8 RECURSIVIDAD

Dios mueve al jugador y éste, la pieza. ¿Qué Dios detrás de Dios la trama empieza de polvo y tiempo y sueño y agonía?

Jorge Luis Borges (El Ajedrez)

Una carpeta puede contener archivos y carpetas, que a su vez pueden contener archivos y carpetas, que también pueden contener archivos y carpetas y así... hasta que tengamos carpetas que solo tengan archivos o estén vacías. Un menú de una interfaz de usuario puede tener submenús que a su vez tengan también submenús. El clásico juguete ruso matrioshka tiene dentro otra matrioshka, que tiene dentro otra matrioshka, hasta que lleguemos a una matrioshka muy pequeña que ya no contiene más matrioshkas. El anidamiento de triángulos de la Figura 8-1 también muestra un comportamiento similar.

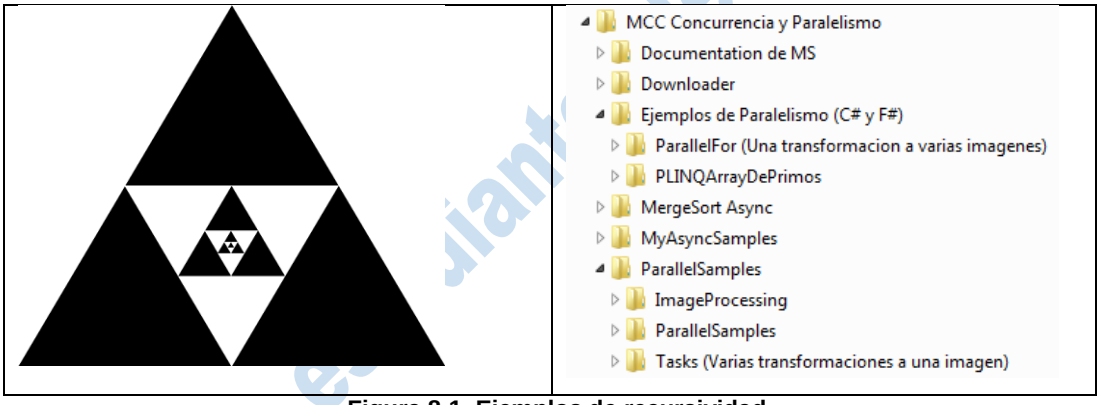

Figura 8-1. Ejemplos de recursividad

La propia naturaleza nos presenta este tipo de comportamientos: observe, por ejemplo, las ramas de un árbol de las que a su vez salen ramas, de las cuales salen ramas y así hasta terminar en hojas.

Este fenómeno se conoce como **recurrencia** o **recursividad** y juega un papel muy importante para expresar la solución algorítmica de muchos problemas y también en la forma de estructurar muchos datos (el concepto de árbol se estudia en el Capítulo 14).

Uno de los algoritmos más antiguos es el algoritmo de Euclides[1] para determinar el máximo común divisor entre dos números *a* y *b*. Este algoritmo se basa en el cumplimiento de la siguiente propiedad:

> *Para hallar el máximo común divisor de **a** y **b**, supongamos sin perder generalidad que **a** es mayor que **b**. S*ea ***d*** divisor común de **a** y ***b***; eso quiere decir que ***a = p*d***

[1] Claro que por aquel entonces a esto no se le llamaba algoritmo.

y $b = q*d$, de modo que $a - b = (p - q) * d$; lo cual quiere decir que d es también divisor de $(a - b)$. Por otro lado, si d es divisor de a - b y b, eso quiere decir que $a - b = m * d$ y $b = n * d$, de donde $a = (m - n) * d$, por lo que d también divide a a.

De lo anterior se deduce que si a, b y a-b tienen los mismos divisores, entonces tienen el mismo máximo común divisor

De modo que sin pérdida de generalidad, calcular el m.c.d. de a y b lo podemos reducir a calcular el m.c.d de $a - b$ y b, pudiéndose expresar la función mcd de la forma:

$$mcd(a,b) = \begin{cases} a & b = 0 \\ mcd(b,a) & b > a \\ mcd(a-b,b) & e.o.c. \end{cases}$$

Probablemente para el lector la función más conocida que tiene un planteamiento recursivo es la función factorial, que se plantea como:

$$N! = \begin{cases} 1 & N = 0 \\ N * (N-1)! & N > 0 \end{cases}$$

Aunque también sería fácil definir que el factorial de N es el producto de todos los números de 1 a N.

Todas estas definiciones tienen en común que se usan a su vez en sí mismas para expresarse, pero también que en cierto punto deben referirse a una situación base que ya no necesita usarse a sí misma para explicarse (la matrioshka final que ya no contiene más matrioshkas, el triángulo que ya no puede ser más pequeño según la resolución de la pantalla, la carpeta que está vacía o solo tiene archivos, la rama que solo tiene hojas, el que factorial de 0 que se define como 1).

8.1 MIS PRIMEROS CÓDIGOS RECURSIVOS

El código del Listado 8-1 calcula el factorial de un número. Note que dentro de la definición del método `Factorial` estamos llamando al propio método `Factorial`. La condicional `if (N == 0)` representa el caso base del algoritmo y su solución directa, es decir, que ya no hace otra llamada recursiva. En la próxima línea se plantea la dependencia a la propia función factorial, pero evaluada en un valor menor (`N - 1` en este caso). Es evidente que cualquier valor positivo para N producirá entonces sucesivas llamadas recursivas hasta que `N` llegue a 0 (convergencia al caso base); de lo contrario, se caería en un ciclo "infinito" de llamadas.

```
static long Factorial(int N)
{
  if (N == 0)
    return 1;
  return N * Factorial(N - 1);
}
```

Listado 8-1 Definición recursiva del método Factorial

De forma similar se puede expresar el código (Listado 8-2) para resolver el máximo común divisor (m.c.d.) entre dos números *a* y *b*. La primera condicional representa un

Draft para estudiantes de MATCOM UH

caso recursivo "básico" que resuelve operar siempre con los parámetros `a` y `b` ordenados de mayor a menor. Esto es válido, puesto que la función MCD es conmutativa. La segunda condicional evalúa el caso base (cualquier valor es divisor de 0, por lo que el MCD entre cualquier número positivo y 0 es el propio número). Por último, se expresa la llamada recursiva que resuelve el MCD "reduciendo" el valor de los parámetros con los que se llama y por tanto convergiendo al caso base.

```
static int MCD(int a, int b) {
  if (a < b)
    return MCD(b, a);
  if (b == 0)
    return a;
  return MCD(a - b, b);
}
```

Listado 8-2 Método recursivo para hallar el máximo común divisor entre dos números

8.2 LA MAGIA DE LA RECURSIVIDAD. LAS TORRES DE HANÓI

Un ejemplo que ilustra de forma sencilla el poder expresivo de la recursividad es dar solución a los movimientos del juego conocido como las torres de Hanói.

El juego consiste de 3 varillas clavadas cada una en una base y en las que se pueden colocar *N* discos de distintos tamaños con una configuración inicial como se muestra en la Figura 8-2. La regla es que sólo se puede mover un disco a la vez y no puede colocarse un disco encima de otro de menor tamaño. El objetivo es pasar en la menor cantidad de movimientos (y sin violar las reglas anteriores) todos los discos de una varilla origen (*A* en este caso) a una varilla destino (*C*).

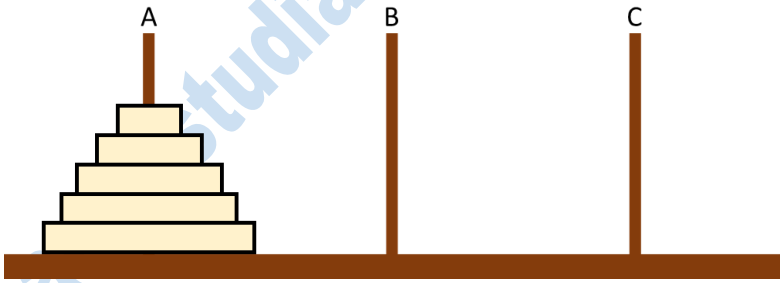

Figura 8-2 Configuración inicial de las torres de Hanoi para 5 discos

Draft para estudiantes de MATCOM UH

El juego fue propuesto en 1883 por el matemático francés Édouard Lucas, y posiblemente deba su nombre a la similitud con la arquitectura de algunas construcciones vietnamitas.

Hay una leyenda (que se cree fue un invento publicitario del propio Lucas para promocionar su juego) que cuenta que Dios, al crear el mundo, colocó tres varillas de diamante con 64 discos en la primera. También creó un monasterio con monjes, quienes tenían la tarea de resolver el problema para esta Torre divina. «Si la leyenda fuera cierta, ¿cuándo sería el fin del mundo?». La mínima cantidad de movimientos para resolver este problema es de $2^{64} - 1$; si los monjes hicieran un movimiento por segundo (y sin equivocarse), los 64 discos estarían en la tercera varilla en alrededor de 585 mil millones de años. (Tenga en cuenta que se calcula que la Tierra tiene unos 5 mil millones de años, y el Universo, unos 14 mil millones ☺)

Para resolver este problema, pensemos en "reducirlo". Es decir, se quieren mover *N* discos y no se sabe cómo... pero... si se supiera cómo mover una cantidad menor de discos, ¿sabríamos entonces cómo se podrían mover los *N* discos?

Queremos mover *N* discos de la varilla *A* a la varilla C apoyándonos en la varilla *B*. Supongamos que sabemos cómo mover *N-1* cumpliendo las reglas; entonces podemos hacer los movimientos que se ilustran en la Figura 8-3. Los *N-1* discos que están en el tope de la varilla *A* los movemos entonces a la varilla *B* (utilizando la varilla *C* como auxiliar). En el fondo de la varilla *A* ha quedado un disco de mayor tamaño que los que se están moviendo, por tanto durante todos esos movimientos no se viola nunca la regla. Una vez que ya tenemos los *N-1* discos en la varilla *B*, entonces movemos el disco que está en el fondo de la varilla *A* hacia la varilla *C*. Como hemos supuesto que sabemos mover una cantidad de discos menor que *N*, entonces podemos mover los *N-1* discos que están en la varilla *B* hacia la varilla *C* (auxiliándonos en la varilla *A*). Y como en el fondo de la varilla *C* habíamos puesto el mayor de todos, entonces durante esos movimientos tampoco se viola nunca la regla.

Como se muestra en el `static void Hanoi(int n, string orig, string dest, string aux)`

```
{
  if (n = 1)
    Console.WriteLine("Mover de " + orig + " a " + dest);
  else
  {
    Hanoi(n - 1, orig, aux, dest);
    Hanoi(1, orig, dest, aux);
    Hanoi(n - 1, aux, dest, orig);
  }
}
```

Listado 8-3, usando la recursividad es muy simple expresar la solución a este problema. El caso base de la recursividad es aquí cuando la cantidad de discos a mover es 1, en que entonces se mueve directamente el disco hacia el destino indicado y se retorna a quien llamó. Si la cantidad a mover no es 1, entonces hay dos llamadas recursivas en las que en ambas se disminuye la cantidad de discos a mover (*N-1*), de modo que en algún

Draft para estudiantes de MATCOM UH

momento se alcanzará el caso base. Es importante destacar que al regresar de la primera llamada recursiva se vuelven a usar los parámetros `n`, `orig`, `dest` y `aux`, por lo que se supone que durante el desencadenamiento de las llamadas recursivas anteriores, y su posterior retorno, no se han alterado los valores de estos parámetros. Esto es posible gracias a la forma en que ejecuta internamente la recursividad (ver epígrafe 8.4).

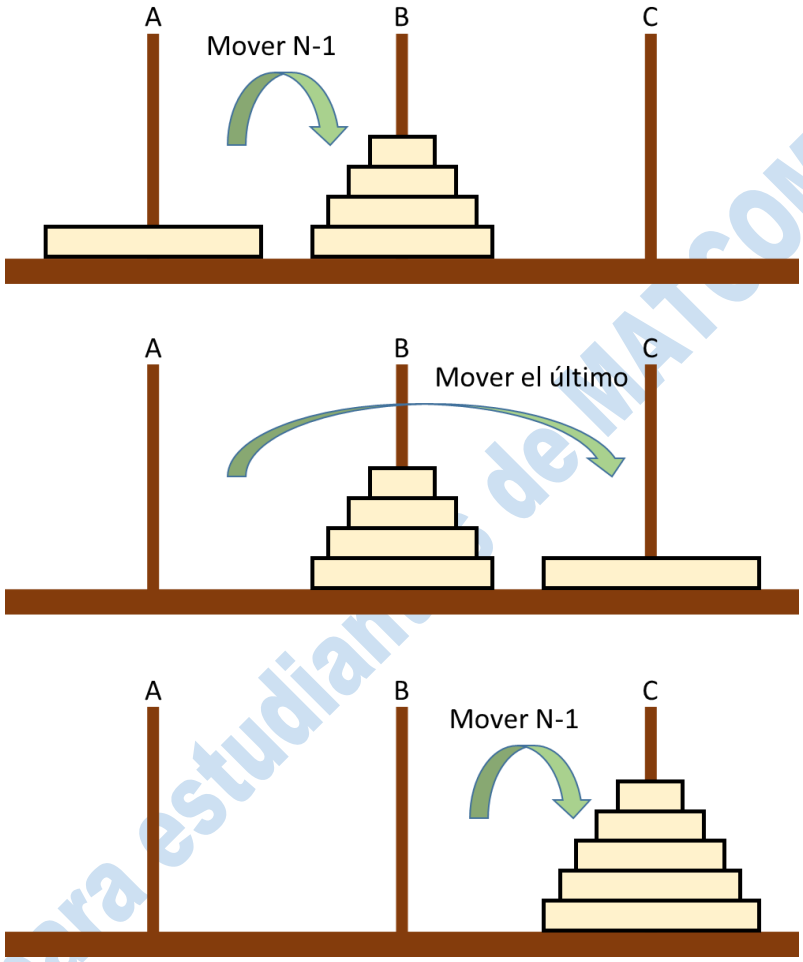

Figura 8-3 Movimiento de discos en las torres de Hanoi

```
static void Hanoi(int n, string  orig, string dest, string aux)
{
  if (n = 1)
    Console.WriteLine("Mover de " + orig + " a " + dest);
  else
  {
    Hanoi(n - 1, orig, aux, dest);
    Hanoi(1, orig, dest, aux);
    Hanoi(n - 1, aux, dest, orig);
  }
```

Draft para estudiantes de MATCOM UH

```
}
```

Listado 8-3 Método recursivo de las torres de Hanoi considerando que n es mayor que 0

8.3 SUCESIONES RECURSIVAS

Hay sucesiones numéricas en la matemática que expresan un comportamiento recursivo, es decir, en las que el cálculo del término *n*-ésimo de la sucesión se basa en el cálculo de algunos términos anteriores. En esta sección se verán algunos ejemplos de cómo estas sucesiones son útiles para solucionar algunos problemas.

8.3.1 LA SUCESIÓN DE FIBONACCI

La sucesión de Fibonacci es la conocida como la de la reproducción de conejos:

"Si una pareja de conejos adultos es capaz de reproducirse cada mes y producir una pareja descendiente (que se hace adulta a su vez al mes), ¿cuántas parejas de conejos adultos habrán al cabo de un año si una granja comienza la cría de conejos a partir de una pareja de conejos adultos?"

En la siguiente tabla se ilustra el crecimiento poblacional de los conejos.

Mes	Parejas de Adultos	Parejas de Crías	Total
1	1	0	1
2	1	1	2
3	2	1	3
4	3	2	5
5	5	3	8
6	8	5	13
7	13	8	21
8	21	13	34
9	34	21	55
10	55	34	89
11	89	55	144
12	*144*		

La columna de la derecha nos expresa la cantidad de parejas de conejos que habrá cada mes. Esta sucesión debe su nombre a que fue planteada por el italiano Leonardo de Pisa (conocido por Fibonacci).

Draft para estudiantes de MATCOM UH

Leonardo de Pisa (c. 1170 - 1250) recibió póstumamente el apodo de Fibonacci (por ser filius Bonacci, hijo de Bonacci). Su padre, el italiano Bonacci, dirigía un puesto de comercio en el norte de África (en lo que es hoy Argelia) y su hijo le ayudaba en la contabilidad.

Posiblemente abrumado por tener que hacer los engorrosos cálculos en lo que conocemos como números romanos, y habiendo conocido la superioridad de sistema de numeración indo-arábigo que empleaba una notación posicional (de base 10) y que usaba un dígito especial (el cero) para expresar el valor nulo, viajó a través de los países del Mediterráneo para aprender de los matemáticos árabes ese sistema de numeración.

En 1202, a los 32 años de edad, publicó el libro titulado Libro del Ábaco para difundir en Occidente todo lo que había aprendido. En este libro mostró la importancia del nuevo sistema de numeración aplicándolo a la contabilidad comercial, conversión de pesos y medidas, cálculo, intereses, cambio de moneda, y otras muchas aplicaciones. En sus páginas describe el cero, la notación posicional, la descomposición en factores primos, los criterios de divisibilidad. El libro fue recibido con entusiasmo en la Europa ilustrada, y tuvo un impacto profundo en el posterior pensamiento matemático europeo.

Esta sucesión se puede plantear entonces recursivamente en la forma siguiente:

$$C_{adultos}(n) = \begin{cases} 1, & n \leq 2 \\ C_{adultos}(n-1) + C_{adultos}(n-2), & x > 1 \end{cases}$$

Es decir, los dos primeros términos de la sucesión son 1, y a partir del tercero cada término es igual a la suma de los dos anteriores.

El Listado 8-4 nos muestra la forma tan sencilla en que puede plantearse este método recursivamente. Fíjese como en este caso hay dos llamadas al método.

```
static long Fibonacci(int n) {
  if (n <= 2)
    return 1;
  return Fibonacci(n - 1) + Fibonacci(n - 2);
}
```

Listado 8-4 Método recursivo para calcular el enésimo elemento de la sucesión de Fibonacci

El cálculo del término n-ésimo de la sucesión de Fibonacci se puede implementar de forma sencilla de forma iterativa haciendo un ciclo en el que se van conservando en cada iteración los dos últimos valores calculados de la sucesión (Listado 8-5).

```
public static long FibonacciIterativo( int n )
{
  long ultimo = 1, penultimo = 1;
  for (int k = 3; k <= n; k++)
  {
    long temp = penultimo;
    penultimo = ultimo;
    ultimo = ultimo + temp;
  }
  return ultimo;
}
```

Listado 8-5 Método iterativo para calcular el enésimo elemento de la sucesión de Fibonacci

Draft para estudiantes de MATCOM-UH

Invitamos al lector a medir el tiempo que demora la ejecución de ambas implementaciones. Apreciará que el método recursivo ejecutará mucho más lento[2]. No concluya que esto se debe a que la recursividad es ineficiente; lo que se sucede es que en este caso se ha hecho un uso inadecuado de la recursividad (como se verá en el Capítulo 9, en el que se trata del costo de los algoritmos).

8.3.2 EL PROBLEMA DE LA DISTRIBUCIÓN DE LOSAS, OTRA APLICACIÓN DE LA SUCESIÓN DE FIBONACCI

Una situación similar a la de la reproducción de los conejos la tenemos con el siguiente problema:

"Se tienen N *losas rectangulares de dimensiones* 1×2(*el largo es el doble del ancho*). *Se quiere cubrir un pasillo rectangular de* $2 \times$ N. *¿De cuántas formas distintas se pueden colocar las losas?"*

Para $N = 4$, las formas posibles de cubrir el pasillo se muestran en la Figura 8-4. La solución para este problema particular sería 5.

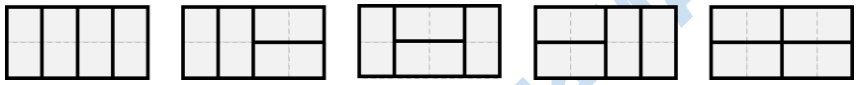

Figura 8-4. Configuraciones posibles para 4 losas

El orden en que se ubican las losas no importa, y cualquier disposición de las losas o tiene la losa más a la izquierda vertical o tiene a la izquierda dos losas horizontales. Por lo tanto, podemos contar la cantidad de formas con N losas como la suma de aquellas que empiezan con una losa vertical (quedando $N-1$ losas por poner) más aquellas que empiezan con dos losas horizontales (quedando $N-2$ losas por ubicar). Al final se puede expresar la cantidad de configuraciones con N losas como:

$$C(N) = \begin{cases} 1, & N = 1 \\ 2, & N = 2 \\ C(N-1) + C(N-2), & N > 2 \end{cases}$$

Nuevamente, una sucesión similar a la de Fibonacci, con la diferencia de que los dos primeros valores fueron 1 y 2 en lugar de 1 y 1.

Se invita al lector a solucionar el problema siguiente.

"Se tienen N peldaños que pueden ser ubicados uno a continuación de otro de forma horizontal o vertical. De esta manera se pueden formar escaleras ascendentes hacia la derecha (como ilustra la Figura 8-5). Si no puede haber dos peldaños verticales contiguos, determine la cantidad de escaleras posibles que pueden formarse con N peldaños."

Figura 8-5. Escaleras válidas para 3 peldaños

[2] De hecho, si es usted ansioso le sugerimos (en dependencia de su CPU) no intentar calcular un término más allá del 40 o 50.

Draft para estudiantes de MATCOM UH

8.3.3 LA SUCESIÓN DE LOS NÚMEROS DE CATALAN. EL PROBLEMA DE LOS PARÉNTESIS BALANCEADOS

Los números de Catalan son otra sucesión interesante que sirve para aplicar a la solución de varios problemas matemáticos.

Tomemos como primer ejemplo el siguiente problema:

"Se tienen N paréntesis abiertos '(' y N paréntesis cerrados ')'. Se desea conocer la cantidad de formas en que se pueden tener cadenas de paréntesis dispuestos de manera balanceada, es decir, que siempre se tenga la misma cantidad de abiertos que de cerrados y que si se recorre la cadena de izquierda a derecha nunca se encuentre un paréntesis cerrado si antes no apareció ya su correspondiente abierto."

La Figura 8-6 muestra algunos ejemplos de configuraciones posibles que se pueden formar con un par de paréntesis, con dos pares y con tres pares.

```
()      ()()    ()()()
        (())    ()(())
                (()())
                ((()))
                (())()
```

Figura 8-6. Formas balanceadas para 1, 2 y 3 pares de paréntesis

Para este problema se puede seguir un razonamiento parecido al problema de las losas. El primer paréntesis siempre tiene que ser un paréntesis abierto, al cual tiene que corresponder un paréntesis cerrado. Cualquier disposición con *N* pares de paréntesis tendrá por tanto la forma que muestra la Figura 8-7.

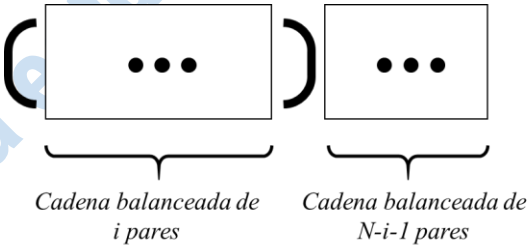

Figura 8-7. Estructura de cualquier cadena balanceada de paréntesis

Donde *i* puede tomar los valores desde 0 hasta *N-1*. Note que para *N* igual a 1 nos daría

(*cadena balanceada de* 0 *pares*) *cadena balanceada* de 0 *pares*

que no es más que la cadena ( ).

Para contar la cantidad total de configuraciones, basta sumar las configuraciones para cada valor válido de *i*. Para un valor de *i* determinado, las configuraciones posibles están dadas por una configuración válida para el primer bloque (una cadena balanceada de *i* pares) y una para el segundo bloque (una cadena balanceada de *N-i-1* pares). Por tanto, la cantidad de formas para determinado valor de *i* es la multiplicación de la cantidad de

Draft para estudiantes de MATCOM UH

formas posibles para el primer bloque por la cantidad de formas posibles para el segundo bloque. Quedando la expresión:

$$C(N) = \sum_{i=0}^{N-1} C(i) * C(N-i-1)$$

Donde el caso $C(0) = 1$, porque con 0 pares de paréntesis se forma una única cadena (vacía) con paréntesis balanceados.

La Figura 8-8 muestra las combinaciones válidas que se formarían para N=5, y asumiendo que hay dos pares de paréntesis dentro del primer bloque (i=2) y 2 pares de paréntesis dentro del segundo (N-i-1 = 2).

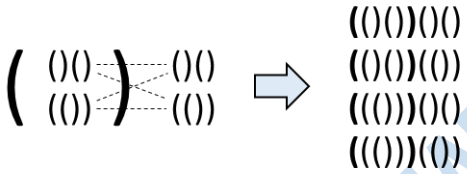

Figura 8-8. Configuraciones válidas de 5 pares de paréntesis con 2 pares dentro del primer bloque

El Listado 8-6 nos muestra la implementación para calcular el enésimo elemento de la sucesión de los números de Catalan. Note que la serie se expresa con un ciclo que acumula los valores del argumento de la serie en la variable `total`. Además, las llamadas recursivas siempre son en determinado valor menor estricto que el parámetro `N` (el valor de `i` es siempre menor estricto que `N` y el valor de `N-i-1` también es menor estricto que `N` para el mayor caso que es cuando `i` es 0). Esto garantiza que las llamadas recursivas convergen al caso base.

```
static int Catalan(int N) {
  if (N == 0) return 1;
  int total = 0;
  for (int i = 0; i < N; i++)
    total += Catalan(i) * Catalan(N - i - 1);
  return total;
}
```

Listado 8-6 Cálculo del e-nésimo término de la sucesión de Catalan

En el epígrafe anterior se vio un código iterativo para calcular la sucesión de Fibonacci; encuentre una solución iterativa al cálculo de los números de Catalan.

8.3.4 TRIANGULACIÓN DE POLÍGONOS, OTRA APLICACIÓN DE LA SUCESIÓN DE CATALAN

El siguiente problema también se puede resolver con la sucesión de los números de Catalan:

Una triangulación de un polígono es la descomposición del polígono en triángulos tales que sus lados no se corten, como se muestra en la Figura 8-9, y que cubran toda el área del pológono. ¿Cuántas posibles triangulaciones distintas se puede tener de un polígono convexo de N *puntos?*

Draft para estudiantes de MATCOM UH

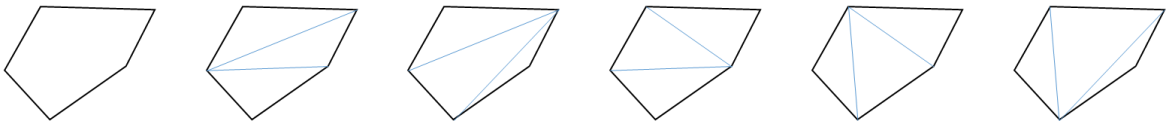

Figura 8-9. Descomposición en triángulos de un polígono de 5 vértices

El lector puede analizar la analogía entre un caso particular en este problema (Figura 8-10) y un caso particular de los paréntesis (Figura 8-7).

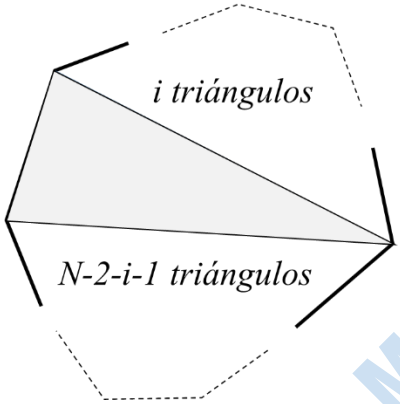

Figura 8-10. Caso particular en la descomposición de un polígono de N vértices en N-2 triángulos.

8.4 ¿CÓMO EJECUTA INTERNAMENTE LA RECURSIVIDAD?

En la mayoría de los lenguajes de programación, la ejecución de un método[3] provoca que internamente se reserve una memoria para almacenar los valores de los parámetros, las variables locales del método, además de la dirección en memoria hacia donde hay que retornar cuando la ejecución del método termine. Esta memoria se almacena en una estructura interna que se denomina **registro de activación** (*stack frame*) y que funciona como una **pila**[4].

Por ejemplo, si tenemos un código como el del Listado 8-7, durante la ejecución la pila pasará por cada uno de los estados que se muestran en la Figura 8-11.

```
static void H(string s) {
  Console.WriteLine(s);
}

static void G(int a) {
  H(a + " meters");
}

static void F(int x) {
  G(4 * x);
  H("ok");
}

static void Main() {
  F(5);
}
```

[3] Dependiendo del momento histórico y del lenguaje de programación, se le ha llamado también función, procedimiento, subrutina.
[4] Las pilas son estructuras de datos muy utilizadas y que se estudian especialmente en el Capítulo 13.

Prohibida para estudiantes de MATCOM UH

Listado 8-7 Programa con una secuencia de llamadas

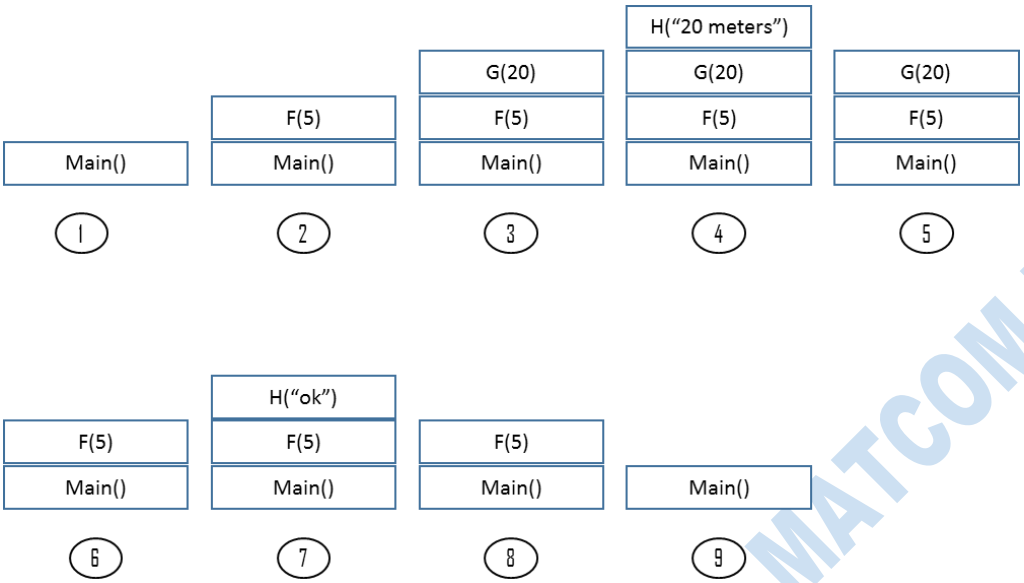

Figura 8-11. Transiciones entre los estados de la pila a medida que avanza la ejecución del programa

Para simplificar, en la figura sólo se ha mostrado una representación informal de los valores almacenados en la pila, omitiendo algunos registros como el valor de retorno y variables locales, así como las llamadas a métodos de la biblioteca estándar. También las cadenas de texto se están representando directamente como un valor, aunque realmente lo que se almacena en la pila es una referencia a otra zona de memoria que se conoce como heap.

Lenguajes como C#, Java y otros ejecutan en lo que se llama un contexto de ejecución administrado (managed runtime). La zona de memoria conocida como *heap* es administrada automáticamente para atender las peticiones de memoria que ocurren según la ejecución de la aplicación; por ejemplo, para ubicar la memoria de un array cuando se hace `new int[100]`. Esta ejecución administrada utiliza lo que se conoce como recolector de basura (garbage collector) para liberar y reubicar la memoria ocupada en heap.

Cuando aquí se ha explicado el funcionamiento de la recursividad por mediación de una pila, lo que se ubica y quita de la pila son las referencias a los objetos cuya memoria está realmente en el heap. Cuando no haya disponibilidad de memoria, es el recolector de basura quien se encarga de determinar cuáles son las zonas de memoria que ya no están siendo referenciadas para entonces recuperarlas.

Una vez que se termina cada método, el control de la ejecución retorna al método en la instrucción siguiente a la que hizo la llamada a dicho método y se retira de la pila el registro de activación del método, quedando en el tope el registro de activación del método previo (el que estaba ejecutando cuando se llamó al que ahora acaba de terminar). Esta representación de la ejecución puede entenderse de forma más compacta en el diagrama de la Figura 8-12.

Draft para estudiantes de MATCOM UH

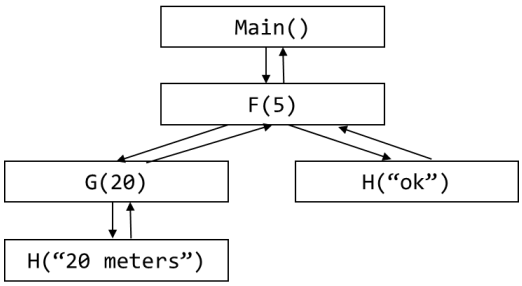

Figura 8-12. Ejecución de un programa. Las flechas hacia abajo indican llamadas a métodos; las flechas hacia arriba, retornos

Esta forma de almacenamiento de los registros de activación usando una pila es lo que permite el funcionamiento de un método recursivo (un método que en la ejecución de una llamada puede desencadenar una sucesión de llamadas que terminen llamándolo a él mismo). Una nueva llamada al método sin haber terminado la ejecución de la llamada anterior no sobrescribiría la información de los datos de la llamada anterior, porque cada vez que se efectúa una llamada se crea un nuevo registro de activación con sus propios valores (parámetros y variables locales) sobre el tope de la pila.

Por eso es importante que la secuencia de llamadas recursivas alcance en algún momento a alguno de los casos base. Si por una mala concepción de la solución recursiva, o por un error de programación, no se alcanza ningún caso base (y por consiguiente nunca se retorna), entonces el espacio en memoria de la pila, en la que se guardan los registros de activación, terminará por agotarse, lo que se manifestará en que el sistema lanzará una excepción `StackOverflowException`.

Se le llama recursividad directa cuando la llamada se hace al propio método y recursividad indirecta significa que la llamada se realiza a otro método que a su vez llama a otro y así sucesivamente hasta que se vuelve a llamar al método original.

Es poco probable que se llegue a ser un buen programador si alguna vez no se ha pasado por la experiencia de equivocarse y haber provocado excepción de desbordamiento de pila (stack overflow). Si no le ha ocurrido, por lo general esto no quiere decir que usted no sea un programador infalible, sino que aún no ha programado lo suficiente.

La Figura 8-13 muestra las llamadas que se efectúan hasta alcanzar la condición de parada (caso base) en el cálculo de factorial de 3. Observe el resultado retornado en cada llamada, el cual va a ser utilizado en el método que la invocó.

Draft para Estudiantes de MATCOM UH

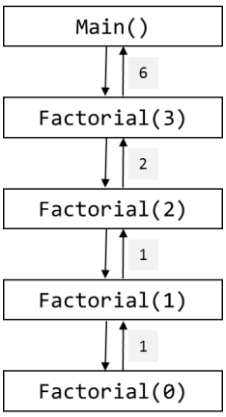

Figura 8-13.Secuencia de llamadas recursivos involucrados en el cálculo del Factorial(3)

Se puede realizar este mismo análisis en el cálculo recursivo de la sucesión de Fibonacci, pero note que en este caso cada método realiza dos llamadas, por lo que la cantidad de invocaciones es mucho mayor.

La explosión de llamadas recursivas para el cálculo de `Fibonacci(5)` puede verse en la Figura 8-14.

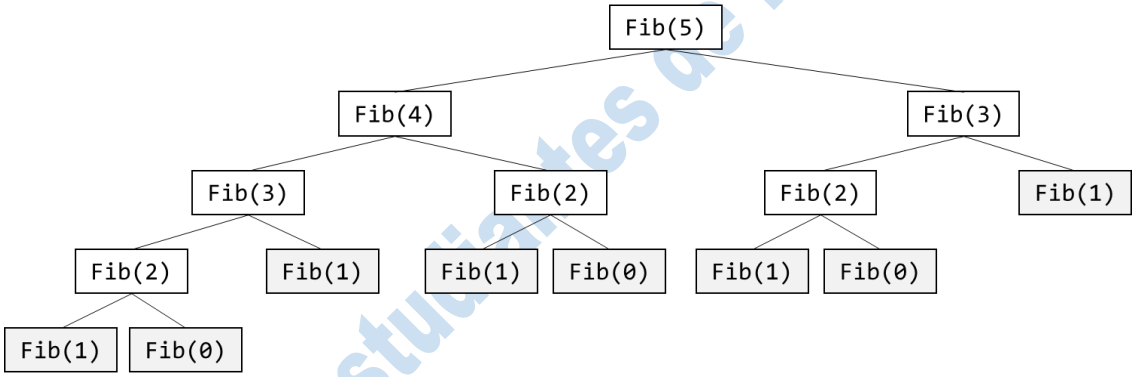

Figura 8-14. Representación gráfica de las llamadas recursivas en la ejecución de la función de Fibonacci para N=5

La sucesión de Fibonacci crece de forma exponencial y la cantidad de llamadas a casos bases de la solución recursiva coincide con el valor de Fibonacci devuelto como resultado; por lo tanto, el tiempo de ejecución aumenta también de forma exponencial. Este es un ejemplo de mal uso de la recursividad, puesto que el término enésimo de la sucesión puede ser calculado eficientemente de manera iterativa o usando otra forma de recursividad conocida como recursividad de cola. El tema de Costo de los Algoritmos se estudia en el Capítulo 9.

8.5 RECURSIVIDAD DE COLA

Si nos fijamos en el código del cálculo de factorial (Listado 8-1), luego de la llamada recursiva lo que se hace es un cálculo en el que participa el valor devuelto por la llamada recursiva y retornar a quien llamó (`return n*Factorial(n-1)`). Incluso esta multiplicación podría colocarse dentro de la propia llamada recursiva para que sea la propia llamada recursiva la que vaya haciendo las multiplicaciones (Listado 8-8).

Draft para estudiantes de MATCOM UH

```
public static long Factorial(int n) {
  return Factorial(n, 1);
}

private static long Factorial(int n , long accum) {
  if (n == 0) return accum;
  return Factorial (n-1, n*accum);
}
```

Listado 8-8 Factorial con recursividad de cola

Este tipo de recursividad se denomina **recursividad de cola** (*tail recursion*), y tales casos son fácilmente sustituibles por una implementación iterativa.

El caso de la sucesión de Fibonacci también se podría expresar con recursividad de cola, como se muestra en el Listado 8-9. Pruebe ejecutando este código para que vea que es tan eficiente como el del Listado 8-5.

```
public static long FibonacciCola(int n)
{
  return FibonacciCola(n, 1, 1);
}

static long FibonacciCola(int n, long a, long b)
{
  if (n <= 2)
    return b;
  else
    return FibonacciCola(n - 1, b, a+b);
}
```

Listado 8-9 Fibonacci con recursividad de cola

El Listado 8-10 nos muestra dos implementaciones para sumar todos los elementos de un array. El método `SumArrayIterativo` suma todos los elementos iterativamente utilizando un ciclo, mientras que el método `SumArrayRec` suma los elementos usando recursividad de cola. Note como en la llamada recursiva se pasa como parámetro `indice+1` y que el caso base hace que la recursividad termine cuando `indice==a.Length`; esto equivale al incremento de la variable del ciclo en la solución iterativa.

```
public static double SumArrayIterativo(double[] a)
{
  double sum = 0;
  for (int k = 0; k < a.Length; k++)
    sum += a[k];
  return sum;
}

public static double SumArrayRec(double[] a)
{
  return SumArrayRec(a, 0, 0);
}
private static double SumArrayRec(double[] a,
```

Draft para estudiantes de MATCOM UH

```
    double acumulado, int indice) {
  if (indice == a.Length)
    return acumulado;
  return SumArrayRec(a, acumulado + a[indice], indice+1);
}
```

Listado 8-10 Sumar todos los elementos de un array iterativa y recursivamente

Obviamente, si usted prueba ambas soluciones verá que dan el mismo resultado. La secuencia de llamadas recursivas simula el efecto de las iteraciones repetidas. Sin embargo, según lo explicado en el epígrafe 8.4 (Ejecución de la Recursividad), la solución recursiva va guardando en la pila los registros de activación (que luego va desempilando al regresar cuando se alcance el caso base). Esos registros de activación ocupan memoria y pueden llegar a desbordar el tamaño dedicado a dicha pila. Pruebe ejecutando ambos métodos incrementando el tamaño del array `a` hasta que la ejecución del método `SumArrayRec` provoque la excepción `StackOverflowException`.

Los algoritmos de recursividad de cola pueden ser fácilmente expresables con código iterativo, evitando con ello apilamiento innecesario de llamadas recursivas.

Incluso en lenguajes funcionales (como es el caso de F#), en los que la recursividad de cola juega un papel importante en la expresividad, los compiladores suelen generar internamente código iterativo equivalente, optimizando el uso de la pila (liberando de este trabajo al programador y evitando con ello la posibilidad de cometer errores). Con ello se evita que la secuencia de llamadas recursivas ocupe memoria innecesariamente por los registros de activación.

De modo que usted no se preocupe: exprese su solución como le sea más sencillo, ya sea haciendo recursividad de cola o iterativamente.

8.5.1 EJERCICIOS CON RECURSIVIDAD DE COLA

Implemente con recursividad de cola la solución a los siguientes problemas:

1- Implemente el máximo, mínimo y búsqueda de un elemento en un array.
2- Implemente la conversión de un número a su representación en binario. Ejemplo: para un valor de N igual a 14, se debe devolver "1110".
3- Dado un conjunto de puntos sobre el eje x, determine la menor cantidad de segmentos de tamaño 1 unidad que es necesario tener para cubrir todos los puntos del conjunto.

Varios de los problemas que se plantean en los capítulos anteriores en los que se tratan los ciclos y los arrays pueden ser resueltos utilizando recursividad de cola. Se sugiere al lector que como entrenamiento transforme los códigos correspondientes.

8.6 REDUCE Y VENCERÁS…

Como ya se ha dicho anteriormente, la expresión de una solución basándose en la recursividad debe garantizar que la recursividad converja, es decir, que se alcance algún caso base. Esta convergencia por lo general se expresa "reduciendo" el problema original hasta que éste llegue a ser igual a algún caso base. Y esta reducción se logra

Draft para estudiantes de MATCOM UH

manipulando los valores de algunos de los parámetros del método recursivo. Por ejemplo, calcular el factorial de *N* se reduce a calcular el factorial de *N*-1; calcular Fibonacci de *N* se reduce a calcular Fibonacci de *N*-1 y de *N*-2; saber mover *N* discos en las torres de Hanoi se reduce a saber mover *N*-1 discos.

Los problemas más sencillos de resolver recursivamente son aquellos en los que el problema inicial depende de la solución de un único sub-problema, como fueron los casos que ya vimos de factorial y máximo común divisor.

Veamos a continuación algunos problemas en que además esta reducción puede ser significativa porque acelera la convergencia al caso base y por tanto el tiempo de ejecución del método recursivo.

8.6.1 BÚSQUEDA BINARIA

En el Capítulo 7, Arrays, se estudia una solución al problema de buscar si un valor *x* está en un array aprovechando el caso en el que los valores del array estén ordenados. Esto que se denomina **búsqueda binaria** es un ejemplo de **reduce y vencerás** y puede también ser expresado de manera recursiva (Listado 8-11).

Al tomar el elemento *y* que se encuentre en la mitad del array y compararlo con el valor *x* que se está buscando para decidir con ello si seguir buscando en la primera mitad o en la segunda mitad se está **reduciendo** a la mitad la cantidad de elementos en los que continuar la búsqueda.

Si *x* es menor que *y*, eso quiere decir entonces que de estar en el array estaría en la mitad a la izquierda de *y*, y si es mayor que *y* entonces, de estar en el array, estaría en la mitad a la derecha. Por último, si *x* es igual a *y* entonces ya lo encontramos, y la respuesta es `true` y estamos en un caso base de la recursividad.

El otro caso base de la recursividad es si llegamos a un segmento de longitud cero y por tanto no se puede seguir dividiendo porque ya no queda en dónde buscar. En este caso la respuesta es `false`.

Note que el método `Pos` original llama a su vez a una sobrecarga del mismo en la que se añaden dos parámetros que indican los índices que delimitan el segmento del array en el que se va a buscar (en la primera llamada éstos son `0` y `a.Length-1`).

En la variable `medio` queda el índice de la posición intermedia del segmento en el que se está buscando.

Note que en las dos posibles llamadas recursivas se **reduce** a la mitad la longitud del segmento en el que se busca, de modo que en algún momento o encontramos al elemento buscado o llegará a cumplirse que `inf > sup` porque ya no tenemos segmento en el que buscar.

```
public static int Pos(double[] a, double x)
{
  return Pos(array, 0, array.Length - 1, x);
}

private static int Pos(double[] a, int inf, int sup, double x)
```

Draft para estudiantes de MATCOM UH

```
{
  if (inf > sup)
    return -1;
  int medio = (inf + sup) / 2;
  double y = a[medio];
  if (x < y)
    return Pos(a, inf, medio - 1, x);
  if (x > y)
    return Pos(a, medio + 1, sup, x);
  return medio;
}
```

Listado 8-11 Implementación recursiva del algoritmo de búsqueda binaria

Este ejemplo nos ilustra a su vez un patrón que es frecuente en la implementación de métodos recursivos. Se tiene un primer método que es el que es público a los usuarios del código y el cual no es explícitamente recursivo (la primera sobrecarga de `Pos`). Este método es a su vez el que prepara las condiciones para llamar por primera vez a un segundo método que es el que será verdaderamente recursivo (la segunda sobrecarga de `Pos`). En este caso particular, al llamar al `Pos` recursivo se le pasan los valores iniciales a los parámetros `inf` y `sup`.

Note que no hay por qué exigirle al usuario del método que quiere buscar un valor `x` en un array `b` que invoque al método en la forma

```
Pos(b, 0, b.Length-1, x)
```

cuando lo natural para él debe ser escribir solamente

```
Pos(b, y)
```

Esto, además de ser más simple, le evita errores por confusión en el uso de los parámetros `inf` y `sup`. Es por eso que la segunda sobrecarga de `Pos` se definió `private` para que no pueda ser invocada directamente.

Siempre es una buena práctica de programación no hacer un uso excesivo de parámetros en el diseño de un método, evitando con ello confusión en cuanto a los valores que deben pasársele y el orden en que deben utilizarse.

8.6.2 CONVERSIÓN DE DECIMAL A BINARIO

Otra aplicación del método de "reduce y vencerás" la podemos apreciar en el ejemplo siguiente. Se quiere convertir un valor entero positivo N a una cadena de texto (`string`) que sea la representación en binario de dicho número entero.

Para resolver este problema con un enfoque recursivo de reducción, veamos si hay alguna relación entre la representación en binario de N y la de un número menor que N.

Sea la representación en binario de N el número $\overline{d_k d_{k-1} \dots d_1 d_0}$ donde los valores d_i son dígitos binarios (0 o 1). Esto significa que N se obtiene como:

$$N = d_k 2^k + d_{k-1} 2^{k-1} + \cdots + d_2 2^2 + d_1 2^1 + d_0 2^0$$

Draft para estudiantes de MATCOM UH

Todos los dígitos desde d_1 hasta d_k están multiplicados por una potencia de 2. Ello significa que si el número es par el valor de d_0 es 0 y si es impar es 1. Por otra parte, se puede observar que si se divide N entre 2 se obtiene el número:

$$\frac{N}{2} = d_k 2^{k-1} + d_{k-1} 2^{k-2} + \cdots + d_2 2^1 + d_1 2^0 = \overline{d_k d_{k-1} \ldots d_1}$$

Aplicando el mismo razonamiento para este número podemos saber entonces cuál es el último dígito. Esto nos permite obtener la cadena representación en binario de N obteniendo recursivamente la cadena con la representación en binario de N / 2 y concatenándola con "0" si N es par o "1" si N es impar. La escritura del código en C# se le deja al lector.

8.7 DIVIDE Y VENCERÁS

En el epígrafe anterior los problemas que se resuelven aplicando la solución recursiva a una versión "reducida" del problema hasta que la versión reducida sea un caso base.

Vamos a ver ahora una estrategia de un estilo similar a la de "reduce y vencerás", pero que implica descomponer el problema en varios sub-problemas de menor complejidad para luego combinar las soluciones a los subproblemas y conformar la solución del problema original. Esta estrategia se denomina **Divide y Vencerás**.

Las máxima latina **divide et impera** fue utilizada por el emperador romano Julio César como estrategia del imperio romano. Con ello se buscaba mantener bajo control un territorio y/o una población, dividiendo y fragmentando el poder de las distintas facciones o grupos existentes en los territorios que se pretendían someter. También fue muy usada por el emperador Napoleón. Al margen de su interpretación política o militar de dominación, pudiera ser aplicada a cualquier ámbito en el que para obtener un resultado (o para obtener un mejor resultado), es necesario o conveniente en primer lugar romper o dividir lo que se opone o frena la obtención de la solución.

A diferencia del caso de "reduce y vencerás" en la búsqueda binaria, en el que el array se divide a la mitad, pero luego de buscar en una de las dos mitades ya luego no hay que hacer nada más, en la estrategia de "divide y vencerás" luego de las llamadas recursivas para solucionar los subproblemas hay que saber combinar éstas para resolver el problema original, como veremos en los subepígrafes a continuación.

8.7.1 ORDENACIÓN POR MEZCLA. MERGESORT

Analicemos de nuevo el problema de ordenar los valores en un array, cuya solución iterativa con dos ciclos anidados se ve en el Capítulo 7. Suponga que se divide el array en dos mitades y que aplicando recursividad se ordena cada una de las mitades. Tendríamos ahora dos mitades ordenadas; pero para obtener la solución al problema original de tener ordenado el array completo, habría ahora que **mezclar** ambas mitades para obtener un array con los mismos valores y que se mantenga ordenado. Este algoritmo se conoce como **ordenación por mezcla** (en inglés *mergesort*).

```
public static void OrdenarPorMezcla(int[] a)
```

Draft para estudiantes de MATCOM UH

```
{
  OrdenarPorMezcla(a, aux, 0, a.Length - 1);
}

private static void OrdenarPorMezcla(int[] a, int inf, int sup)
{
  if (inf < sup)
  {
    int medio = (inf + sup) / 2;
    OrdenarPorMezcla(a, inf, medio);
    OrdenarPorMezcla(a, medio + 1, sup);
    Mezclar(a, inf, medio, sup);
  }
}
```

Listado 8-12 Ordenación por mezcla

La recursividad nos hará buena parte del trabajo (Listado 8-12) al ordenar cada una de las dos mitades. El caso base de esta recursividad es cuando `inf < sup`, es decir, que el segmento a ordenar está compuesto por un solo elemento y por tanto obviamente ya estará ordenado. Lo que queda por hacer es mezclarlos de modo tal de mantener el orden (Listado 8-13).

```
private static void Mezclar(int[] a, int inf, int medio, int sup)
{
  //Array auxiliar para ir dejando el resultado de la mezcla
  int[] aux = new int[sup - inf + 1];
  //Para recorrer la 1ra mitad
  int izq = inf;
  //Para recorrer la 2da mitad
  int der = medio + 1;
  //Para ir dejando la mezcla en aux
  int pos = 0; //

  while (izq <= medio && der <= sup)
  {
    if (a[izq] < a[der]) aux[pos++] = a[izq++];
    else aux[pos++] = a[der++];
  }
  if (izq <= medio) Array.Copy(a, izq, aux, pos, medio - izq + 1);
  if (der <= sup) Array.Copy(a, der, aux, pos, sup - der + 1);

  //Pasar el resultado de la mezcla de nuevo para a
  Array.Copy(aux, 0, a, inf, sup - inf + 1);
}
```

Listado 8-13 Mezclar los dos segmentos del array ordenado

Note que el método `Mezclar` define internamente un array auxiliar `aux` para dejar el resultado de lo que va mezclando para antes de finalizar pasar los valores en `aux` al array original `a`.

Esta solución está creando estos arrays auxiliares por cada llamada recursiva al método `OrdenarPorMezcla`, que es quien está llamando al método `Mezclar`. Esto provoca una fragmentación innecesaria de la memoria, que en este caso puede evitarse si se crea un único array `aux` que sea global a todos los procedimientos recursivos, como se muestra en el Listado 8-14. Note que en este caso se crea un array `aux` de tamaño igual al del

Biblioteca estudiantes de MATCOM UH

array original antes de hacer la primera llamada al método recursivo `OrdenarPorMezcla`, y luego dicho array es el que se pasa en cada una las llamadas recursivas.

Ciertamente esto implica estar pasando un parámetro más (`aux`) en cada llamada, pero como los arrays se tratan por referencia esto no significa estar copiando cada vez todos los elementos del array, sino solo copiar la referencia. En este caso no hay tanta fragmentación de la memoria porque el espacio para el array `aux` solo se separó espacio una vez.

```
public static void OrdenarPorMezcla(int[] a)
{
  int[] aux = new int[a.Length];
  OrdenarPorMezcla(a, aux, 0, a.Length - 1);
}
private static void OrdenarPorMezcla(int[] a, int[] aux, int inf, int sup)
{
  if (inf < sup)
  {
    int medio = (inf + sup) / 2;
    OrdenarPorMezcla(a, aux, inf, medio);
    OrdenarPorMezcla(a, aux, medio + 1, sup);
    Mezclar(a, aux, inf, medio, sup);
  }
}
private static void Mezclar(int[] a, int[] aux, int inf, int medio, int sup)
{
  int izq = inf; // para recorrer la 1ra mitad
  int der = medio + 1; // para recorrer la 2da mitad
  int pos = inf; // para ir dejando la mezcla en aux

  while (izq <= medio && der <= sup)
  {
    if (a[izq] < a[der]) aux[pos++] = a[izq++];
    else aux[pos++] = a[der++];
  }
  if (izq <= medio) Array.Copy(a, izq, aux, pos, medio - izq + 1);
  if (der <= sup) Array.Copy(a, der, aux, pos, sup - der + 1);

  Array.Copy(aux, inf, a, inf, sup - inf + 1);
}
```

Listado 8-14 Ordenación por mezcla mejorado sin fragmentación de la memoria

Draft para estudiantes de MATCOM UH

Realmente en una plataforma como .NET, y en un lenguaje como C#, el espacio que ocupan los arrays se ubica en una zona de memoria que se conoce como *heap*. La memoria de este heap se administra dinámicamente durante la ejecución, y se dispone de un mecanismo conocido como recolector de basura (garbage colletor) que detecta automáticamente las zonas que fueron ocupadas en el heap (por ejemplo, los arrays que fueron siendo creados en las llamadas recursivas en la variante del Listado 8-13) y que recupera la memoria que ya no está siendo utilizada (la "basura").

En principio, el recolector de basura trabaja trasparente y silenciosamente, por lo que el programador no debe preocuparse de esto para que pueda dedicarse a escribir el código como le resulte más sencillo. Pero un poco de cultura (como se ha pretendido mostrar en este ejemplo) para no crear innecesariamente basura no viene mal. Esto además le puede ser útil cuando el tema de la eficiencia de su aplicación en situaciones límites sea crítico y no conviene que el recolector de basura arranque a funcionar cuando se espera, por ejemplo, una respuesta en tiempo real.

Este método de ordenación por mezcla disminuye considerablemente el tiempo de ejecución de la ordenación si se le compara con el método de **ordenación por mínimos sucesivos** que se estudia en el capítulo de Arrays. Mida con ayuda de un `Stopwatch` el tiempo de ejecución de cada una de las variantes. Una explicación más formal sobre el costo se estudia en el Capítulo 9.

8.7.2 QUICKSORT

En el caso anterior de ordenación por mezcla, se escogió la posición intermedia del array como índice para particionar el problema en dos subproblemas intuitivamente parejos (la mitad) en cuanto al tamaño. No obstante, no siempre se puede asumir que un problema se pueda subdividir en dos subproblemas de igual tamaño. En este subepígrafe se desarrollará el algoritmo conocido como **QuickSort**.

Este algoritmo hace también el ordenamiento basándose en ordenar dos subarrays, pero bajo otro enfoque. La idea se basa en poder "particionar" el array en dos subarrays, dejando en uno de los subarrays todos los valores menores que un cierto valor *p* (pivote) y en el otro subarray todos los valores mayores que *p*. Se puede obtener entonces una ordenación del array original si se ordena el primer subarray y (en lugar de mezclar, como se hace en MergeSort) se **concatena** con el resultado de la ordenación del segundo. Con la concatenación basta porque ningún elemento de la primera parte será mayor que ninguno de la segunda, y a su vez, ningún elemento de la segunda parte será menor que ningún elemento de la primera.

El algoritmo de QuickSort fue propuesto por Charles Anthony Richard Hoare (1934), familiarmente conocido como "Tony Hoare". Hoare es un reconocido científico británico de la Ciencia de la Computación, merecedor del premio Turing en 1980 por sus trabajos en el desarrollo y diseño de los lenguajes de programación. Desarrolló en 1960 el algoritmo de ordenación Quicksort, que es en la actualidad uno de los más utilizados.

Prof. para estudiantes de MATCOM IH

El quid del algoritmo recae entonces en el método `Particiona`. Si se va a ordenar el array de menor a mayor, este método opera de la siguiente manera (Listado 8-15):

1. Recorriendo el array de izquierda a derecha, encontrar el primer valor `a[i]` tal que sea mayor que el privote *p*.
2. Recorriendo el array de derecha a izquierda, encontrar el primer valor `a[j]` tal que sea menor o igual que el privote *p*.
3. Si la posición `i` del valor `a[i]` encontrado en (1) primero es inferior al valor `a[j]` encontrado en (2), entonces se intercambian los valores `a[i]` y `a[j]`.
4. Se repite este proceso mientras la posición `i` del valor encontrado de izquierda a derecha esté en una posición inferior a la posición `j` del encontrado de derecha a izquierda. En caso contrario, se termina el proceso y se devuelve el valor `j`, que indicará respectivamente la cota superior e inferior de los dos subarrays a ordenar recursivamente.

De modo que este proceso garantiza que van quedando los valores menores que *p* en el segmento izquierdo del array y los valores mayores o iguales que p en el segmento derecho del array. El punto en que se encuentran las búsquedas indica el final de la primera parte y el comienzo de la segunda.

```
static int Particiona(int[] a, int ini, int fin, int pivote)
{
    int i = ini - 1;
    int j = fin + 1;
    while (true) {
        do i++; while (a[i] < pivote);
        do j--; while (a[j] > pivote);
        if (i >= j)
            return j;
        var temp = a[i];
        a[i] = a[j];
        a[j] = temp;
    }
}

static void QuickSort(int[] a, int ini, int fin) {
    if (ini >= fin) return;
    int piv = a[ini];
    int pos = Particiona(a, ini, fin, piv);
    QuickSort(a, ini, pos);
    QuickSort(a, pos + 1, fin);
}

public static void QuickSort(int[] a) {
    QuickSort(a, 0, a.Length - 1);
}
```

Listado 8-15 Ordenación por Quicksort usando como pivote el primer elemento del array

Queda una incógnita por solucionar: ¿cuál es el valor que se debe tomar como valor inicial del pivote? Este algoritmo funciona tomando como pivote cualquier valor *p* que esté en el array (en el Listado 8-15 se ha escogido el primer elemento del array); sin

Diapositivas para estudiantes de WATCOM UH

embargo, una adecuada selección del valor pivote puede ser clave en la eficiencia del algoritmo Quicksort. Esto será analizado en el Capítulo 9.

8.7.3 ENCONTRANDO LA MEJOR FORMA DE DIVIDIR EN CADA OCASIÓN . . .

Tanto en el algoritmo MergeSort como en Quicksort, el criterio de particionamiento es el mismo en cada llamada de la secuencia de llamadas recursivas (dividir a la mitad en MergeSort o tomar el primer elemento como pivote en el caso del QuickSort). En este epígrafe se analizarán otros ejemplos en los que el criterio de división en subproblemas no tiene que ser el mismo en cada llamada recursiva.

8.7.3.1 EL PROBLEMA DE ENCONTRAR LA MAYOR SUBSECUENCIA COMÚN DE CARACTERES PRESENTE EN DOS CADENAS

Veamos el problema de obtener la subsecuencia de caracteres común de mayor longitud, conocido por las siglas LCS (*largest common subsequence*) de dos cadenas. La subsecuencia no tiene por qué ser de caracteres contiguos en la cadena. Por ejemplo, la mayor subsecuencia común entre las cadenas `"computar"` y `"permutacion"` es la subsecuencia `"muta"`.

Una primera solución a este problema sería encontrar un carácter a_i de a que sea igual a un carácter b_j de b. Si dicho carácter es "parte de la solución óptima", entonces la solución óptima se formaría concatenando LCS(a[0...i-1], b[0...j-1]) + a[i] + LCS(a[i+1...], b[j+1...])[5].

Como no se sabe si la decisión de asumir que el a_i tal que es igual que b_j es realmente parte de la solución óptima, entonces el algoritmo debe probar con todas las combinaciones posibles de i y j tales que a_i sea igual que b_j y quedarse con la que obtiene la subsecuencia de mayor longitud (Listado 8-16).

```
public static string LCS(string a, string b) {
    string mejor = "";
    for (int i=0; i<a.Length; i++)
        for (int j=0; j<b.Length; j++)
            if (a[i] == b[j]) {
                var actual = LCS(a.Substring(0, i), b.Substring(0, j))
                           + a[i]
                           + LCS(a.Substring(i + 1), b.Substring(j + 1));
                if (actual.Length > mejor.Length)
                    mejor = actual;
            }
    return mejor;
}
```

Listado 8-16 Mayor subsecuencia común en dos cadenas

Este algoritmo obtiene la misma solución óptima siempre que subdivida utilizando alguna correspondencia de dicha solución. Para el caso de ejemplo, la solución `"muta"` se puede obtener de subdividir el problema de cualquiera de las siguientes formas:

```
LCS("co", "per") + 'm' + LCS("putar", "utacion")
LCS("comp", "perm") + 'u' + LCS("tar", "tacion")
```

[5] Para simplificar, se ha utilizado aquí informalmente la notación a[i...j] para referirse al substring que se forma con los caracteres desde i hasta j en a, o a[i...] para expresar el substring con los caracteres desde i hasta el final de a. Lamentablemente, una tal notación no es válida en código C#.

```
LCS("compu", "permu") + 't' + LCS("ar", "acion")
LCS("comput", "permut") + 'a' + LCS("r", "cion")
```

Repitiéndose un cómputo innecesario. Es decir, el algoritmo intentará en algún momento la variante de hacer coincidir la *m* entre las dos cadenas como parte de la solución óptima; en otro momento intentará asumirá la *u, y* así sucesivamente. En todos esos casos estará consiguiendo la misma subsecuencia óptima.

Supongamos que se quiere depender únicamente de la subdivisión que se refiere a la primera correspondencia entre las cadenas *a* y *b*. En este ejemplo:

```
LCS("co", "per") + 'm' + LCS("putar", "utacion")
```

En estos casos, la primera llamada siempre devolvería "" (cadena vacía), puesto que la opción es la primera coincidencia y los prefijos antes de dicha coincidencia no tienen ningún carácter en común. En el ejemplo éste es el caso con `LCS("co", "per")`. Queda resolver el LCS del texto que continúa a la primera coincidencia en ambas cadenas de entrada, en el ejemplo, `LCS("putar", "utacion")`. Para garantizar que las correspondencias que se procesan sean solamente las primeras correspondencias, basta con ir llevando un límite de hasta dónde buscar correspondencias en la cadena *b*.

Si se encontró en algún punto que el carácter a_i es la primera coincidencia y le corresponde el carácter b_j (es decir, el primer carácter en *b* igual a a_i), entonces debe considerarse cualquier correspondencia posterior a la posición *i* en *a* si coincide con un carácter anterior a *j* en *b*, puesto que la correspondencia entre un carácter posterior que *i* en *a* y posterior que *j* en *b* no sería la "primera coincidencia". El código quedaría como se muestra en el Listado 8-17.

```
public static string LCS(string a, string b) {
  string mejor = "";
  int ultimo = b.Length;
  for (int i = 0; i < a.Length; i++)
  {
    for (int j=0; j < ultimo; j++)
      if (a[i] == b[j])
      {
        var actual = a[i] + LCS(a.Substring(i + 1), b.Substring(j + 1));
        if (actual.Length > mejor.Length)
        mejor = actual;
        ultimo = j;
      }
  }
  return mejor;
}
```

Listado 8-17 Algoritmo de Mayor Subsecuencia Común optimizado

El algoritmo puede simplificarse considerando únicamente subproblemas que tengan un carácter menos en alguna de las cadenas de entrada. Al fin de cuentas, cualquiera de los subproblemas del algoritmo anterior puede analizarse si se "esquivan" el conjunto de caracteres de cada una de las cadenas *a* y *b* hasta llegar a la primera coincidencia.

En este caso, analizando únicamente los inicios de *a* y *b* podemos plantear la dependencia recursiva. Si el primer carácter de *a* es igual al primer carácter de *b*, entonces la mayor subsecuencia obligatoriamente tiene a `a[0]` como primer carácter, y le

Draft para estudiantes de MATCOM UH

continúa el `LCS (a[1…], b[1…])`. Si son distintos los caracteres iniciales de *a* y *b*, entonces uno de los dos obligatoriamente no puede estar en la subsecuencia resultante. Si el carácter de *a* fuese el que no puede estar en el resultado, entonces el resultado es igual al `LCS (a[1…], b)`; si no, si el primer carácter de *b* es el excluido entonces el resultado es igual al `LCS (a, b[1…])`. Por supuesto, como no podemos saber de antemano cuál de estas dos subdivisiones es la que resulta más conveniente, entonces evaluamos las dos posibilidades y nos quedamos con la que resulte más larga.

El código del Listado 8-18 muestra esta solución. Aunque aparentemente el código es más legible que las anteriores, es el más ineficiente. El lector puede comprobar los tiempos de ejecución de estas tres soluciones y analizar la aparición de "subproblemas" repetidos en esta última solución.

```
public static string LCS(string a, string b)
{
    if (a.Length == 0 || b.Length == 0) return "";
    if (a[0] == b[0]) return a[0] + LCS(a.Substring(1), b.Substring(1));
    string lcs1 = LCS(a.Substring(1), b);
    string lcs2 = LCS(a, b.Substring(1));
    return (lcs1.Length > lcs2.Length) ? lcs1 : lcs2;
}
```

Listado 8-18 Otra solución para la Mayor Subsecuencia Común

Un problema que puede resolverse de forma similar es el siguiente:

"*Dadas dos cadenas a y b, determinar cuál es la menor cantidad de operaciones que se deben realizar sobre los caracteres de la cadena a para obtener la cadena b. Donde las operaciones permitidas son insertar un carácter, eliminar un carácter o reemplazar un carácter por otro.*"

Se propone al lector implementar la solución a este problema.

8.7.3.2 PROBLEMA DE LA MULTIPLICACIÓN DE UNA CADENA DE MATRICES

Otro problema en que en cada llamada del proceso recursivo hay que encontrar la mejor forma de ir dividiendo es el siguiente.

"Considerando que la multiplicación de matrices es asociativa, se quiere determinar, dada una secuencia de matrices, cuál es la asociación que permita hacer la multiplicación que implique la menor cantidad posible de operaciones de multiplicación entre los elementos de la matrices".

Recuerde que para poder realizar la secuencia de multiplicaciones de matrices $m_0 \times m_1 \times \ldots \times m_{n-2} \times m_{n-1}$ se tiene que cumplir que $columnas(m_i) = filas(m_{i+1})$ para *i* entre 0 y n-2. Es decir, que la cantidad de columnas de la matriz operando izquierdo de la multiplicación sea igual a la cantidad de filas de la matriz operando derecho de la multiplicación.

Cuando se multiplican dos matrices:

$$A_{n\times k} \times B_{k\times m} = C_{n\times m}$$

Se realizan *n*k*m* operaciones de multiplicación entre los valores escalares elementos de las matrices de *A* y *B*.

Draft para estudiantes de MATCOM UH

Como la multiplicación de matrices es asociativa, se tiene que

$$A_{n\times k} \times (B_{k\times j} \times C_{j\times m}) = (A_{n\times k} \times B_{k\times j}) \times C_{j\times m}$$

En la asociación de la izquierda se realizan $k*j*m + n*k*m$ operaciones y en la de la derecha se realizan $n*k*j + n*j*m$ operaciones.

La optimización consiste en determinar cuál debe ser la asociación que conviene hacer en cada paso de la secuencia recursiva para que la cantidad total de multiplicaciones de escalares sea la menor.

La entrada para el problema de multiplicar *n* matrices (para simplificar, se considerará que las dimensiones son válidas, es decir que siempre se puede hacer la multiplicación) puede describirse con un array de *n+1* enteros, donde el elemento en la posición *i* indica la cantidad de filas de la matriz *i* (que para que la multiplicación sea válida se asume es igual a la cantidad de columnas de la matriz *i-1*); el elemento en la posición *n* (última posición del array de longitud *n+1*) corresponde entonces a la cantidad de columnas de la última matriz de la secuencia.

El array entrada para la multiplicación de matrices de la Figura 8-15 sería entonces {3, 2, 4, 3, 3}.

Para resolver este problema, analizaremos la estructura de una solución óptima. En cualquier asociación que sea una solución óptima, habrá una multiplicación de matrices que se hará de última (Figura 8-15).

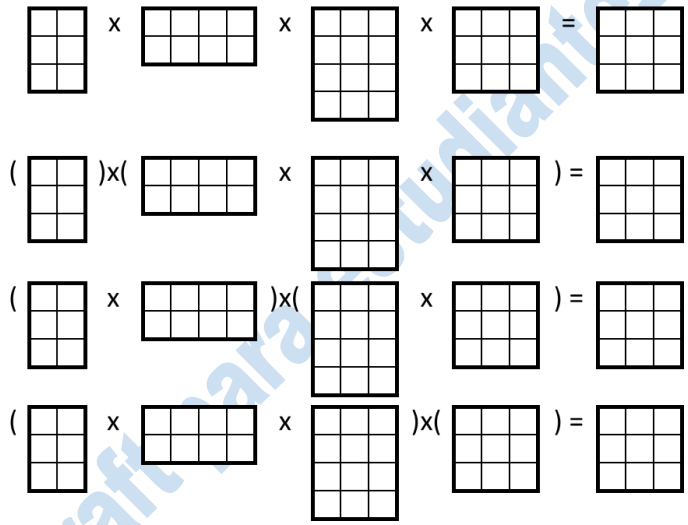

Figura 8-15 Posibles asociaciones para determinar la última multiplicación

Sea esta última operación la encargada de multiplicar la matriz resultante de la asociación óptima de las *i* primeras matrices con la matriz resultante de la asociación óptima de las *n–i* restantes matrices. La asociación óptima del primer grupo es independiente de la del segundo grupo. Por tanto, el algoritmo recursivo sólo tiene que tomar la decisión de cuál es la última multiplicación a realizar y explorar todas las posibilidades (suponer valores de *i* entre *1* y *n-1*). Luego, resolver recursivamente el óptimo para cada una de las dos secuencias de matrices en que se ha dividido la asociación y calcular el costo total (que será igual al costo de las multiplicaciones

Draft para estudiantes de MATCOM UH

obtenido gracias a la recursividad más la cantidad de operaciones de la última multiplicación). Este cómputo se realiza para cada posible partición y se selecciona aquella con la que obtenga la menor cantidad de operaciones.

```
public static int MultOptimaDeMatrices(int[] matrices) {
  return MultOptimaDeMatrices(matrices, 0, matrices.Length - 2);
}

private static int MultOptimaDeMatrices(int[] matrices, int ini, int fin) {
  if (ini == fin) return 0;
  int mejor = int.MaxValue;
  for (int i = 1; i < fin - ini + 1; i++)
  {
    int costoIzq = MultOptimaDeMatrices(matrices, ini, ini + i - 1);
    int costoDer = MultOptimaDeMatrices(matrices, ini + i, fin);
    int costoUltOperacion = matrices[ini] * matrices[ini+i] * matrices[fin+1];
    int costoGlobal = costoIzq + costoDer + costoUltOperacion;
    if (costoGlobal < mejor)
      mejor = costoGlobal;
  }
  return mejor;
}
```

Listado 8-19 Asociación óptima para la multiplicación de una cadena de matrices

El Listado 8-19 muestra la implementación. Note que al variar los valores de los parámetros `ini` y `fin` y el valor de la variable de control `i` del ciclo `for`, se está cambiando la forma de "particionar" el array.

8.8 BACKTRACKING

En muchos de los ejemplos vistos anteriormente, la solución óptima se ha obtenido analizando las características del problema para decidir cómo se puede "reducir" el problema general en subproblemas más pequeños. En otros casos se ha visto que no es posible saber con antelación cuál es la mejor reducción, por lo que se hace necesario iterar por todas las posibles reducciones (por ejemplo, analizar todas las posiciones donde particionar un array) para encontrar la mejor estrategia.

En este epígrafe se verá una estrategia de solución denominada **backtracking** (vuelta atrás) que se basa en analizar todas las posibles secuencias de decisiones ordenadas, que pueden depender unas de otras, y que nos pueden llevar a una solución. Si por el camino se determina que no se llegará a una solución, o que se llega a una solución que no es la óptima, entonces se **vuelve atrás** a la decisión que puede permitirnos explorar otra secuencia.

El ejemplo a continuación, conocido como el Problema de las 8 reinas, es el clásico para ilustrar esta estrategia.

8.8.1 EL PROBLEMA DE LAS N REINAS DEL AJEDREZ

Este problema consiste en encontrar una forma de ubicar N reinas de ajedrez en un tablero de N x N casillas, de forma que ningún par de reinas se amenacen mutuamente. Por ejemplo, una posible solución para este problema ubicando 8 reinas en un tablero estándar del ajedrez de 8 x 8 casillas se muestra en la Figura 8.16.

Draft para estudiantes de MATCOM UH

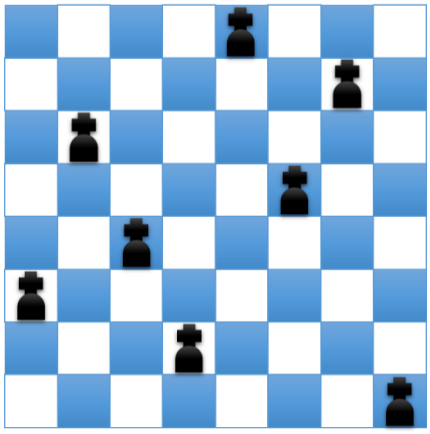

Figura 8.16 Ubicando 8 reinas en un tablero de 8 x 8 sin que ninguna esté amenazada

En una primera aproximación, concentrémonos en determinar todas las posibles formas de ubicar *N* reinas en el tablero. Para ello, es necesario diseñar una estrategia de solución que permita ir construyendo paso a paso cada una de las formas de ubicar a las reinas.

La reina del ajedrez se mueve cualquier cantidad de casillas en las direcciones horizontal, vertical y diagonal. Por tal motivo, es evidente que en cualquier posible ubicación que sea una solución, hay a lo sumo una reina en cada fila, y una reina en cada columna. Es decir, si ya se ha ubicado una reina en la fila *i*, no hay que probar a poner otra reina en otra casilla de la misma fila puesto que se amenazarían entre ellas. Es decir, que si hay una solución, en esta tendrá que haber ubicada una y solo una reina en cada fila.

Siguiendo este razonamiento, podemos empezar ubicando una reina en una casilla de la primera fila y probar otra reina en alguna casilla de la siguiente fila y que no amenace las reina puesta en la fila anterior. Por ejemplo, la solución mostrada en la Figura 8.16 consiste en la secuencia de decisiones que ubican a la reina de la fila 0 en la columna 4, la reina de la fila 1, en la columna 6, la reina de la fila 2 en la columna 1, y así sucesivamente[6].

El problema consiste entonces en cómo determinar las secuencia de ubicaciones para ver si alguna nos lleva a una solución. Para ilustrar esto, veamos manualmente cómo ubicar 4 reinas en un tablero de 4 x 4 (Figura 8.17).

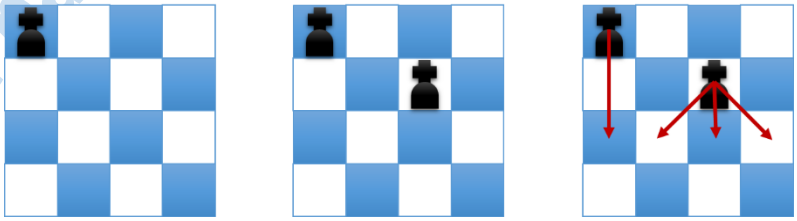

Figura 8.17 Ubicando 4 reinas en un tablero de 4 x 4, llegamos rápidamente a una configuración inválida

Si ubicamos la primera reina en la celda (0,0), entonces la segunda reina a ubicar en la siguiente fila no puede ser ubicada en la columna 0 ni en la columna 1, por lo que el primer lugar donde podemos ubicarla es en la casilla (1,2). Una vez llegados a este

[6] El lector recordará que en programación comenzamos a contar desde cero.

Draft para estudiantes de MATCOM UH

punto, descubrimos que es imposible ubicar la tercera reina, pues todas las casillas de la fila 2 están amenazadas. Por lo tanto, hemos descubierto que la secuencia que estamos siguiendo no lleva una solución válida.

En este punto solo nos queda "deshacer" la última acción (es decir no ubicar una reina en la casilla (1,2)), y probar por otro camino (Figura 8.18). Decidimos ahora probar a ubicar la segunda reina en la casilla (1,3). Ahora es posible ubicar la tercera reina en la casilla (2,1). Sin embargo, una vez hecho esto, descubrimos nuevamente que no es posible ubicar la cuarta reina en ninguna casilla de la fila 3, pues todas las casillas quedan amenazadas.

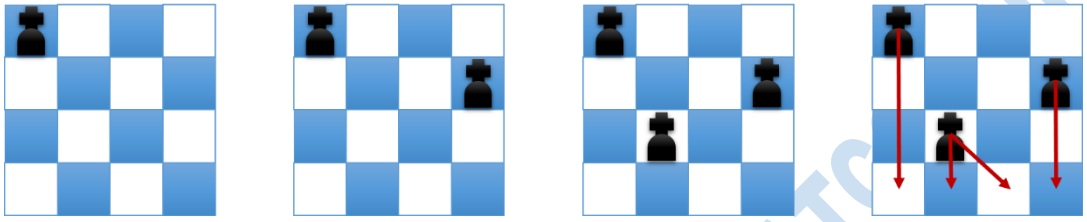

Figura 8.18 Otra posible solución que también resulta ser inválida

En este punto hay que volver atrás y "deshacer" la última acción, que fue ubicar a la tercera reina en la casilla (2,1); pero como en este caso no existe ninguna otra forma de ubicarla (pues las restantes casillas de la fila 2 están amenazadas), entonces es necesario "deshacer" además la acción anterior a esta, que fue ubicar a la segunda reina en la casilla (1,3). Sin embargo, la segunda reina tampoco puede ser ubicada en más ninguna posición, por que tenemos que retroceder aún más y "deshacer" la primera acción, que fue ubicar la primera reina en la casilla (0,0) y tratar de ubicarla en otra columna de la fila 0 y volver a empezar.

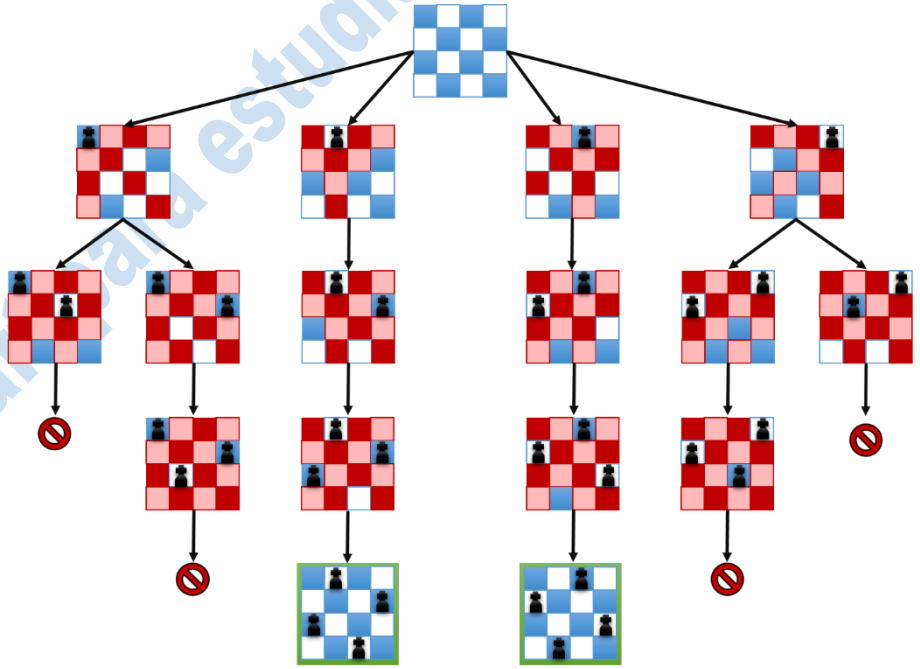

Figura 8.19 Todas las posibles formas de ubicar 4 reinas, hasta el punto donde se verifica que un tablero es inválido

Draft para estudiantes de MATCOM

En la Figura 8.19 se muestran todas las posibles formas de ubicar las 4 reinas en un tablero de 4 x 4. En cada tablero se ha señalado en color rojo o rosa las casillas donde es imposible ubicar una nueva reina porque se amenazaría con las reinas anteriores.

Como se puede observar, esta estrategia requiere que se vayan "probando" posibles alternativas, y cada vez que se determine que una alternativa no es posible de aplicar, entonces es necesario "deshacer" las decisiones tomadas anteriormente, hasta el punto donde exista una opción aún no probada. Es para implementar este concepto de "probar y deshacer" que nos auxiliaremos de la recursividad.

El método recursivo tendrá la responsabilidad, en cada llamada, de probar todas las posibles opciones para la decisión en cuestión. En este ejemplo de ubicar *N* reinas, una decisión consiste en ubicar a la i-ésima reina (que ya sabemos tiene que ir en la i-ésima fila) en alguna de las *N* casillas (columnas) posibles de esa fila. Por tanto, el método recursivo "probará" en cada llamada con todas las posibles columnas donde ubicar a la i-ésima reina. Una vez tomada esta decisión, se invocará recursivamente para decidir dónde ubicar a la reina *i+1*.

Cada vez que se toma una decisión, es necesario realizar cierta acción, que indique que la decisión ha sido tomada. En este ejemplo, dicha acción consiste en ubicar una reina en el tablero en la posición decidida, lo cual podemos modelar simplemente mediante un array de dos dimensiones de tipo `bool[,]` en el que un valor `true` en el elemento *i,j* indique que se está probando a poner una reina en dicha casilla *i,j*. Deshacer esta acción para probar con otras se expresa volviendo a poner en `false` el valor de este elemento *i,j*. Este paso es importante, pues es el que garantiza que los futuros llamados recursivos "sepan" cuál fue la decisión tomada.

```
static int UbicaReinas(int n)
{
  bool[,] tablero = new bool[n, n];
  return UbicaReinas(n, 0, tablero);
}

private static int UbicaReinas(int n, int i, bool[,] tablero)
{
  if (i == n)
    return EsValido(tablero) ? 1 : 0;

  int total = 0;

  for (int j = 0; j < n; j++)
  {
    tablero[i, j] = true;
    total += UbicaReinas(n, i + 1, tablero);
    tablero[i, j] = false;
  }

  return total;
}
```

Listado 8-20 Contando la cantidad posible de formas de ubicar N reinas

Draft para estudiantes de MATCOM

El Listado 8-20 muestra una implementación para contar de cuántas formas diferentes se pueden ubicar las N reinas en un tablero. Nótese que el método recursivo `UbicaReinas` recibe un parámetro `i` que indica cuál es la reina que debe ser ubicada en esa llamada. El ciclo `for` explora todas las posibles casillas donde ubicar a una reina haciendo `tablero[i,j]=true` y realizando la llamada recursiva para ubicar a la reina $i+1$, y luego se "deshace" la acción tomada haciendo `tablero[i,j]=false`. En el caso base, cuando se determina (`i==n`) que se han puesto N reinas en el tablero, lo que se hace es simplemente verificar si las posiciones en que están las reinas son válidas (no se amenazan). El método `EsValido` (Listado 8-21) determina esto, empleando las técnicas vistas en el capítulo 7 sobre arrays para movernos en todas las direcciones.

```
private static bool EsValido(bool[,] tablero)
{
  int[] di = { 1, 1, 1, 0, 0, -1, -1, -1 };
  int[] dj = { 0, 1, -1, 1, -1, 0, 1, -1 };

  for (int i = 0; i < tablero.GetLength(0); i++)
  {
    for (int j = 0; j < tablero.GetLength(1); j++)
    {
      if (!tablero[i, j])
        continue;

      for (int d = 0; d < di.Length; d++)
      {
        for (int k = 1; ; k++)
        {
          int ni = i + di[d] * k;
          int j = j + dj[d] * k;

          if (ni < 0 || ni >= tablero.GetLength(0) ||
              j < 0 || j >= tablero.GetLength(1))
            break;

          if (tablero[ni, j])
            return false;
        }
      }
    }
  }

  return true;
}
```

Listado 8-21 Verificando si un tablero es válido

Como es fácil de apreciar, esta implementación es bastante ineficiente, pues ubica hsta la última reina para luego descubrir que se amenazan entre sí (`EsValido` devuelve `false`). Como se ilustró en el ejemplo con las 4 reinas, sería mejor determinar si una posible ubicación para una reina ya no es válida porque de ponerla ahí entraría en conflicto con las ya ubicadas. Para ello basta con verificar la condición de tablero válido en el cuerpo del ciclo `for`, antes de hacer la llamada recursiva. De este modo ya no hay que verificar la validez en el caso base, porque solo se puede llegar a éste si ya se tuvo la certeza de que

Draft para estudiantes de MATCOM UH

esta última ubicación fue válida (Listado 8-22). Este tipo de estrategia, que consiste en "cortar" la recursividad antes de llegar a un caso base, se suele denominar "poda"[7].

```
private static int UbicaReinas(int n, int i, bool[,] tablero)
{
  if (i == n)
    return 1;

  int total = 0;

  // Tratar de poner la i-ésima reina en la j-ésima columna
  for (int j = 0; j < n; j++)
  {
    tablero[i, j] = true;

    if (EsValido(tablero))
      total += UbicaReinas(n, i + 1, tablero);

    tablero[i, j] = false;
  }

  return total;
}
```

Listado 8-22 Podando la recursividad en las soluciones inválidas parciales

De forma general, toda solución empleando la técnica de backtrack tiene una estructura similar. Una solución se define a partir de una secuencia de decisiones (ubicar la i-ésima reina en este ejemplo), y para cada decisión se definen un conjunto de posibles opciones (cada una de las casillas de una fila en la que se puede ubicar una reina). El método recursivo se construye de forma que en cada llamada se toma solamente una decisión, iterando por cada una de las posibles opciones para esa decisión. Para explorar una posible opción, se realiza la acción correspondiente (que debe poderse deshacer), se invoca recursivamente para tomar la siguiente decisión, y luego se "deshace" la acción recién tomada.

8.8.2 SALIDA DEL LABERINTO

Otro ejemplo típico de aplicación de backtrack es encontrar la salida en un laberinto. Se considera un laberinto expresado mediante un array bidimensional `bool[,]`, donde los valores `true` representan obstáculos. Consideraremos que se parte de la posición (0,0) y se desea llegar a la posición (N-1,N-1). En la Figura 8.20 se muestra un ejemplo de laberinto y un posible camino de salida.

[7] Se han cortado (podado) las ramificaciones de caminos que pueden llevar a una solución.

Draft para estudiantes de MATCOM UH

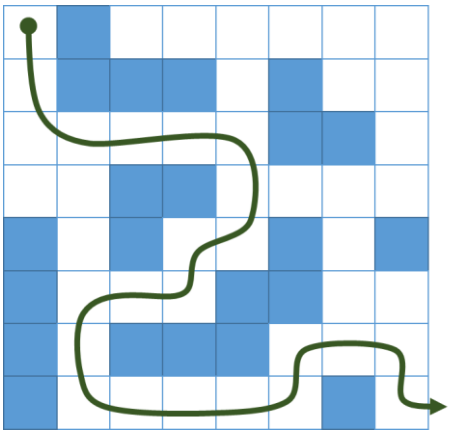

Figura 8.20 Ejemplo de una instancia del problema del laberinto y una posible solución (no necesariamente el camino más corto)

Para resolver este problema empleando backtrack, se debe considerar cuáles son las decisiones a tomar en cada paso, y cuáles son las posibles opciones para dicha decisión. Considere una solución a este problema como una secuencia de posiciones (i,j) adyacentes entre sí, de forma que la primera sea la (0,0) y la última la (N-1,N-1), en donde los valores de cada posición es `false` (no hay obstáculos en el camino). En este caso una decisión consiste en determinar, a partir de una posición (i,j), a cuál de las 4 posiciones adyacentes $(i+1,j)$, $(i-1,j)$, $(i,j+1)$ o $(i,j-1)$ moverse y ver si desde alguna de ellas hay salida.

¿Qué significa en este caso ejecutar y deshacer cada decisión? En cada llamada recursiva, cuál ha sido la secuencia de decisiones tomadas hasta el momento. Hay que poder determinar cuáles son las posiciones que se han escogido como camino de salida hasta el momento para no caer en ciclos infinitos. Para no volver a pasar de nuevo por una misma celda se "marcará" cada celda visitada como un obstáculo (poniendo dicha celda a `true`). De modo que al regresar en la recursividad, y explorar otro posible camino, hay que volver a "liberar" la celda que antes de la llamada recursiva fue marcada como obstáculo.

```
static bool HaySalida(bool[,] laberinto)
{
  return HaySalida(laberinto, 0, 0,
                   laberinto.GetLength(0) - 1,
                   laberinto.GetLength(1) - 1);
}

private static bool HaySalida(bool[,] laberinto, int orig_i, int orig_j,
                              int salida_i, int salida_j)
{
  if (orig_i == salida_i && orig_j == salida_j)
    return true;

  int[] di = {1, -1, 0, 0};
  int[] dj = {0, 0, -1, 1};

  for (int d = 0; d < di.Length; d++)
  {
    int i = orig_i + di[d];
```

```
    int j = orig_j + dj[d];

    if (i < 0 || i >= laberinto.GetLength(0) ||
        j < 0 || j >= laberinto.GetLength(1))
      continue;

    if (!laberinto[i, j])
    {
      laberinto[i, j] = true;

      if (HaySalida(laberinto, i, j, salida_i, salida_j))
        return true;

      laberinto[i, j] = false;
    }
  }

  return false;
}
```

Listado 8-23 Determinando si hay salida del laberinto mediante backtracking

En el Listado 8-23 se muestra una solución para este problema. Note que en este caso el método recursivo devuelve `bool`, y por tanto, al hacer la llamada recursiva, si la respuesta obtenida es `true`, ya se retorna inmediatamente al padre. Esto tiene el efecto de que una vez se encuentre un primer camino, no se exploran otros posibles porque no se deshacen las decisiones tomadas. Esto tiene el inconveniente de que, si se encuentra un camino de salida, no se deshacen las decisiones tomadas, es decir, le dejamos el laberinto cerrado al próximo explorador (note que no se ha "deshecho" la acción de la asignación `laberinto[i, j] = true`).

Si esto no bastase, porque se desean obtener todos los posibles caminos, o porque se quiere encontrar el camino de menor cantidad de pasos, entonces hay que modificar el código.

Una sencilla modificación consiste en usar una lista de posiciones para almacenar cada una de las posiciones por las que se va pasando. Esto nos permitirá no solo determinar si hay un camino y terminar, sino devolver cada camino encontrado (Listado 8-24). En este caso, la acción a realizar consiste no solo en marcar en el array la celda como `true` para no volver a pasar por ella, sino añadirla a la lista (`camino.Add(new Celda { I = i, J = j })`). Por tal motivo, al deshacer hay que no solo desmarcar en el array `laberinto`, sino además quitar la celda de la lista (`camino.RemoveAt(camino.Count - 1)`).

```
struct Celda
{
  public int I;
  public int J;
}

static List<Celda> EncontrarSalida(bool[,] laberinto)
{
  List<Celda> camino = new List<Celda>();

  if (HaySalida(laberinto, 0, 0,
```

Draft para estudiantes de MATCOM UH

```
                laberinto.GetLength(0) - 1,
                laberinto.GetLength(1) - 1,
                camino))
    return camino;

  return null;
}

private static bool HaySalida(bool[,] laberinto, int orig_i, int orig_j,
                              int salida_i, int salida_j, List<Celda> camino)
{
  if (orig_i == salida_i && orig_j == salida_j)
    return true;

  int[] di = { 1, -1, 0, 0 };
  int[] dj = { 0, 0, -1, 1 };

  for (int d = 0; d < di.Length; d++)
  {
    int i = orig_i + di[d];
    int j = orig_j + dj[d];

    if (i < 0 || i >= laberinto.GetLength(0) ||
        j < 0 || j >= laberinto.GetLength(1))
      continue;

    if (!laberinto[i, j])
    {
      laberinto[i, j] = true;
      camino.Add(new Celda { I = i, J = j });

      if (HaySalida(laberinto, i, j, salida_i, salida_j, camino))
        return true;

      laberinto[i, j] = false;
      camino.RemoveAt(camino.Count - 1);
    }
  }

  return false;
}
```

Listado 8-24 Salida del laberinto devolviendo además el camino encontrado

Los dos ejemplos antriores, ubicar las reinas en el tablero y encontrar la salida en un laberinto nos ilustran el uso de la técnica del backtracking y la recursividad para explorar posibles secuencias de decisiones que nos llevan a encontrar una solución válida, todas las soluciones válidas o encontrar entre las válidas la que sea la mejor posible.

De forma general, este "patrón" de backtracking siempre consiste en lo siguiente: para cada posible forma en la que se puede tomar una decisión, se toma ésta y se invoca recursivamente para tomar la siguiente decisión, y luego de regresar de la recursividad se "deshace" la acción correspondiente a la toma de la decisión. Es justamente esta capacidad para "deshacer" y explorar nuevas soluciones lo que hace de la aplicación

Draft para estudiantes de MATCOM UH

combinada del backtracking y la recursividad una herramienta muy poderosa en la solución de problemas complejos.

Las formas en que se organiza la secuencia de pasos que pueden llevar a una solución pueden cumplir determinadas características particulares bien definidas que se conocen como **combinatoria**. Esto nos permitirá distinguir algunas familias de problemas en que los problemas de cada familia pueden ser resueltos siguiendo un mismo patrón de solución. Este tema se presenta en los siguientes epígrafes.

8.9 EL PROBLEMA DEL VIAJANTE

Otro problema clásico de la programación es el denominado **Problema del viajante**. El término viene por analogía a la de un viajante de comercio, que debe visitar N ciudades para promocionar sus productos pero quiere determinar cuál es el recorrido que le genera menor costo.

Puesto que en los tiempos actuales la imagen del viajante con su maleta de productos a cuestas ya es obsoleta, planteémonos un problema similar. Considere una agencia de prensa que tiene corresponsales en N ciudades. Se puede viajar directamente de una ciudad a otra con un boleto que dependiendo del trayecto y del origen y el destino puede tener un costo diferente en cada caso. La Figura 8-21 nos muestra los costos de un boleto[8] entre las ciudades París, Zurich, Madrid, Berlín y Roma.

La agencia quiere hacer un reportaje para el que necesita que uno de los corresponsales recorra todas las ciudades; el problema consiste en encontrar cuál es el menor costo posible con el que un reportero puede hacer tal recorrido.

	París	Zurich	Madrid	Berlín	Roma
París	0	40	50	40	70
Zurich	60	0	40	55	30
Madrid	45	35	0	75	25
Berlín	50	55	65	0	80
Roma	65	30	45	85	0

Figura 8-21 Tabla de costos de boleto

Un recorrido completo de menor costo es Madrid, Roma, Zurich, París, Berlín que tiene un costo total de 40 + 50 + 25 + 30, para un total de 155.

Claro que para este ejemplo de pocas ciudades la solución puede encontrarse prácticamente por tanteo. Pero ¿cómo plantear una solución general?

La solución de fuerza bruta sería intentar todas las **combinaciones** de posibles de recorridos, calcular los costos de cada uno y quedarnos con el menor. En este caso, las

[8] Claro que éstos son costos hipotéticos en aerolíneas Low Cost y en los que el costo de un mismo trayecto puede ser distinto según el origen y el destino

combinaciones que se quieren probar deben pasar por todas las ciudades sin repetir ninguna.

Cuando se quieren tener todas las combinaciones y que cada una tenga todos los valores, esto es lo que se llama una **permutación**.

Note que no se puede escribir un código iterativo, basado en ciclos, que valga para un *N* cualquiera, porque no se pueden escribir *N* ciclos anidados si no se conoce a priori el valor de *N*.

Para escribir un código que genere todas las permutaciones, vamos a echar mano a la recursividad y del backtracking que vimos anteriormente. El código del Listado 8-25 nos ilustra esto.

El array `ciudades` que se recibe como parámetro tiene la secuencia de ciudades de las que se quieren generar las permutaciones para calcular su costo. Se llama a un primer método `Viajante`, pasándole un array con valores de 0 a *N-1* dando un número para cada ciudad. Para los datos del ejemplo sería:

```
string[] nombres = {"París", "Zurich", "Madrid", "Berlín", "Roma"};

int[] ciudades = {0, 1, 2, 3, 4};

Viajante(ciudades);
```

El método `Viajante` es el que a su vez hará la llamada

```
Viajante(ciudades, new int[ciudades.Length], 0, new bool[ciudades.Length]);
```

Esta sobrecarga del método recursivo `Viajante` (Listado 8-25) tiene además como parámetros un segundo array `recorrido` donde se va a ir construyendo cada uno de los posibles recorridos, un parámetro `cuantas` que nos va a ir diciendo cuántos valores ya hay puestos en la permutación que se está construyendo en `recorrido`, y un array `visitada` de tipo `bool[]` donde se indica (con valor `true`) si una ciudad ya ha sido incluida en el recorrido que se está construyendo para que no se vuelva a incluir en el resto de la construcción de ese recorrido que irá haciendo el desencadenamiento de la secuencia de llamadas recursivas hasta llegar al caso base.

El caso base de la recursividad será entonces

```
if (cuantas == ciudades.Length)
  {
    //...Acción a realizar con el recorrido que se ha construido
  }
```

en el que se incluirá el procesamiento que quiera hacerse con el recorrido que se ha construido.

En cada iteración del ciclo `for` se formarán todos los recorridos que incluyan a `ciudades[k]` en la posición `cuantas` del array `recorrido`, indicando en el array `visitada` que ya esa ciudad forma parte del recorrido para que no sea considerada en la secuencia de llamadas recursivas que se producirá a continuación:

```
visitada[k] = true;
recorrido[cuantas] = ciudades[k];
Viajante(ciudades, recorrido, cuantas+1, visitada)
```

Draft para estudiantes de MATCOM UH

Note que en la llamada se pasa el valor `cuantas+1` porque esa ciudad se ha incluido en el recorrido. Al regresar de la llamada recursiva (porque finalmente se alcanzó el caso base y se completó la generación de un recorrido) se vuelve a asignar `false` a `visitada[k]` (backtracking) para que en la próxima iteración del ciclo se terminen de generar los recorridos que tienen un nuevo valor en la posición `recorrido[cuantas]`. Este "backtracking" no tiene que "quitar" el valor que se había puesto en `recorrido[cuantas]` porque el nuevo valor sobreescribirá a éste.

```
static void Viajante(int[] ciudades)
{
  Viajante(ciudades, new int[ciudades.Length], 0, new bool[ciudades.Length]);
}
static void Viajante(int[] ciudades, int[] recorrido, int cuantas, bool[] visitada)
{
  if (cuantas == ciudades.Length)
  {
    //...Acción a realizar con una permutación
  }
  else
  {
    for (int k = 0; k < ciudades.Length; k++)
    {
      if (!visitada[k])
      {
        visitada[k] = true;
        permutacion[cuantas] = ciudades[k];
        Viajante(ciudades, recorrido, cuantas+1, visitada);
        visitada[k] = false;
      }
    }
  }
}
```

Listado 8-25 Generación de las permutaciones de n ciudades para el problema del viajante

Para solucionar el problema del viajante, una vez que se ha formado un recorrido hay que calcular su costo y quedarnos con ese recorrido si es menor que el menor calculado hasta el momento.

El
```
static int[] ciudades;
static string[] nombres;
static int[,] costos;
static int menorCosto;
static int[] mejorRecorrido;

static int SumaCostos(int[] recorrido, int m)
{
  //...
}

static void Viajante(int[] ciudades)
{
  Viajante(ciudades, new int[ciudades.Length], 0, new bool[ciudades.Length]);
}
static void Viajante(int[] ciudades, int[] recorrido, int cuantasVisitadas, bool[]
yaVisitada)
{
```

Draft, para estudiantes de MATCOM UH

```
  if (cuantas == ciudades.Length)
  {
    int suma = SumaCostos(recorrido, recorrido.Length);
    if (suma < menorCosto)
    {
      menorCosto = suma;
      System.Array.Copy(recorrido, 0, mejorRecorrido, 0, recorrido.Length);
    }
  }
  else
  {
    // ...
  }
}
```

Listado 8-26 nos muestra el código completo. En él se han incorporado las variables globales `menorCosto` y `mejorRecorrido` para ir dejando el resultado obtenido.

```
static int[] ciudades;
static string[] nombres;
static int[,] costos;
static int menorCosto;
static int[] mejorRecorrido;

static int SumaCostos(int[] recorrido, int m)
{
  //...
}

static void Viajante(int[] ciudades)
{
  Viajante(ciudades, new int[ciudades.Length], 0, new bool[ciudades.Length]);
}
static void Viajante(int[] ciudades, int[] recorrido, int cuantasVisitadas, bool[]
yaVisitada)
{
  if (cuantas == ciudades.Length)
  {
    int suma = SumaCostos(recorrido, recorrido.Length);
    if (suma < menorCosto)
    {
      menorCosto = suma;
      System.Array.Copy(recorrido, 0, mejorRecorrido, 0, recorrido.Length);
    }
  }
  else
  {
    // ...
  }
}
```

Listado 8-26 Código a añadir para quedarnos con la combinación de menor costo

Para aplicar esta solución a los datos del ejemplo habría que hacer lo que se muestra en el Listado 8-27, que dará el resultado del recorrido Madrid, Roma, Zurich, París, Berlín y el costo de 155.

```
static void Main(string[] args)
{
  costos = new int[,] { { 0, 40, 50, 40, 70},
```

Draft para estudiantes de MATCOM UH

```
                        {60,  0, 40, 55, 30},
                        {45, 35,  0, 75, 25},
                        {50, 55, 65,  0, 80},
                        {65, 30, 45, 85,  0}};
  nombres = new string[] { "París", "Zurich", "Madrid", "Berlín", "Roma" };
  menorCosto = int.MaxValue;

  ciudades = new int[] { 0, 1, 2, 3, 4 };
  mejorRecorrido = new int[ciudades.Length];
  Viajante(ciudades);
  Console.WriteLine("\nEl mejor camino es ");
  for (int k = 0; k < mejorRecorrido.Length; k++)
    Console.Write("{0}  ", nombres[mejorRecorrido[k]]);
  Console.WriteLine("\nA un costo en boletos de {0}", menorCosto);

}
```

Listado 8-27 Aplicación de la solución del viajante para los datos de la Figura 8-21

8.9.1 LA INTRATABILIDAD DEL PROBLEMA DEL VIAJANTE. LA ESTRATEGIA DEL GOLOSO

El costo de la solución anterior al problema del viajante aumenta considerablemente en la medida que aumenta la cantidad *N* de ciudades.

Si podemos empezar por cualquiera de las *N* ciudades, entonces a partir de cada una de esas podemos ir a cualquiera de las *N-1* restantes, por lo que tenemos *N*(N-1)* combinaciones posibles. De cada una de éstas podemos ir a las *N-2* restantes y así sucesivamente, por lo que tenemos *N*(N-1)*(N-2)* ... 1* posibles recorridos, es decir *N*!.

Para una cantidad de 5 ciudades como la del ejemplo se tendrían 120 posibles recorridos, lo cual es una cantidad de combinaciones viables de probar para encontrar una que sea la de menor costo. Sin embargo, si fuesen por ejemplo 20 ciudades (lo cual no parece ser una cantidad exagerada de ciudades), el valor sería 20!, es decir 2432902008176640000 recorridos. Para que se tenga una idea de esta magnitud, calcule que si se tuviese un CPU capaz de calcular 10,000 millones de recorridos por segundo esto consumiría unos 7.7 años.

De modo que una solución como la anterior se dice intratable, porque la cantidad de permutaciones es muy grande para poder aplicar una solución de fuerza bruta que las genere y pruebe todas. Hay que buscar entonces una vía de solución alternativa.

Una estrategia que suele aplicarse a problemas como el del viajante es la que se conoce como **estrategia del goloso**(greedy). Esta estrategia se basa en que intuitivamente parece sensato partir de la ciudad que nos ofrezca el trayecto de menor costo y luego aplicar el mismo razonamiento con la siguiente ciudad hasta haber recorrido todas las ciudades. Sin embargo, esto no nos da necesariamente el mejor recorrido, porque cada vez van quedando menos opciones a escoger, y por tanto puede suceder que por haber escogido inicialmente un trayecto más barato luego no nos quede más opción que escoger uno muy caro. Si aplicamos esta estrategia a las ciudades de la Figura 8-21, se tendría que el trayecto inicial más barato es 25 (Madrid-Roma), luego partiendo de Roma el siguiente trayecto más barato es el Roma-Zurich que es de costo 30, después el Zurich-Berlín que cuesta 55, y por último ya estamos obligados a hacer el Berlín-París que es 50. El total de este recorrido es 25 + 30 + 55 + 50 para un total de 160, que es un valor mayor que la mejor solución, de costo 155.

Draft para estudiantes de MATCOM UH

Se deja al lector que programe una solución basada en la estrategia del goloso.

Aunque no nos de la mejor solución, la estrategia del goloso puede ser útil para combinarla con otra estrategia, como veremos en el subepígrafe a continuación.

8.9.2 DISMINUYENDO LA CANTIDAD DE RECORRIDOS A ANALIZAR

Para lidiar con la intratabilidad de la solución al problema del viajante, tenemos que aplicar alguna estrategia que disminuya la cantidad de combinaciones a probar.

Si mientras se va generando un recorrido se va calculando a la par el costo del mismo y ese costo parcial llegase a ser mayor que el costo total menor encontrado hasta ese momento, entonces ya se debería terminar de generar los posibles recorridos que continúen a partir de ahí. Note en el `static void Viajante(int[] ciudades, int[] recorrido, int cuantas, bool[] visitada)`

```
{
  if (cuantas == ciudades.Length)
  {
    int suma = SumaCostos(recorrido, recorrido.Length);
    if (suma < menorCosto)
    {
      menorCosto = suma;
      System.Array.Copy(recorrido, 0, mejorRecorrido, 0, recorrido.Length);
    }
  }
  else
  {
    for (int k = 0; k < ciudades.Length; k++)
    {
      if (!visitada[k])
      {
        if (SumaCostos(recorrido, cuantas) < menorCosto)
        {
          visitada[k] = true;
          permutacion[cuantas] = ciudades[k];
          Viajante(ciudades, recorrido, cuantas+1, visitada);
          visitada[k] = false;
        }
      }
    }
  }
}
```

Listado 8-28 que si la suma del recorrido generado hasta el momento `SumaCostos(recorrido, cuantas)` no es menor que `menorCosto`, entonces no se hace la llamada recursiva porque no es necesario continuar generando los recorridos que comienzan con éste.

```
static void Viajante(int[] ciudades, int[] recorrido, int cuantas, bool[] visitada)
{
  if (cuantas == ciudades.Length)
  {
    int suma = SumaCostos(recorrido, recorrido.Length);
    if (suma < menorCosto)
    {
      menorCosto = suma;
```

Draft para estudiantes de MATCOM UH

```
        System.Array.Copy(recorrido, 0, mejorRecorrido, 0, recorrido.Length);
      }
    }
    else
    {
      for (int k = 0; k < ciudades.Length; k++)
      {
        if (!visitada[k])
        {
          if (SumaCostos(recorrido, cuantas) < menorCosto)
          {
            visitada[k] = true;
            permutacion[cuantas] = ciudades[k];
            Viajante(ciudades, recorrido, cuantas+1, visitada);
            visitada[k] = false;
          }
        }
      }
    }
}
```

Listado 8-28 Problema del viajante mejorado descartando permutaciones a probar

Si se aplica esta estrategia a los datos del ejemplo, se llegan a generar solo 67 recorridos completos en lugar de los 120 que es el valor de 5!

Si se combina esta solución tomando como valor inicial para la variable `menorCosto` el valor 160, que es el resultado que produce la estrategia del goloso, la cantidad de recorridos que es necesario generar completamente sería en este caso de 63.

El problema del viajante es el más famoso de los problemas de la familia "NP-Duros", denominados así por la complejidad de encontrar una solución que pueda ejecutarse en un tiempo razonable.

Una exposición detallada de las familias de problemas P, NP, NP-Completos y NP-Duros se sale de los objetivos de este libro, aunque en el Capítulo 9, "Complejidad de los algoritmos", serán abordados brevemente.

8.10 COMBINATORIA

El caso anterior del problema del viajante no es más que un caso particular de un tipo de problemas en los que para encontrar la solución hay que generar diferentes secuencias finitas de valores formadas con los valores de un conjunto base de N elementos. En este epígrafe vamos ver algunas formas de **combinar** los valores para la solución de diferentes tipos de problemas.

Para ilustrarlo genéricamente, vamos a hablar de las diferentes formas en que se pueden formar secuencias de pelotas (que están numeradas consecutivamente en un cajón de pelotas).

Draft para estudiantes de MATCOM UH

8.10.1 PERMUTACIONES

El caso más conocido es cuando las secuencias a formar tienen igual longitud que la cantidad de valores del conjunto base. Estas secuencias se llaman **permutaciones** y fueron las generadas en el problema del viajante.

¿Cuántas/cuáles son las formas distintas en que se pueden sacar las N pelotas del cajón y ubicarlas en una secuencia? (Figura 8-22)

En este caso tenemos N elementos de los que tomar, tenemos N lugares que llenar, el orden importa y no se pueden repetir elementos.

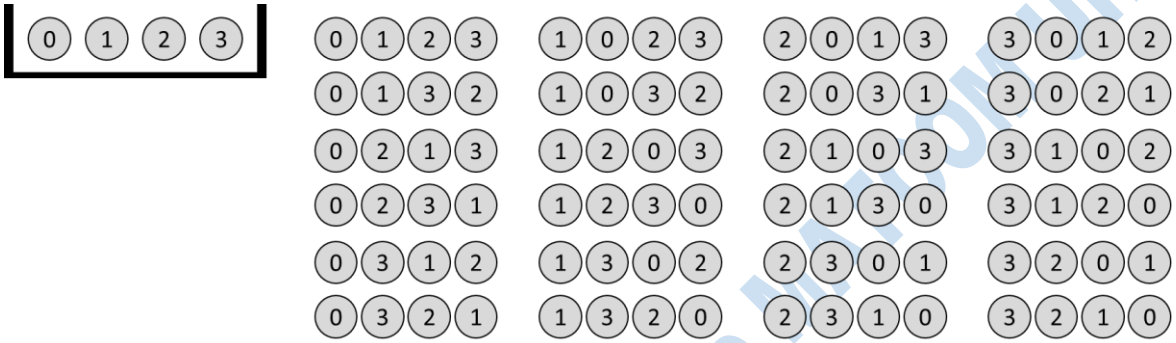

Figura 8-22Permutaciones para N=4

Para este caso es fácil calcular la cantidad de formas posibles. Puesto que el orden importa, podemos contar utilizando el principio de multiplicación. En la primera posición podemos ubicar cualquiera de las N pelotas distintas. Por cada una de estas formas, en la segunda posición se puede ubicar cualquiera de las $N-1$ pelotas restantes. Luego en la tercera cualquiera de las $N-2$, y así sucesivamente. Por tanto, la cantidad de formas distintas en que las N pelotas se pueden ubicar en una secuencia es:

$$P_N = N * (N-1) * \ldots * 2 * 1 = N!$$

La implementación (Listado 8-29) es similar que la que usamos en el problema del viajante. Note que se ha prescindido del parámetro array con los valores de las pelotas de 0 a $N-1$ porque para ello podemos usar la propia variable `k` de control del ciclo.

```
static void Permutaciones(int[] permutacion, int cuantas, bool[] utilizada)
{
  if (cuantas == permutacion.Length)
  {
    //...Acción a realizar con una permutación
  }
  else
  {
    for (int k = 0; k < permutacion.Length; k++)
    {
      if (!visitada[k])
      {
        utilizada[k] = true;
        permutacion[cuantas] = k;
        Permutaciones(permutacion, cuantas+1, utilizada);
        utilizada[k] = false;
      }
    }
```

Draft para estudiantes de UH. ©WEBOO

```
    }
}
```

Listado 8-29 Generación de permutaciones

No hay pérdida de generalidad en usar los valores enteros de 0 a $N-1$; si en un problema real trabajamos con objetos de tipo `T` (como fue el caso de `string` cuando estábamos hablando de ciudades), basta con tener un array `T[]` que permita asociar cada índice en la secuencia `permutacion` con su correspondiente valor de tipo `T`.

8.10.2 VARIACIONES SIN REPETICIÓN

Supongamos ahora que se quieren formas secuencias de pelotas pero no necesariamente de la longitud N, donde N es la cantidad original de pelotas diferentes. Este podría ser el caso, por ejemplo, del problema del viajante, pero que en lugar de tener que recorrer todas las ciudades se deba visitar una cantidad M (donde $M <= N$) de ciudades.

Planteado genéricamente: *¿Cuántas/cuáles son las formas distintas en que se pueden sacar M pelotas del cajón y ubicarlas en una secuencia?*

En este caso tenemos N pelotas diferentes donde escoger, M lugares que llenar, el orden importa y no se pueden repetir pelotas.

La Figura 8-23 nos muestra las variaciones sin repetición de 4 pelotas tomadas de 2 en 2.

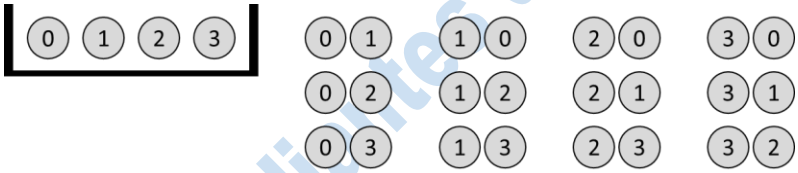

Figura 8-23 Variaciones sin repetición de 4 a tomados de 2 en 2

La cantidad de estas formas se puede calcular con un análisis similar al que se hizo con las permutaciones. En este caso la diferencia radica en que solo se multiplican los factores desde N hasta $N-M+1$, quedando:

$$V_M^N = N * (N-1) * \ldots * (N-M+1) = \frac{N!}{(N-M)!}$$

Note que realmente las permutaciones son un caso particular de variaciones sin repetición formando secuencias de exactamente N elementos.

$$P_N = V_N^N$$

Para obtener las **variaciones sin repetición**, el único cambio que habría que hacer al código de las permutaciones es considerar que la solución puede ser de menor tamaño que N y por tanto el algoritmo tendría que recibir un parámetro inicial `N` que da la cantidad de pelotas y la cantidad de pelotas M a poner en cada variación sería la longitud del array `variacion`, que es donde se iría formando la variación. El código quedaría como muestra el Listado 8-30.

```
static void VariacionesSinRepeticion(int N, int[] variacion, int cuantas,
                                      bool[] utilizada)
{
```


Draft para estudiantes de MATCOM UH

```
  if (cuantas == variacion.Length)
  {
    //...Acción a realizar con una variación
  }
  else
  {
    for (int k = 0; k < N; k++)
    {
      if (!visitada[k])
      {
        utilizada[k] = true;
        variacion[cuantas] = k;
        VariacionesSinRepeticion(variacion, cuantas+1, utilizada);
        utilizada[k] = false;
      }
    }
  }
}
```

Listado 8-30 Código para obtener todas las variaciones sin repetición de N elementos tomados de M en M

Analice el siguiente problema y vea cómo pueden ser utilizadas las variaciones sin repetición para generar las posibles soluciones candidatas, cuál sería la implementación de la acción a aplicar cuando se ha generado una variación y cuáles son las condiciones iniciales que hay que preparar para iniciar la búsqueda de la solución (la llamada a la generación de las variaciones).

Se tiene los resultados de un torneo efectuado entre N jugadores en el que compitieron todos contra todos. El resultado se expresa en una matriz booleana de NxN que tiene en la posición (i,j) valor true si el jugador i le ganó a j o false en caso contrario. Se desea saber si se puede encontrar una secuencia de M jugadores tal que para todo jugador a y b en la secuencia, si a está antes que b en dicha secuencia entonces a le ganó a b.

8.10.3 COMBINACIONES

¿Cuántas/cuáles son las formas distintas en que se pueden formar secuencias de M pelotas sacando pelotas de un cajón que tiene N pelotas y en el que no importa la ubicación de una pelota en la secuencia?

En este caso tenemos *N* pelotas para escoger, tenemos *M* lugares que llenar, no importa el orden en que pongamos la pelota y no se pueden repetir pelotas.

La Figura 8-24 nos muestra (en oscuro) las distantas **combinaciones** de 3 pelotas que se pueden formar de un conjunto de 4 pelotas.

Draft para estudiantes de MATCOM UH

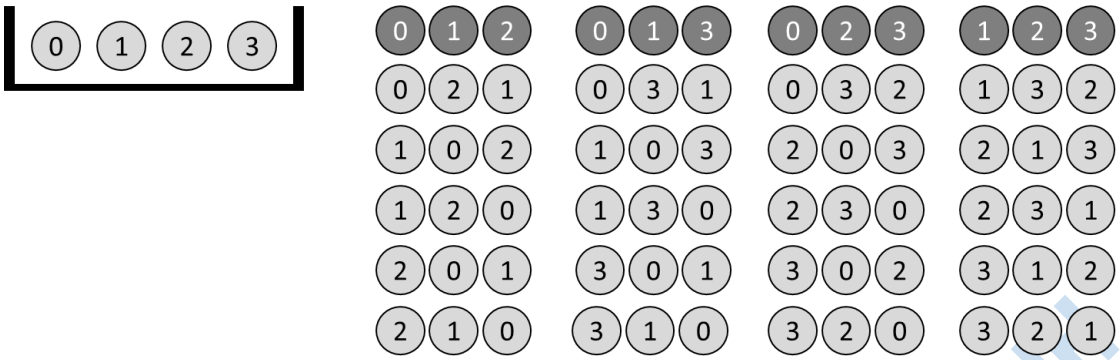

Figura 8-24 Combinaciones de 4 a tomar de 3 en 3 resaltado del resto de las variaciones.

Es fácil ver que las combinaciones pueden ser generadas a partir de las variaciones sin repetición. En efecto, cada combinación de M elementos aparecerá en M! variaciones. Es por ello que el número de combinaciones puede escribirse como:

$$C_M^N = \binom{N}{M} = \frac{V_M^N}{M!} = \frac{N!}{(N-M)! * M!}$$

Para generar las combinaciones se puede utilizar el código de las variaciones (ver Listado 8-30) evitando repetir todas las permutaciones de una misma combinación. Esto último se puede resolver si cuando se va formando una combinación solo se intentan poner valores mayores a los que ya se han puesto en la combinación que se está formando. Por ejemplo, si se están generando combinaciones de 10 pelotas tomadas de 4 en 4 y se estuviese formando la combinación que empieza con 5, 7 ya no habría que generar combinaciones como 5,7,1,2 ó 5,7,4,6, porque respectivamente las combinaciones 1,2,5,7 y 4,5,6,7 ya se hubiesen generado antes.

El código siguiente solo tiene que mantener entonces en el parámetro `menorAPoner` el valor menor de los que se pueden continuar poniendo en una combinación, de modo que ya no es necesario usar un array `utilizada` para llevar la cuenta de las pelotas que ya han sido puestas en la combinación.

```
static void Combinaciones(int N, int[] combinacion, int cuantas, int menorAPoner)
{
    if (cuantas == combinacion.Length)
    {
        //... Invocar acción a realizar con la combinacion
    }
    else
    {
      for (int k = menorAPoner; k < N; k++) {
        combinacion[cuantos] = k;
        Combinaciones(N, combinacion, cuantos + 1, k + 1);
      }
    }
}
static void Combinaciones(int N, int M) {
  Combinaciones(N, new int[M], 0, 0);
}
```

Listado 8-31 Código para generar las combinaciones de N en M

Draft para estudiantes de MATCOM UH

Note que cada llamada recursiva se le pasa al parámetro `menorAPoner` el valor `k+1`, que es mayor que `k` (el último que se ha puesto) y en la primera invocación se le ha pasado el valor `0`.

Una mejora que puede hacerse al código del Listado 8-31 es que el ciclo `for` no tiene por qué iterar hasta el valor `N-1`. Si en la combinación que se está formando se han colocado `cuantos` valores, entonces quedarán por poner `combinacion.Length-cuantos`. Por tanto el ciclo debe iterar mientras `k < N - (combinación.Length - cuantos) + 1`, o sea mientras `k <= N - combinación.Length + cuantos`

8.10.4 CONJUNTO POTENCIA DE COMBINACIONES

Hay problemas en los que no siempre se puede conocer la longitud M de la secuencia que puede servir de solución. En tales casos, necesitaremos recorrer todas las posibles secuencias de las diferentes longitudes. La Figura 8-25 ilustra los valores del conjunto potencia de un conjunto de 4 pelotas (incluyendo el conjunto vacío).

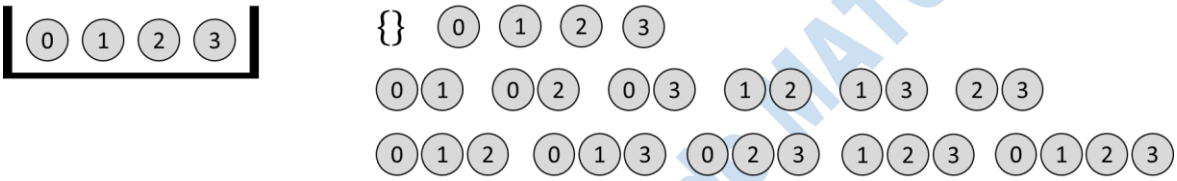

Figura 8-25 Secuencias del conjunto potencia de 4 pelotas

Para generar todos los posibles subconjuntos de secuencias bastaría con invocar repetidamente al código de generación de las combinaciones con todos los valores posibles de M entre 0 (conjunto vacío) y N (conjunto con una sola combinación formada por todos los valores). El Listado 8-32 muestra esta solución.

Este enfoque es preferible si el problema se puede plantear en términos de encontrar la secuencia de menor cantidad de elementos que cumpla con cierta condición.

```
static void GeneraConjuntoPotencia(int N)
{
  for (int M = 0; M <= N; M++)
    Combinaciones(N, M);
}
```

Listado 8-32 Generación del conjunto potencia utilizando combinaciones

Una forma más sencilla de expresar todas las posibles secuencias del conjunto potencia es mediante un array `bool[]` de longitud igual a la cantidad de valores del conjunto base, de modo que un valor `true` en el elemento en la posición `i` quiere decir que el valor `i` está en la secuencia. En este caso, si el conjunto base es el de las pelotas `{0,1,2,3}`, la secuencia `{1, 3}` quedaría representada por el array `{false, true, false, true}`.

El código del Listado 8-33 nos muestra cómo pueden ser generadas las secuencias del conjunto potencia de N valores, donde N es la longitud del array `secuencia`. La primera llamada recursiva produce las secuencias (de diferentes longitudes) en las que el valor `i` no está en la secuencia; luego se hace `solución[i]=true` para que la siguiente llamada recursiva genere las secuencias en las que `i` está.

```
static void GeneraConjuntoPotencia(bool[] combinacion, int i) {
```

Draft para estudiantes de MATCOM UH

```
  if (i == combinacion.Length) {
    Accion(combinacion); //... Acción para procesar dicha combinacion
  }
  else
  { // Suponer que i no debe estar en la combinación
    GeneraConjuntoPotencia(combinacion, i + 1);
    // suponer que i debe estar en la combinación
    combinacion [i] = true;
    GeneraConjuntoPotencia(combinacion, i + 1);
    combinacion [i] = false; // restaurar el estado de la solución
  }
}

static void GeneraConjuntoPotencia(int N) {
  GeneraConjuntoPotencia(new bool[N], 0);
}
```

Listado 8-33 Generación del Conjunto Potencia representado con un array booleano

Esta solución tiene el incoveniente de que no genera las secuencias ordenadamente, es decir, con todas las secuencias de una misma longitud antes de todas las de la siguiente longitud, por lo que no es la conveniente a aplicar cuando se quiere obtener la secuencia de menor longitud que cumpla con los requerimientos del problema.

Existen diversos problemas clásicos que tienen su solución en la exploración del conjunto potencia. En los subepígrafes a continuación se ilustran dos de ellos y la forma en que pueden ser resueltos.

8.10.4.1 PROBLEMA DE LA MOCHILA

Suponga que se tiene una mochila con capacidad de volumen C. Se tiene un conjunto de N de valores que enumeramos del 0 a N-1. Para cada elemento i del conjunto se obtiene una ganancia g_i por ser escogido y guardado en la mochila (puede interpretarlo como que g_i es el valor de un cierto producto) y a su vez se conoce el volumen que ocupa dicho elemento en la mochila (v_i). El problema de la mochila (*Knapsack problem*) consiste en determinar cuál es la combinación de productos que sea la de mayor valor, pero que a su vez por su volumen total quepa en la mochila. Tal combinación la expresaremos con un array `bool[]` de longitud N en el que valor en la posición i será `true` si el elemento i debe ser incluido en la mochila y `false` en caso contrario.

$$\max \sum g_i x_i$$

Sujeto a la condición de que los elementos seleccionados quepan en la mochila:

$$\sum v_i \, x_i \leq T$$

Usando la generación de combinaciones del conjunto potencia del Listado 8-33, para cada combinacion generada habría que invocar el método `Accion(combinacion)`. En este caso `Accion` debe verificar si por el volumen total esa combinación cabe en la mochila, y en tal caso calcular su valor y si es mayor que el mayor valor que haya sido obtenido hasta ahora entonces quedarnos con dicha combinación (Listado 8-34).

```
static int mejorGanancia;
static int capacidad;
```

Draft para estudiantes de MATCOM UH

```
static int[] volumenes;
static int[] ganancias;
static int[] mejorCombinacion;

static void Mochila(int C, int[] g, int[] v) {
    mejorGanancia = 0;
    capacidad = C;
    ganancias = g;
    volumenes = v;
    GeneraConjuntoPotencia(g.Length);
}
static void Accion(bool[] combinacion) {
  int volumenCombinacion = 0; // volumen total de la combinación seleccionada
  int gananciaCombinacion = 0; // ganancia total de la combinación seleccionada
  for (int i = 0; i < combinacion.Length; i++)
    if (combinacion[i])
    {
       volumenCombinacion += volumenes[i];
       gananciaCombinacion += ganancias[i];
    }
  if (volumenCombinacion <= capacidad)
  { // la combinación cabe en la mochila
    if (gananciaCombinacion > mejorGanancia)
    {
      mejorGanancia = gananciaCombinacion;
      System.Array.Copy(combinacion, 0, mejorCombinacion, 0, combinacion.Length);
    }
  }
}
```

Listado 8-34 Solución al problema de la mochila reutilizando la generación del conjunto potencia

Note la característica genérica de esta solución: por una parte tenemos el método `GeneraConjuntoPotencia` que nos genera las diferentes combinaciones para buscar la que venga mejor al problema a resolver y por otra parte tenemos el método `Accion` que se aplica a cada combinación. El método `Accion` se ha definido con una signatura sin parámetros para garantizar que pueda ser utilizado genéricamente desde `GeneraConjuntoPotencia` sin modificar el código de éste; por eso la información que necesita cada implementación de `Accion` para cada problema particular se han trabajado con variables estáticas globales. El interesado en aplicar la solución de la mochila a su problema particular es quien debe aportar los parámetros de la llamada a `Mochila` según su conocimiento del dominio del problema: la capacidad volumen de la mochila, el array con las ganancias de cada producto y el array con los volúmenes de cada producto.

La facilidad de reuso que por un lado nos propicia este enfoque genérico de la solución, por otro lado nos limita a aprovechar el conocimiento de un problema particular para introducir mejoras en el algoritmo. Por ejemplo, en este caso de la mochila se pueden descartar combinaciones que se vayan formando si ya alcanzan un volumen que sobrepasa a la capacidad.

Una vez que usted domine cómo generar el conjunto potencia, puede adaptar éste según las características del problema que esté resolviendo y mezclar la lógica de la generación de las combinaciones con la mejora para descartar aquellas combinaciones cuando se detecten que ya sobrepasan la capacidad (Listado 8-35).

Draft para estudiantes de MATCOM-UH

```
static void Mochila(int Capacidad, int[] ganancias, int[] volumenes,
                    bool[] combinacion, int i, int gananciaActual,
                    int volumenActual, ref int mejorGanancia,
                    ref bool[] mejorCombinacion)
{
  if (i == ganancias.Length) {
    mejorGanancia = Math.Max(mejorGanancia, gananciaActual);
    System.Array.Copy(combinacion, 0, mejorCombinacion, 0, combinacion.Length);
    return;
  }
  // Suponer que el elemento i no estará en la combinación
  Mochila(Capacidad, ganancias, volumenes, combinacion,
          i + 1, gananciaActual, volumenActual, ref mejorGanancia);
  // Determinar si el elemento i pudiera estar en la combinación
  if (volumenActual + volumenes[i] <= Capacidad)
  { // Suponer que el elemento i estará en la combinación
    combinacion[i] = true;
    Mochila(Capacidad, ganancias, volumenes, combinacion,
            i + 1, gananciaActual + ganancias[i], volumenActual + volumenes[i],
            ref mejorGanancia);
    combinacion[i] = false;
    //Backtrack para probar con otra combinación que no lo incluya
  }
}
```

Listado 8-35 Solución mejorada al Problema de la mochila

En este caso, el array con la solución devuelve dos valores a través de los parámetros `mejorGanancia` y `mejorCombinacion`. Note que el uso del array `combinacion` solo es necesario si en el resultado queremos saber también cuál es la combinación que genera la mejor ganancia. Si solo nos interesara saber cuál es la mayor ganancia, entonces se podría prescindir de usar el array `combinacion`.

8.10.5 VARIACIONES CON REPETICIÓN

Hasta ahora hemos analizado problemas en los que en las secuencias que pueden ser solución no se repiten los valores del conjunto inicial.

Dado un valor positivo N se tiene un cajón con cualquier cantidas de pelotas de las numeraciones del 0 al N - 1 se quieren formar las secuencias de M pelotas en las que pueden haber pelotas con la misma numeración (Figura 8-26)

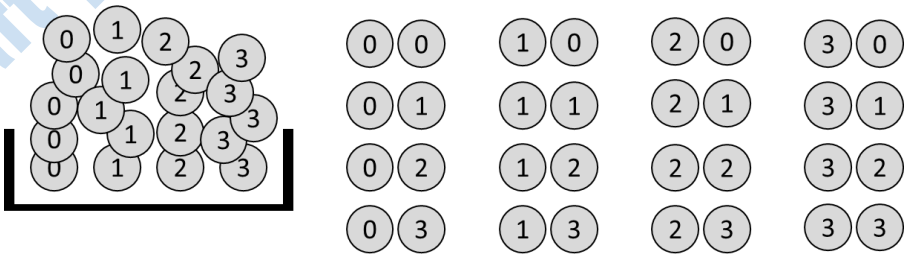

Figura 8-26 Variaciones con repetición de 4 a tomar de 2 en 2

En este tipo de combinaciones se toman de *N* elementos para ubicar en *M* posiciones, los elementos se pueden repetir y el orden es importante (es decir 0,1 y 1,0 son

Draft para estudiantes de MATCOM UH

secuencias distintas). Es fácil demostrar por el principio de la multiplicación que si en cada posición de la secuencia de M elementos puede estar cualquiera de los N elementos, la cantidad total de formas es igual a:

$$V_{R_M}^N = N * N * \ldots * N \ (M \ veces) = \ N^M$$

El código para la generación de estas variaciones (Listado 8-36) es más simple que el de las variaciones sin repetición, puesto que no necesita verificar si el valor que se toma para cada posición se repite. Incluso, como se parte de la disponibilidad de cualquier cantidad de pelotas el valor M (longitud de las secuencias) puede ser mayor que N (la cantidad de valores diferentes).

```
static void VariacionesConRepeticion(int N, int M)
{
  VariacionesConRepeticion(N, new int[M], 0);
}
static void VariacionesConRepeticion(int N, int[] variacion, int cuantas) {
    if (cuantas == variacion.Length)
    {
      Accion(variacion); //Código para procesar la variación
      return;
    }
    for (int k = 0; k < N; k++)
    {
      variacion[cuantas] = k;
      VariacionesConRepeticion(N, variacion, cuantas + 1);
    }
}
```

Listado 8-36 Generación de las variaciones con repetición

Se invita al lector a resolver el siguiente problema:

¿Cómo se pudiera solucionar el problema de la mochila si se permite colocar en la mochila más de un producto de un mismo tipo?

8.11 APOSTILLA

¿Cómo se puede saber que la solución a un problema se puede plantear de manera recursiva?

Si P son los datos de entrada a través de los cuales se plantea un problema y S son los datos de salida con los que se expresa la solución del mismo, desarrollar una solución usando un algoritmo recursivo debe verse en términos de cómo se transforman P y S entre una llamada recursiva y otra.

Como se ha visto en este capítulo, una estrategia muy utilizada por el sentido común es reducir el tamaño o complejidad de P, obteniendo uno o varios subproblemas p (problemas de menor magnitud, más simples) y determinar cómo se pueden combinar las soluciones obtenidas para cada uno de los subproblemas para con ello obtener la salida S solución del problema original. Un caso base o caso de parada de la recursividad serán aquellos subproblemas que por su simplicidad ya se pueden solucionar "de manera directa", sin necesidad de hacer una llamada recursiva. Esta estrategia es la que se ha denominado **divide y vencerás**.

Draft para estudiantes de MATCOM UH

Otra estrategia para solucionar un problema es usar la recursividad para explorar posibles soluciones parciales a *P* hasta que se encuentre una solución adecuada *S*. En esta estrategia no se parte de un problema *P* que se va fraccionando con cada llamada, sino que lo que se va haciendo en la secuencia de llamadas recursivas es ir construyendo la posible solución de manera incremental hasta llegar a un estado en que se pueda verificar si es adecuada como solución al problema. Si no es la adecuada, o si por el camino se puede determinar que no se llegará a una posible solución, entonces se vuelve atrás (***backtracking***) hasta un punto en que se pueda intentar explorar otro posible camino de solución. Si se agotan todos los posibles caminos sin encontrar una solución adecuada, entonces puede afirmarse que con esta implementación el problema *P* no tiene solución.

No existe una receta universal para obtener soluciones computacionales a los problemas, ni para determinar cuál enfoque o estrategia aplicar; de ser así no habrían problemas porque todo se podría automatizar. Lo que sí es bien conocido es que mientras más estrategias de solución se conozcan, como ha sido el caso de las expuestas en el epígrafe de combinatoria, entonces ante el planteamiento de un nuevo problema será más fácil encontrar una que por analogía se parezca al nuevo problema y que por tanto sirva de ayuda a implementar su solución (recuerde por ejemplo, cómo aplicar una estrategia de solución similar a la de la sucesión de Fibonacci sirvió para implementar la solución al problema de las lozas).

La recursividad tiene "magia" por el poder expresivo que encierra. Es una arma poderosa que ayuda a expresar de manera más simple y abreviada muchas soluciones, pero no es en sí misma una bala de plata[9] con la que se puede resolver cualquier problema si usted no la ha aplicado adecuadamente.

Draft para estudiantes de MATCOM UH

[9] El término bala de plata (*silver bullet*) es con frecuencia utilizado en la literatura computacional en inglés. Se refiere a que, según la leyenda, con una bala de plata se podía matar al hombre lobo (que sería en este caso el problema a resolver).