa el tablero y establece el valor de `exito` a *falso*.

```
class CaballoSaltador
{
   static final int N = 8;
   static final int n = (N+1);
   private int [][] tablero = new int[n][n];
   private boolean exito;
   private int [][]SALTO = {{2,1}, {1,2}, {-1,2}, {-2,1},
                            {-2,-1}, {-1,-2}, {1,-2}, {2,-1}};
   private int x0, y0;
      // constructor
   public CaballoSaltador(int x, int y) throws Exception
   {
      if ((x < 1) || (x > N) ||
         (y < 1) || (y > N))
         throw new Exception("Coordenadas fuera de rango");
      x0 = x;
      y0 = y;
      for(int i = 1; i<= N; i++)
         for(int j = 1; j<= N; j++)
            tablero[i][j] = 0;
      tablero[x0][y0] = 1;
      exito = false;
   }
   public boolean resolverProblema()
   {
      saltoCaballo(x0, y0, 2);
      return exito;
   }
   private void saltoCaballo(int x, int y, int i)
   {
      int nx, ny;
      int k;
      k = 0; // inicializa el conjunto de posibles movimientos
      do {
         k++;
         nx = x + SALTO[k-1][0];
```

```
            ny = y + SALTO[k-1][1];
            // determina si nuevas coordenadas son aceptables
            if ((nx >= 1) && (nx <= N) && (ny >= 1) && (ny <= N)
                          &&
                (tablero[nx][ny] == 0))
            {
               tablero[nx][ny]= i; // anota movimiento
               if (i < N*N)
               {
                  saltoCaballo(nx, ny, i+1);
                  // se analiza si se ha completado la solución
                  if (!exito)
                  { // no se alcanzó la solución
                     tablero[nx][ny] = 0;  // se borra anotación
                  }
               }
               else
                  exito = true;   // tablero cubierto
            }
         } while ((k < 8) && !exito);
      }
      //muestra por pantalla los pasos del caballo
      void escribirTablero()
      {
         for(int i = 1; i <= N; i++)
         {
            for(int j = 1; j <= N; j++)
               System.out.print(tablero[i][j] + "  ");
            System.out.println();
         }
      }
   }
```

Ejercicio 5.3

Escribir una aplicación que, a partir de una casilla inicial del caballo, resuelva el problema del salto del caballo.

La aplicación lee las coordenadas iniciales, crea un objeto de la clase `CaballoSaltador` y llama al método `resolverProblema()`. Se escribe el contenido del tablero si ha resuelto el problema.

```
import java.io.*;

public class Caballo
{
   public static void main(String[] ar)
   {
      int x, y;
      boolean solucion;
      BufferedReader entrada = new BufferedReader(
                           new InputStreamReader(System.in));
      try {
         System.out.println("Posicion inicial del caballo. ");
         System.out.print(" x = ");
```

```
            x = Integer.parseInt(entrada.readLine());
            System.out.print(" y = ");
            y = Integer.parseInt(entrada.readLine());
            CaballoSaltador miCaballo = new CaballoSaltador(x,y);
            solucion = miCaballo.resolverProblema();
            if (solucion)
               miCaballo.escribirTablero();
         }
         catch (Exception m)
         {
            System.out.println("No se pudo probar el algoritmo, " + m);
         }
      }
   }
```

Nota de ejecución

Los algoritmos que hacen una búsqueda exhaustiva de la solución tienen tiempos de ejecución muy elevados. La complejidad es exponencial, puede ocurrir, que si no se ejecuta en un ordenador potente, este se bloquee.

5.5.2. Problema de las ocho reinas

El problema se plantea de la forma siguiente: *dado un tablero de ajedrez (8 x 8 casillas), hay que situar ocho reinas de forma que ninguna de ellas pueda actuar ("comer") sobre ninguna de las otras.* Éste es otro de los ejemplos del uso de los métodos de búsqueda sistemática y de los algoritmos de *vuelta atrás*. Hay que recordar, en primer lugar, la regla del ajedrez respecto de los movimientos de la reina. Ésta puede moverse a lo largo de la columna, fila y diagonales de la casilla donde se encuentra.

Antes de empezar a aplicar los pasos que siguen estos algoritmos, se van a *podar* posibles movimientos de las reinas. Cada columna puede contener una y sólo una reina, la razón de esta restricción es inmediata: si en la columna 1 se encuentra la reina 1, y en esta columna se sitúa otra reina, entonces se *atacan* mutuamente. De las `n x n` casillas que puede ocupar una reina, se limita su ubicación a las 8 casillas de la columna en la que se encuentra, de tal forma que si se numeran las reinas de 1 a 8, entonces la reina `i` se sitúa en alguna de las casillas de la columna `i`.

Lo primero a considerar, a la hora de aplicar el algoritmo de vuelta atrás, es la *tarea básica* que exhaustivamente se realiza; la propia naturaleza del problema de las 8 reinas determina que la tarea sea colocar la reina `i`, *tantear* si es posible ubicar la reina número `i`, para lo que hay 8 alternativas, que son las 8 posibles filas de la columna `i`. La comprobación de que una selección es válida tiene que hacerse investigando que en la fila seleccionada y en las dos diagonales en las que una reina puede atacar no haya otra reina colocada anteriormente. En cada paso se amplía la solución parcial al problema, ya que aumenta el número de reinas colocadas.

La segunda cuestión es analizar la *vuelta atrás*. ¿Qué ocurre si no se alcanza la solución, si no se es capaz de colocar las 8 reinas? Se borra la ubicación de la reina, la fila donde se ha colocado, y se ensaya con la siguiente fila válida.

La realización de cada tarea supone ampliar el número de reinas colocadas hasta llegar a la solución completa, o bien determinar que no es posible colocar la reina actual en las 8 posibles filas. Entonces, en la *vuelta atrás* se coloca la reina actual en otra fila válida para realizar un nuevo *tanteo*.

Representación del problema

En cuanto a los tipos de datos para representar las reinas en el tablero, hay que tener en cuenta que éstos permitan realizar las acciones de verificar que *no se coman* las reinas lo más eficientemente posible.

Debido a que el objetivo del problema es encontrar la fila en la que se sitúa la reina de la columna `i`, se define el array entero, `reinas[]`, de tal forma que cada elemento contenga el índice de fila donde se sitúa la reina, o bien cero. El número de reina, `i`, es a su vez el índice de columna dentro de la cual se puede colocar entre los ocho posibles valores de fila.

El array que se define tiene una posición más, y así la numeración de las reinas coincide con el índice del array.

```
final int N = 8;
final int n = (N+1);
int [] reinas = new int[n];
```

Verificación de la ubicación de una reina

Al colocar la reina `i` en la fila `j` hay que comprobar que no se ataque con las reinas colocadas anteriormente. Un reina `i` situada en la fila `j` de la columna `i` ataca, o es atacada, por otra reina que esté en la misma fila, o bien por otra que esté en la misma diagonal `(i-j)`, o en la diagonal `(i+j)`. El método `valido()` realiza esta comprobación mediante un bucle con tantas iteraciones como reinas situadas actualmente.

Parámetros de la llamada recursiva

El método recursivo tiene los parámetros necesarios para situar una nueva reina. En este problema sólo es necesario el número de reina que se va a colocar; el array con las reinas colocadas actualmente es un atributo de la clase, al igual que la variable lógica que se pone a *verdadero* si han sido puestas todas las reinas.

Codificación del algoritmo de las 8 reinas

En la clase `OchoReinas` se declaran como atributos los datos que se han establecido para representar el problema. El método `solucionReinas()` es la interfaz de la clase para resolver el problema. Este llama a `ponerReina()` que realiza todo el trabajo de resolución con llamadas recursivas.

```
public boolean solucionReinas()
{
   solucion = false;
   ponerReina(1);
   return solucion;
}
private void ponerReina(int i)
{
   int j;
   j = 0; // inicializa posibles movimientos
   do {
      j++;
      reinas[i] = j; // prueba a colocar reina i en fila j,
                     // a la vez queda anotado el movimiento
      if (valido(i))
      {
         if (i < N) // no completado el problema
```

```
         {
            ponerReina(i+1);
            // vuelta atrás
            if (!solucion)
               reinas[i] = 0;
         }
         else // todas las reinas colocadas
            solucion = true;
      }
   } while(!solucion && (j < 8));
}
private boolean valido(int i)
{
      /* Inspecciona si la reina de la columna i es atacada por
         alguna reina colocada anteriormente */
   int r;
   boolean libre;
   libre = true;
   for (r = 1; r <= i-1; r++)
   {
      // no esté en la misma fila
      libre = libre && (reinas[i] != reinas[r]);
      // no esté en alguna de las dos diagonales
      libre = libre && ((i + reinas[i]) != (r + reinas[r]));
      libre = libre && ((i - reinas[i]) != (r - reinas[r]));
   }
   return libre;
}
```

5.6. SELECCIÓN ÓPTIMA

En los problemas del *Salto de caballo* y *Ocho reinas* se ha aplicado la estrategia de *vuelta atrás* para encontrar una solución del problema y, a continuación, dejar de hacer llamadas recursivas. Para resolver el problema de encontrar las variaciones se ha seguido una estrategia similar, con la diferencia de que no se detiene al encontrar una variación, sino que sigue haciendo llamadas hasta *agotar todas las posibilidades,* todos los elementos.

Por consiguiente, un sencillo cambio al esquema algorítmico permite encontrar *todas las soluciones* a los problemas planteados; todo consiste en que el algoritmo muestre la solución encontrada en lugar de activar una variable `boolean` y siempre realice *la vuelta atrás, desanotando lo anotado* antes de hacer la llamada recursiva. Por ejemplo, el problema de las *Ocho reinas* encuentra una solución, pero realizando el cambio indicado se encuentran todas las formas de colocar la reinas en el tablero sin que se "coman".

Ejercicio 5.4

Dado un conjunto de pesos, se quieren escribir todas las combinaciones de ellos que totalicen un peso dado.

El problema se resuelve probando con todas las combinaciones posibles de los objetos disponibles. Cada vez que la suma de los pesos de una combinación sea igual al peso dado, se escribe en los objetos de los que consta. El problema se resuelve modificando el método `seleccion()` de

la clase `SeleccionObjeto`[1]. El atributo `encontrado` ya no es necesario, el bucle tiene que iterar con todos los objetos, cada vez que `suma` sea igual al peso objetivo, `objetivo`, se llama a `escribirSeleccion()`. Ahora, en la *vuelta atrás* siempre se borra de la bolsa la anotación del objeto realizada, para así probar con los `n` objetos.

```
public void seleccion(int candidato, int suma)
{

   while (candidato < n)
   {
      candidato++;
      if ((suma + objs[candidato-1]) <= objetivo)
      {
         k++; // es anotado
         bolsa[k-1] = candidato - 1;
         suma += objs[candidato-1];
         if (suma < objetivo)    // ensaya con siguiente objeto
         {
            seleccion(candidato, suma);
         }
         else // objetos totalizan el objetivo
            escribirSeleccion();
         // se borra la anotación
          k--;
         suma -= objs[candidato-1];
      }
   }
}
```

Los problemas que se adaptan al esquema de *selección óptima* no persiguen encontrar una situación fija o un valor predeterminado, sino averiguar, del conjunto de todas las soluciones, la óptima según unas condiciones que establecen qué es lo óptimo. Por tanto, hay que probar con todas las posibilidades que existen de realizar una nueva tarea para encontrar entre todas las soluciones la que cumpla una segunda condición o requisito, la *solución óptima.*

> ***Nota***
>
> La selección óptima implica probar con todas los posibles elementos de que se disponga para así de entre todas las configuraciones que cumplan una primera condición, seleccionar la más próxima a una segunda condición, que será la selección óptima.

El esquema general del algoritmo para encontrar la *selección óptima* es:

```
procedimiento ensayarSeleccion(objeto i)
inicio
   <inicializar cuenta de posibles selecciones>
   repetir
      <tomar siguiente selección (tarea)>
      <determinar si selección es valida>
```

[1] Consultar anexo en la web: Resolución de problemas con algoritmos de vuelta atrás.

```
      si válido entonces
         <anotar selección  >
         si  <es solución>  entonces
            si <mejor(solución)> entonces
               <guardar selección>
            fin _ si
         fin _ si
          ensayarSeleccion(objeto i+1)
         <borrar anotación> {el bucle se encarga de probar con otra
                              selección }
      fin _ si
   hasta (<no más posibilidades>)
fin
```

Nota de eficiencia

El tiempo de ejecución de estos algoritmos crece muy rápidamente, y el número de llamadas recursivas crece exponencialmente (*complejidad exponencial).* Por ello, es importante considerar el hecho de evitar una llamada recursiva si se sabe que no va a mejorar la actual selección óptima *(poda de las ramas en el árbol de llamadas)*.

5.6.1. Problema del viajante

El ejemplo del problema del viajante es el que mejor explica la *selección óptima: el viajante (piense en un representante de joyería) tiene que confeccionar la maleta, seleccionando entre n artículos aquellos cuyo valor sea máximo (lo óptimo es que la suma de valores sea máximo) y su peso no exceda de una cantidad, la que puede sustentar la maleta.*

Por ejemplo, la maleta de seguridad para llevar joyas es capaz de almacenar 1,5 kg y se tienen los objetos, representados por el par `(peso, valor)`: `(115 g, 100€), (90 g, 110€), (50 g, 60€), (120 g, 110€)`... Se pretende hacer una selección que no sobrepase los 1,5 kg y que la suma del valor asociado a cada objeto sea el máximo.

Para resolver el problema se generan todas las selecciones posibles con los `n` objetos disponibles. Cada selección debe cumplir la condición de no superar el peso máximo prefijado. Cada vez que se alcance una selección, se pregunta si es mejor que cualquiera de las anteriores y, en caso positivo, se guarda como la actual *mejor selección*. En definitiva, consiste en una *búsqueda exhaustiva*, un *tanteo sistemático* con los `n` objetos del problema; cada tanteo realiza la tarea de probar si incluir el objeto `i` va a dar lugar a una mejor selección, y también si la exclusión del objeto `i` dará lugar a una *mejor selección*.

```
si solucion entonces
   si mejor(solucion) entonces
      optimo = solucion
```

Tarea básica

El objetivo del problema es que el valor que se pueda conseguir sea máximo; por esa razón, se considera que el valor más alto que se puede alcanzar es la suma de los valores de cada objeto; posteriormente, al excluir un objeto de la selección se ha de restar el valor que tiene.

La tarea básica en esta *búsqueda sistemática* es investigar si un objeto `i` es adecuado incluirlo en la selección actual. Será adecuado si el peso acumulado más el del objeto `i` no supera al peso de la maleta. En el caso de que sea necesario excluir el objeto `i` (*vuelta atrás* de las llamadas recursivas) de la selección actual, el criterio para seguir con el proceso de selección es que el valor total todavía alcanzable, después de esta exclusión, no sea menor que el valor óptimo (máximo) encontrado hasta ahora. La inclusión de un objeto supone un incremento del peso de la selección actual con el peso del objeto. A su vez, excluir un objeto de la selección supone que el valor que puede alcanzar la selección tiene que ser decrementado en el valor del objeto excluido. Cada tarea realiza las mismas acciones que la tarea anterior, por ello se expresa recursívamente.

La prueba de si es mejor selección se hace ensayando con todos los objetos disponibles, una vez que se alcance el último, el objeto `n`, es cuando se determina si el valor asociado a la selección es el mejor, el óptimo.

Norma

En el proceso de *selección óptima* se prueba con la inclusión de un objeto `i` y con la exclusión de `i`. Para hacer más eficiente el algoritmo, se hace una *poda* de aquellas selecciones que no van a ser mejores; por ello, antes de probar con la exclusión se determina si la selección en curso puede alcanzar un mejor valor; si no es así ¿para qué ir por esa rama si no se va a conseguir una mejor selección?.

Representación de los datos

Se supone que el viajante tiene `n` objetos. Cada objeto tiene asociado el par `<peso, valor>`, por ello sendos arrays almacenan los datos de los objetos. La selección actual y la óptima van a tratarse como dos conjuntos de objetos; realmente, como la característica de cada objeto es el índice en el array de pesos y valores, los conjuntos incluirán únicamente el índice del objeto. Cada conjunto se representa por un atributo entero, el cardinal, y un array de índices.

Codificación

La clase `Optima` agrupa la representación y el algoritmo recursivo `seleccionOptima()`. El valor máximo que pueden alcanzar los objetos es la suma de los valores de cada uno, está representado por el atributo de la clase `totalValor`. El valor óptimo alcanzado en el proceso transcurrido está en el atributo `mejorValor`. Se inicializa al valor más bajo alcanzable, cero, así al menos habrá una selección que supere a dicho valor.

Los parámetros del método recursivo son los necesarios para realizar una nueva tarea: `i`, número del objeto a probar su inclusión; `pt`, peso de la selección actual y `va`, valor máximo alcanzable por la selección actual.

Ejercicio 5.5

Escribir la clase `Optima` *y una aplicación que solicite como datos de entrada los objetos (valor, peso) y el peso máximo que puede llevar el viajante.*

El método que resuelve el problema va a estar *oculto*, privado, de tal forma que la interfaz, `selecionOptima()`, inicializa los dos conjuntos, que representan a la selección actual y la óptima a *conjuntoVacio,* simplemente poniendo el cardinal a cero. En este método se suman los

valores asociados a cada objeto, que es el máximo valor alcanzable, y se asigna a `totalValor`. También se pone a cero el atributo `mejorValor` y llama al método recursivo.

```
class Optima
{
   private int n;
   private int []pesoObjs;
   private int []valorObjs;
   private int cardinalActual;
   private int [] actual;
   private int cardinalOptimo;
   private int [] optimo;
   private int pesoMaximo;
   private int mejorValor;
   // constructor, leer n y los objetos
   public Optima()
   {
      // realizar la entrada de los objetos
   }
   public void seleccionOptima()
   {
      int totalValor = 0;
      actual = new int[n];
      optimo = new int[n];
      mejorValor = 0;
      cardinalActual = 0; cardinalOptimo = 0
      for (int j = 0; j < n; j++)
         totalValor += valorObjs[j];
      seleccionOptima(1, 0, totalValor);
      escribirSeleccion();
   }
   private void seleccionOptima(int i, int pt, int va)
   {
      int valExclusion;
      if (pt + pesoObjs[i-1] <= pesoMaximo) // objeto i se incluye
      {
         cardinalActual++;
         actual[cardinalActual-1] = i-1; // índices del objeto
         if (i < n)
            seleccionOptima(i+1, pt + pesoObjs[i-1], va);
         else // los n objetos probados
            if (va > mejorValor) // nuevo optimo
            {
               mejorValor = va;
               System.arraycopy(actual,0,optimo,0,cardinalActual);
               cardinalOptimo = cardinalActual;
            }
            cardinalActual-- ; //vuelta atrás, ensaya exclusión de objeto i
      }
         /* proceso de exclusión del objeto i para seguir
                        la búsqueda sistemática con el objeto i+1 */
      valExclusion = va - valorObjs[i-1]; /* decrementa el valor
                                             del objeto excluido */
      if (valExclusion > mejorValor)     /* se puede alcanzar una mejor
                                  selección, sino poda la búsqueda */
         if (i < n)
            seleccionOptima(i+1, pt, valExclusion);
```

```
            else
            {
               mejorValor = valExclusion;
               System.arraycopy(actual,0,optimo,0,cardinalActual);
               cardinalOptimo = cardinalActual;
         }
      }
   }
```

RESUMEN

Un método o función se dice que es recursivo si tiene una o más sentencias que son llamadas a sí mismas. La recursividad puede ser directa o indirecta; ésta ocurre cuando el método `f()` llama a `p()` y ésta, a su vez, llama a `f()`. La recursividad es una alternativa a la iteración en la resolución de algunos problemas matemáticos aunque, en general, es preferible la implementación iterativa debido a que es más eficiente. Los aspectos más importantes a tener en cuenta en el diseño y construcción de métodos recursivos son los siguientes:

- Un algoritmo recursivo contiene dos tipos de casos: uno o más casos que incluyen al menos una llamada recursiva y uno o más casos de terminación o parada del problema, en los que éste se soluciona sin ninguna llamada recursiva, sino con una sentencia simple. De otro modo, un algoritmo recursivo debe tener dos partes: una parte de terminación en la que se deja de hacer llamadas, es el caso base, y una llamada recursiva con sus propios parámetros.
- Muchos problemas tienen naturaleza recursiva y la solución más fácil es mediante un método recursivo. De igual modo, aquellos problemas que no entrañen una solución recursiva se deberán seguir resolviendo mediante algoritmos iterativos.
- Las métodos con llamadas recursivas utilizan memoria extra en las llamadas; existe un límite en las llamadas, que depende de la memoria de la computadora. En caso de superar este límite ocurre un error de *overflow*.
- Cuando se codifica un método recursivo, se debe comprobar siempre que tiene una condición de terminación, es decir, que no se producirá una recursión infinita. Durante el aprendizaje de la recursividad es usual que se produzca ese error.
- Para asegurarse de que el diseño de un método recursivo es correcto, se deben cumplir las siguientes tres condiciones:
  1. No existir recursión infinita. Una llamada recursiva puede conducir a otra llamada recursiva y ésta conducir a otra, y así sucesivamente; cada llamada debe aproximarse más a la condición de terminación.
  2. Para la condición de terminación, el método devuelve el valor correcto para ese caso.
  3. En los casos que implican llamadas recursivas: si cada uno de los métodos devuelve un valor correcto, entonces el valor final devuelto por el método es el valor correcto.
- Una de las técnicas más utilizadas en la resolución de problemas es la denominada *"divide y vence"*. La implementación de estos algoritmos se puede realizar con métodos recursivos.

- Los algoritmos del tipo *vuelta atrás* o *backtracking* se caracterizan por realizar una búsqueda sistemática, exhaustiva, de la solución. Prueba con todas las posibilidades que se tienen para realizar una tarea que vaya encaminada hacia la solución. Cada tarea se expresa por una llamada recursiva, los elementos de la tarea se apuntan (se guardan). En la *vuelta atrás,* retorno de la llamada recursiva, se determina si se alcanzó la solución y en caso contrario, se borra la anotación para probar de nuevo con otra posibilidad.
- La selección óptima se puede considerar como una variante de los algoritmos de *vuelta atrás*, en la que se buscan no sólo los elementos que cumplan una condición, sino que éstos sean los mejores, según el criterio que se haya establecido como *mejor*.

EJERCICIOS

5.1. Convierta el siguiente método iterativo en recursivo. El método calcula un valor aproximado de *e*, la base de los logaritmos naturales, sumando las series

1 + 1/1 ! + 1/2 ! + ... + 1/n !

hasta que los términos adicionales no afecten a la aproximación

```
static public double loge()
{
   double enl, delta, fact;
   int n;
   enl = fact = delta = 1.0;
   n = 1;
   do
   {
      enl += delta;
      n++;
      fact * = n:
      delta = 1.0 / fact;
   } while (enl != enl + delta);
   return enl;
}
```

5.2. Explique por qué el siguiente método puede producir un valor incorrecto cuando se ejecute:

```
static public long factorial (long n)
{
   if (n == 0 || n == 1)
      return 1;
   else
      return n * factorial (--n);
}
```

5.3. ¿Cuál es la secuencia numérica generada por el método recursivo `f()` en el listado siguiente si la llamada es `f(5)`?

```
long f(int n)
{
   if (n == 0 || n == 1)
      return 1;
  else
      return 3 * f(n - 2) + 2 * f(n - 1);
}
```

5.4. Escribir un método recursivo `int vocales(String)` para calcular el número de vocales de una cadena.

5.5. Proporcionar métodos recursivos que representen los siguientes conceptos:

a) El producto de dos números naturales.

b) El conjunto de permutaciones de una lista de números.

5.6. Suponer que la función matemática G está definida recursivamente de la siguiente forma:

$$G(x, y) = \begin{cases} \dfrac{1}{G(x-y+1)} & \text{si } x \le y \\ 2x - y & \text{si } x > y \end{cases}$$

siendo x, y enteros positivos. Encontrar el valor de: (a) G(8, 6); (b) G(100, 10).

5.7. Escribir un método recursivo que calcule la función de Ackermann definida de la siguiente forma:

$A(m, n) = n + 1$ *si* $m = 0$
$A(m, n) = A(m - 1, 1)$ *si* $m > 0, y\ n = 0$
$A(m, n) = A(m - 1, A(m, n - 1))$ *si* $m > 0, y\ n > 0$

5.8. ¿Cuál es la secuencia numérica generada por el método recursivo siguiente, si la llamada es `f(8)`?

```
long f (int n)
{
   if(n == 0||n ==1)
      return 1;
   else if (n % 2 == 0)
      return 2 + f(n - 1);
   else
      return 3 + f(n - 2);
}
```

5.9. ¿Cuál es la secuencia numérica generada por el método recursivo siguiente?

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

5.10. El elemento mayor de un array entero de *n-elementos* se puede calcular recursivamente. Suponiendo que el método:

```
static public int max(int x, int y);
```

devuelve el mayor de dos enteros x e y, definir el método

```
int maxarray(int [] a, int n);
```

que utiliza recursión para devolver el elemento mayor de `a`

Condición de parada: `n == 1`

Incremento recursivo: `maxarray = max(max(a[0]...a[n-2]),a[n-1])`

5.11. Escribir un método recursivo,

```
int product(int[]v, int b);
```

que calcule el producto de los elementos del array `v` mayores que `b`.

5.12. El Ejercicio 5.6 define recursivamente una función matemática. Escribir un método que no utilice la recursividad para encontrar valores de la función.

5.13. La resolución recursiva de las Torres de Hanoi ha sido realizada con dos llamadas recursivas. Volver a escribir la solución con una sola llamada recursiva.

Nota: Sustituir la última llamada por un bucle *repetir-hasta*.

5.14. Aplicar el esquema de los algoritmos *divide y vence* para qué dadas las coordenadas (x,y) de dos puntos en el plano, que representan los extremos de un segmento, se dibuje el segmento.

5.15. Escribir un método recursivo para transformar un número entero en una cadena con el signo y los dígitos de que consta: `String entroaCadena(int n).`

PROBLEMAS

5.1. La expresión matemática *C(m, n)* en el mundo de la teoría combinatoria de los números, representa el número de combinaciones de *m* elementos tomados de *n* en *n* elementos.

$$C(m, n) = \frac{m!}{n!\,(m - n)!}$$

Escribir una aplicación en la que se dé entrada a los enteros *m, n* y calcule *C(m, n)* donde *n!* es el factorial de *n*.

5.2. Un palíndromo es una palabra que se escribe exactamente igual leída en un sentido o en otro. Palabras tales como *level, deed, ala*, etc. son ejemplos de palíndromos. Aplicar un algoritmo *divide y vence* para determinar si una palabra es palíndromo. Escribir un método recursivo que implemente el algoritmo. Escribir una aplicación en la que se lea una cadena hasta que ésta sea un palíndromo.

5.3. Escribir una aplicación en la que un método recursivo liste todos los subconjuntos de `n` letras para un conjunto dado de `m` letras, por ejemplo para `m = 4` y `n = 2`.

`[A, C, E, K]` → `[A, C], [A, E], [A, K], [C, E,], [C, K], [E, K]`

Nota: el número de subconjuntos es $C_{4,2}$.

5.4. El problema de las ocho reinas se ha resuelto en este capítulo de tal forma que en el momento de encontrar una solución se detiene la búsqueda de más soluciones. Modificar el algoritmo de tal forma que el método recursivo escriba todas las soluciones.

5.5. Escribir una aplicación que tenga como entrada una secuencia de números enteros positivos (mediante una variable entera). El programa debe hallar la suma de los dígitos de cada entero y encontrar cual es el entero cuya suma de dígitos es mayor. La suma de dígitos ha de hacerse con un método recursivo.

5.6. Desarrollar un método recursivo que cuente el número de números binarios de *n-dígitos* que no tengan dos 1 en una fila. *Sugerencia:* El número comienza con un 0 o un 1. Si comienza con 0, el número de posibilidades se determina por los restantes *n-1* dígitos. Si comienza con 1, ¿cuál debe ser el siguiente?

5.7 Desarrolar una aplicación que lea un número entero positivo $n < 10$ y calcule el desarrollo del polinomio $(x + 1)^n$. Imprimir cada potencia x^i en la forma `x**i`.

Sugerencia:

$$(x + 1)^n = C_{n,n}x^n + C_{n,n-1}x^{n-1} + C_{n,n-2}x^{n-2} + \ldots + C_{n,2}x^2 + C_{n,1}x^1 + C_{n,0}x^0$$

donde $C_{n,n}$ y $C_{n,0}$ son 1 para cualquier valor de *n*.

La relación de recurrencia de los coeficientes binomiales es:

C(n, 0) = 1
C(n, n) = 1
C(n, k) = C(n-1, k-1) + C(n-1, k)

Estos coeficientes constituyen el famoso Triángulo de Pascal y será preciso definir el método que genera el triángulo

```
                1
             1     1
          1     2     1
       1     3     3     1
    1     4     6     4     1
...
```

5.8. Sea `A` una matriz cuadrada de `n x n` elementos, el determinante de `A` se puede definir de manera recursiva:

a) Si `n = 1`, entonces `Deter(A)=` $a_{1,1}$.

b) Para `n > 1`, el determinante es la suma alternada de productos de los elementos de una fila o columna elegida al azar por sus menores complementarios. A su vez, los menores complementarios son los determinantes de orden `n-1` obtenidos al suprimir la fila y la columna en que se encuentra el elemento.

Puede expresarse:

$$\text{Det(A)} = \sum_{i=1}^{n} (-1)^{i+j} * A[i,j] * \text{Det(Menor(A[i,j]))}; \text{ para cualquier columna } j$$

o

$$\text{Det(A)} = \sum_{j=1}^{n} (-1)^{i+j} * A[i,j] * \text{Det(Menor(A[i,j]))}; \text{ para cualquier columna } i$$

Se observa que la resolución del problema sigue la estrategia de los algoritmos *divide y vence*.

Escribir una aplicación que tenga como entrada los elementos de la matriz `A` y tenga como salida la matriz `A` y el determinante de `A`. Elegir la fila `1` para calcular el determinante.

5.9. Escribir una aplicación que transforme números enteros en base 10 a otro en base b, siendo ésta de 8 a 16. La transformación se ha de realizar siguiendo una estrategia recursiva.

5.10. Escribir una aplicación para resolver el problema de la subsecuencia creciente más larga. La entrada es una secuencia de n números $a_1,a_2,a_3, \ldots ,a_n$; hay que encontrar la subsecuencia mas larga $a_{i1},a_{i2}, \ldots a_{ik}$ tal que $a_{i1}<a_{i2}<a_{i3} \ldots <a_{ik}$ y que $i1<i2<i3< \ldots <ik$. El programa escribirá dicha subsecuencia.
Por ejemplo, si la entrada es `3,2,7,4,5,9,6,8,1`, la subsecuencia creciente más larga tiene longitud cinco: `2,4,5,6,8`.

5.11. El sistema monetario consta de monedas de valores $p_1,p_2,p_3, \ldots , p_n$ (orden creciente). Escribir una aplicación que tenga como entrada el valor de las `n` monedas, en orden creciente, y una cantidad `x` de cambio. Calcule:

a) El número mínimo de monedas que se necesitan para dar el cambio `x`.

b) El número de formas diferentes de dar el cambio de la cantidad `x` con la p_i monedas.

Aplicar técnicas recursivas para resolver el problema.

5.12. En un tablero de ajedrez, se coloca un alfil en la posición (x_0,y_0) y un peón en la posición `(1,j)`, siendo `1 <= j <= 8`. Se pretende encontrar una ruta para el peón que llegue a la fila 8 sin ser comido por el alfil, siendo el único movimiento permitido para el peón el de avance desde la posición `(i,j)` a la posición `(i+1,j)`. Si se encuentra que el peón está amenazado por el alfil en la posición `(i,j)`, entonces debe retroceder a la fila `1`, columna `j+1` o `j-1` `{(1,j+1), (1,j-1)}`.
Escribir una aplicación para resolver el supuesto problema. Hay que tener en cuenta que el alfil ataca por diagonales.

5.13. Dados `n` números enteros positivos, encontrar una combinación de ellos que mediante sumas o restas totalicen exactamente un valor objetivo `Z`. La aplicación debe tener como entrada los `n` números y el objetivo `Z`; la salida ha de ser la combinación de números con el operador que le corresponda.

Tener en cuenta que pueden formar parte de la combinación los `n` números o parte de ellos.

5.14. Dados n números, encontrar la combinación con sumas o restas que más se aproxime a un objetivo `Z`. La aproximación puede ser por defecto o por exceso. La entrada son los `n` números, y el objetivo y la salida son la combinación más proxima al objetivo.

5.15. Podemos emular un laberinto con una matriz `n x n` en la que los pasos libres estén representados por un carácter (el blanco por ejemplo) y los muros por otro carácter (el □ por ejemplo). Escribir una aplicación que se genere aleatoriamente un laberinto, se pidan las coordenadas de entrada (la fila será la `1`), las coordenadas de salida (la fila será la `n`) y encontrar todas las rutas que nos lleven de la entrada a la salida.

5.16. Realizar las modificaciones necesarias en el problema anterior para encontrar la ruta más corta, considerando ésta la que pasa por un menor número de casillas.

5.17. Una región castellana está formada por n pueblos dispersos. Hay conexiones directas entre algunos de ellos, y entre otros no existe conexión, aunque puede haber un camino. Escribir una aplicación que tenga como entrada la matriz que representa las conexiones directas entre pueblos, de tal forma que el elemento `M(i,j)` de la matriz sea:

$$M(i,j) = \begin{cases} 0 & \text{si no hay conexión directa entre pueblo } i \text{ y pueblo } j. \\ d & \text{hay conexión entre pueblo } i \text{ y pueblo } j \text{ de distancia } d. \end{cases}$$

Tenga también como entrada un par de pueblos `(x,y)`. La aplicación tiene que encontrar un camino entre ambos pueblos utilizando técnicas recursivas. La salida ha de ser la ruta que se ha de seguir para ir de `x` a `y` junto a la distancia de la ruta.

5.18. En el programa escrito en el ejercicio anterior, hacer las modificaciones necesarias para encontrar todos los caminos posibles entre el par de pueblos `(x,y)`.

5.19. Un número entero sin signo, m, se dice que es *dos_tres_cinco* si cumple las características:

- Todos los dígitos de m son distintos.
- La suma de los dígitos que ocupan posiciones pares es igual a la suma de los dígitos que ocupan posiciones múltiplos de tres más la suma de los dígitos que ocupan posiciones múltiplos de cinco.

Implementar una aplicación que genere todos los números enteros de cinco o más cifras que séan *dos_tres_cinco.*