Teoría de la Recursividad
Capítulo 5: Análisis y solución de tres problemas

Ahora el lector conoce el mecanismo con que opera la recursividad. Estamos en condiciones entonces de analizar nuevos problemas, lo que haremos con una metodología "inversa" a la aplicada en el capítulo anterior:

Si en aquel partimos del programa para explicar su funcionamiento, ahora, una vez dado el enunciado, lo analizaremos "afinando" cada vez más las acciones necesarias, hasta llegar al programa que le da solución. Con ello, tratamos de reproducir el proceso mental necesario para llegar a resolver un problema; pero a la vez, será buena ocasión para profundizar algunos conceptos; sugerir criterios de trabajo; o mostrar nuevas ideas para enfrentar distintas situaciones. Al final del capítulo se espera que el lector pueda realizar el sólo los ejemplos que se presentan aquí.

Comenzamos con un problema muy conocido, un verdadero "clásico" para ilustrar el empleo de recursividad:

I. El problema "de las 8 damas".

Colocar n damas sobre un tablero de (NxN), de modo que dos cualesquiera de ellas no se amenacen mutuamente. Como en muchos casos vamos a reducir nuestro problema al análisis de uno en caso n = 8 pero dejándolo abierto para toda n. Esta idea es muy práctica ya que materializa las ideas y el problema se hace más fácil de entender.

Análisis.

1. Nuestra idea más general será la de ir colocando las damas una a una, comprobando siempre si la dama que estamos situando en determinado momento no amenaza a alguna de las colocadas anteriormente.
2. Parecería que lo lógico es representar el tablero por un arreglo de 8x8. Pero como en una fila solo se puede colocar una dama, basta un arreglo unidimensional f [0...7], que almacene, en cada fila, la posición que en ella ocupa la dama.
3. Procederemos ordenadamente: Primero, colocar una dama en la fila 0; luego en la 1; y así hasta colocar una en la fila 7. Esto nos sugiere que los "niveles" de trabajo están dados por las sucesivas filas donde iremos situando las damas.
4. En cada fila, comenzaremos colocando la dama lo más a la izquierda posible (posición 0). Inmediatamente, hay que comprobar si en esa posición, amenaza a alguna de las damas anteriores: SI: hay que continuar en el mismo nivel, avanzando la dama una posición más y volver a comprobar. NO: Si no amenaza a otra dama, entonces la posición actual es posible, y pasamos al nivel (fila) siguiente.
5. Cuando llegamos al "nivel" n, ya hemos colocado las n damas en posiciones adecuadas. Por tanto, tenemos una solución, que hay que imprimir.

6. El problema para n = 8 admite 92 soluciones. De ellas, 12 son “básicas” y las restantes son simétricas con las 12 básicas. Si nos interesa solo UNA solución, basta imprimir la primera que nos proporcione el programa y parar. Si nos interesan las 92, después de imprimir cada una, es aconsejable una pausa (`Console.ReadLine();`) de la que saldremos oprimiendo enter, para retornar al nivel anterior (nivel 7) a buscar una nueva solución.
7. A esta altura del análisis, ya va quedando claro que en nuestro programa hará falta:
   - Un método `void` para la impresión de soluciones.
   - Un método `bool` que nos indique si la posición actual de la dama que estamos situando es posible (`true`), o debe evitarse pues amenaza a alguna dama de filas anteriores (`false`).
   - El método recursivo que vaya colocando la dama de cada nivel en cada una de las 8 posiciones de su fila.
8. Es hora de comenzar a pensar en las variables a utilizar, y a definir cuales de ellas nos conviene que sean globales y cuales locales. En general, las variables que almacenan los resultados generales del programa, serán utilizadas por todos o casi todos los métodos que definamos. Por tanto, deben ser globales. (Y nos cuidaremos de declarar cualquier variable local con el mismo identificador). Tal es el caso del arreglo `f`, al que referimos anteriormente en nuestro análisis. Como vamos a hallar todas las soluciones, necesitamos también un contador `c` que las individualice. Es común que los contadores sean también globales: Se inicializan en el programa principal y se incrementan en el o los métodos apropiados. Por el momento, no parecen necesarias otras variables globales. En cuanto a las variables locales de cada procedimiento, las analizaremos al profundizar un poco mas sobre los mismos, en los puntos siguientes.

Colocación de las damas:
Comencemos con el método recursivo `Coloca`. El mismo debe recibir como parámetro el nivel (o sea la fila) donde debe situar la dama. Los requisitos que debe tenerse en cuenta al programar este procedimiento ya han sido analizados anteriormente:
Niveles 0 a n-1:

- Asignar una posición a la dama a colocar en la fila
- Comprobar si la posición es aceptable:
  SI: pasar a nivel siguiente
  NO: sin cambiar de nivel, correr la dama una posición a la derecha y volver a comprobar.

Nivel n: imprimir la solución obtenida.

De ahí, el siguiente modulo de programa:

```
static void Coloca(int niv, int[] f)
{
      if (niv>=f.Length) Imprimir(f);
      else  for(int i=0; i<f.Length; i++)
              {
                    f[niv]=i;
                    if (Posible(niv,f)) Coloca(niv+1,f);  //  {*}
              }
}
```

{*} Equivale a: `if (Posible(n,f)==true) Coloca(n+1,f);`

Es necesario intercalar un breve comentario:

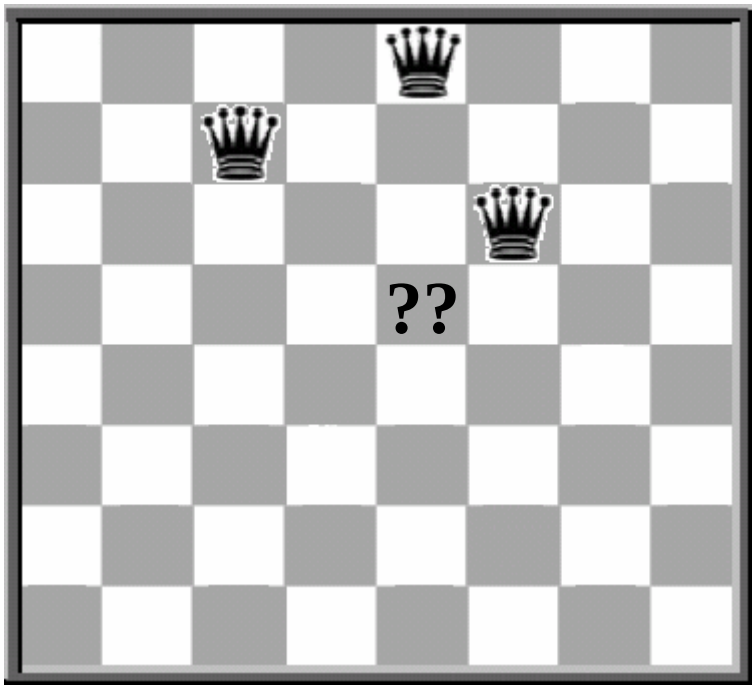
Como puede observarse, el paso de un nivel al siguiente esta condicionado por el control de la posición actual que realiza la función `Posible`. Puede suceder que a partir de cierta posición en una fila, no se pueda encontrar ningún lugar aceptable para la correspondiente dama. En ese caso, es imposible pasar al nivel siguiente y, después de analizar la última posición en la fila (i=7), el ciclo termina y el programa retorna al nivel anterior. Es decir que, a diferencia del ejemplo mostrado en el capitulo precedente, en este caso los retornos pueden producirse "a medio camino", en cuanto se detecta que la actual combinación no conduce a una nueva solución. Esta posibilidad de "tanteo" y en caso necesario "vuelta atrás" (back track) a intentar un nuevo camino, hace mucho más poderosa la herramienta que estamos estudiando. La forma de abordar un problema mediante la obtención de las combinaciones apropiadas, para detectar cuales de ellas son realmente soluciones, es una variante de una técnica, aplicable en muchas situaciones, que se conoce como "método de ensayo y error".

Volvamos a la etapa de trabajo en que estábamos:

Al escribir el método Coloca, nos hemos visto en la necesidad de ponerle nombre a los otros que el mismo utiliza. Ampliaremos ahora su análisis:

`Posible`

(Método `bool` para el control de la posición actual de la dama en el nivel `niv`).
Partimos de que las posiciones de las damas están almacenadas respectivamente en f[0],..., f[niv].Para que la posición en el nivel `niv` sea INCORRECTA, debe cumplirse alguna de las condiciones que se deduce del siguiente diagrama:



La posición 4 para f[3] no es posible porque:

1. Esta en la misma columna que la dama de fila 1: (f[3]=f[0]).
   En general: f[niv]=f[i] para alguna de las filas i < niv.

2. Esta en la misma "diagonal principal" que la de fila 1:
   (f[3]-3 = f[1]-1)
   En general: f[niv]-niv = f[i]-i para alguna de las filas i < niv.

3. Esta en la misma "diagonal se cundaria" que la de fila 3:
   (f[3]+3 = f[2]+2)
   En general, f[niv]+niv=f[i]+i para alguna de las filas i < niv .

Para llegar a estas expresiones hay que pensar un buen rato... Pero ahora ya las conocemos. Luego, podemos pasar directamente a programar la función:

```
static bool Posible(int niv, int[] f)
{
      for(int i=0; i<niv; i++)
            if ( f[i]==f[niv] ||
                  f[niv]-niv==f[i]-i ||
                  f[niv]+niv==f[i]+i ) return false;
      return true;
}
```

`Imprimir`

Para imprimir vamos a utilizan una forma de mostrar una solución, bastante sencilla y a la vez, suficientemente clara:

Imprimiremos en forma de tablero de ajedrez, situando puntos en las casillas vacías y las letras DD en las casillas ocupadas por las Damas, con un espacio de separación entre casillas.

Además, queremos imprimir el número de cada solución.

Ejemplo: Solución 23:
DD .. ..
.. .. DD .. ..
.. .. DD .. .. ..
.. .. DD ..
.. DD ..
.. .. etc.

El procedimiento no requiere el traspaso del parámetro “nivel”, pues solo se utiliza cuando se ha terminado de colocar todas las damas en el tablero pero sí que tablero queremos imprimir (arreglo).

```
static void Imprimir(int[] f)
{
    c++;                                        // <-- Cuento la solucion
    Console.WriteLine("Solucion "+c+":");
    Console.WriteLine();
    for(int i=0; i<f.Length; i++)
    {
          for(int j=0; j<f[i]; j++)             // <- Imprimo casillas vacias
                Console.Write(".. ");           //    hasta donde esta la dama
          Console.Write("DD ");                 // <- Imprimo la dama
          for(int j=f[i]+1; j<f.Length; j++)    // <- Imprimo las otras
                Console.Write(".. ");           //    casillas restantes
          Console.WriteLine();                  // <- Cambio de linea (fila)
    }
    Console.ReadLine();                         // <- Espera un ENTER...
}
```

Main

Bien pocas son las tareas que le quedan: declarar las variables globales, inicializar con 0 el contador c e iniciar el proceso recursivo en el nivel 0:

```
using System;
namespace Ocho_Damas
{
   class Principal
   {
         static int[] tab;
         static int c=0;

         static bool Posible(int niv, int[] f)
         {
               for(int i=0; i<niv; i++)
                     if ( f[i]==f[niv] ||
                           f[niv]-niv==f[i]-i ||
                           f[niv]+niv==f[i]+i ) return false;
               return true;
         }

         static void Imprimir(int[] f)
         {
               c++;
               Console.WriteLine("Solucion "+c+":");
               Console.WriteLine();

               for(int i=0; i<f.Length; i++)
               {
                     for(int j=0; j<f[i]; j++)
                           Console.Write(".. ");

                     Console.Write("DD ");
                     for(int j=f[i]+1; j<f.Length; j++)
                           Console.Write(".. ");

                     Console.WriteLine();
               }
               Console.ReadLine();
         }

         static void Coloca(int niv, int[] f)
         {
               if (niv>=f.Length) Imprimir(f);
               else  for(int i=0; i<f.Length; i++)
                       {
                             f[niv]=i;
                             if (Posible(niv,f)) Coloca(niv+1,f);
                       }
         }
```

```
        static void Main(string[] args)  // <- Programa Principal
        {
              Console.Write("Deme el tamaño del tablero (NxN): ");
              int n=int.Parse(Console.ReadLine());
              tab=new int[n];

              Coloca(0,tab);   // <- Note que a difencia del ejemplo
                               //    anterior este comienza en el nivel 0

              if (c==0) Console.WriteLine("No hay solucion.");
              else Console.WriteLine("Hay "+c+"soluciones.");

              Console.ReadLine(); //<- Pausa antes de terminar
        }
    }
}
```

¿Qué, metodología hemos seguido para llegar al programa completo?

1) De hecho, hemos centrado inicialmente nuestros esfuerzos en concebir correctamente un procedimiento recursivo adecuado para dar solución al problema; este procedimiento es el núcleo de todo el programa.
2) Hemos separado del mismo algunas tareas (`Posible`, `Imprimir`), para "descongestionar" el procedimiento fundamental del exceso de detalle, con lo que ganamos en claridad de concepción y de expresión del mismo.
3) El programa principal ha quedado prácticamente relegado al papel de iniciador del proceso.

Los ejemplos restantes siguen también, en buena medida, esta filosofía de trabajo.

II. Avances y retrocesos.

Un tablero lineal consta de N casillas (N>5), cada una de las cuales tiene un numero 2, 3 o 4 que le es propio. Estos números se distribuyen al azar entre las N casillas y constituyen, conjuntamente con N, los datos del problema. Una ficha situada sobre una casilla cualquiera, puede realizar indistintamente uno de estos tres movimientos (excepto que al hacerlo sobrepase alguno de los límites del tablero):

a) avanzar tantas casillas como indica el número propio de la casilla donde estaba anteriormente la ficha.
b) retroceder igual cantidad de casillas.
c) avanzar una sola casilla.

En cualquiera de estas variantes, se considera que la ficha "toca" solamente la casilla de partida y la de llegada, sin tener en cuenta las casillas intermedias, sobre las cuales se considera que ha "saltado".
Dada una lista con los números propios de cada casilla, halle todas las formas posibles en que la ficha recorra las N casillas, sin "tocar" dos veces ninguna de ellas.

Análisis:

1. El problema siempre tiene al menos una solución: como una de las posibilidades es avanzar una casilla, es obvio que comenzando por la primera, siempre podemos recorrerlas todas.

2. Pero no hay que comenzar siempre en la primera casilla: se pide todas las formas de recorrer las N casillas; y como otra de las posibilidades es la de hacer movimientos “hacia atrás”, nada nos obliga a comenzar por una casilla en especial. De modo que resulta necesario probar con cada una como posible punto de partida. Esta operación la detallaremos oportunamente (programa principal).

3. No se puede tocar dos veces la misma casilla. Por tanto, hay que marcar de alguna forma cada punto al que se llega. Como vamos a intentar muchos caminos, es necesario a su vez poder desmarcar con facilidad (y restaurando el valor original), para intentar otra vía. Entre otras formas posibles, este caso parece apropiado para marcar por el procedimiento de cambiarle el signo a la casilla: `t[p]*=(-1)`. Para desmarcar, basta repetir la operación anterior.

4. En cada “nivel” llegaremos a una nueva casilla. No obstante, no hace falta N niveles, por que en el primer movimiento debemos marcar en realidad dos casillas: la de comienzo y la de llegada. En los restantes, marcaremos cada vez, solo la de llegada. Por tanto: N-1 “niveles”, o, lo que es lo mismo, cada vez que lleguemos al nivel N es que hemos hallado una solución.

5. En cada nivel hay tres posibles movimientos. Resultara sencillo resolverlos con un `switch`, a partir de la posición (`p`) que teníamos en el nivel anterior.

6. Una vez calculada la nueva posición a la que debiera ir la ficha, hay que comprobarla: puede ser una posición fuera del tablero; o puede ser que la misma ya este marcada. De modo que inicialmente la nueva posición es “provisional” y la almacenaremos “momentáneamente” en una variable destinada a este fin (en nuestro programa, `k`). Si realmente se puede pasar a la nueva posición, entonces la guardamos en la variable adecuada, la misma que estamos utilizando para la posición actual. Esta variable será `p`, local. Después de transferir `p=k`, es el momento apropiado para marcar la nueva casilla.

7. Transitando así por los niveles, en algunas ocasiones llegaremos al nivel N con una nueva solución. Hay que pensar pues, en imprimirla. Pero observe que con las acciones analizadas hasta ahora, no nos es posible reproducir el orden en que fuimos recorriendo las casillas. Es claro que nos hace falta ir dejando una “memoria” de lo que hemos hecho en cada nivel, para poder imprimir el nuevo “camino” al llegar al final. Para ello utilizaremos un nuevo arreglo global, `a`. En cada nivel, después de obtener la nueva posición `p`, la almacenaremos en `a[niv]`.

Esta precaución nos resuelve, además, otro problema. Siempre hay que pensar “dinámicamente”: cuando en un nivel se haya agotado las tres posibilidades de movimiento, habrá que retornar al anterior. Pero... ¿en qué casilla estábamos en este nivel? El valor de p en aquel momento, se modifico antes de pasar al siguiente nivel. Por fortuna, basta tomar del arreglo a la “vieja” posición anterior.

8. Por ultimo, siempre hay que pensar que, inmediatamente después de un retorno (que es, de hecho, la única forma de volver a un nivel “inferior”) hay que restaurar la situación que teníamos en el mismo, como en aquel caso “marcamos” la casilla a la que habíamos llegado, ahora debemos “desmarcarla”. También guardamos en el arreglo a la posición a que habíamos llegado, pero no hace falta “anular” esta acción, porque esto se produce “automáticamente” al disminuir en 1 el valor del nivel. No obstante, en otros problemas es necesario “anular” una a una las VARIAS acciones realizadas en un nivel. Para ello, se recomienda proceder en orden INVERSO al empleado para las mismas antes del llamado recursivo. La no observancia de esta regla suele traerá grandes dificultades, ocasionando errores muy difíciles de detectar.

9. Las ideas anteriores han conformado un esbozo del método recursivo. Las tareas del mismo, hasta el momento son:
   Si el nivel es N entonces imprimir la solución (arreglo a)
   si no:
   * Partir de la posición a que habíamos llegado en el nivel anterior: `p = a[niv-1]`
   * Calcular (`switch`) la nueva posición. Resultado provisional en `k`.
   * Si el movimiento es posible:
     - Transferir a `p` valor de `k`. (De hecho, “mover” la ficha).
     - Marcar casilla de posición `p`.
     - asignar `a[niv]=p`
     - pasar a próximo nivel ----> (recursividad)

   al retornar:
   - Desmarcar casilla actual

10. Pensemos ahora en la impresión, que como en el caso anterior, programaremos en un procedimiento específico. Una forma bastante efectiva de realizarla es la de imprimir, para cada “paso”, la casilla de origen y la de destino. A partir de la información que se ha ido almacenando en el arreglo “a”, no debe ofrecernos ninguna dificultad.

11. ¿Qué nos queda para el programa principal?
    Por lo visto, solo falta:
    - Leer el tablero y guardarlo en t;
    - Inicializar en 0 el contador c;
    - En un ciclo, probar con todas las posibilidades de comienzo:
      * Guardar el comienzo en a[0]
      * Marcar la posición de comienzo
      * Iniciar el proceso recursivo
      * Al retorno: desmarcar.

A continuación, el programa completo:

```
using System;
namespace Avances_y_Retrocesos
{
	class Principal
	{
		static int c=0;  // Contador de soluciones
		static int[] a;  // Arreglo que se irá llenando con las soluciones
		static int[] t;  // Tablero con los saltos;
		static int n=0;  // Cantidad de casillas;

		static void Imprimir(int[] sol)
		{
			c++;		// Cuento la solucion

			Console.WriteLine("Camino "+c+":");
			Console.WriteLine("Casilla "+sol[0]+" --> "+sol[1]);

			for(int i=1; i<sol.Length-1; i++)
				Console.WriteLine("		"+sol[i]+" --> "+sol[i+1]);

			Console.ReadLine();	// Hago una pausa
		}

		public static int[] LeerLista(int cantidad)
		{
			int[] a=new int[cantidad];

			Console.WriteLine("Deme elementos de la lista.");

			for(int i=1; i<=cantidad; i++)
			{
				Console.Write(i+" - ");
				a[i-1]=int.Parse(Console.ReadLine());
			}

			return a;
		}
```

```
        static void Movimiento(int niv, int[] tab, int[] s)
        {
              int p=0;
              int k=0;
              if (niv >= n) Imprimir(s);
              else  for(int i=1; i<=3; i++)
                      {
                            p=s[niv-1];
                            k=p;

                            // Seleccionamos a donde sera el salto (*)
                            if (i==1) k+=tab[p];
                            if (i==2) k-=tab[p];
                            if (i==3) k++;

                            // ¿Puedo saltar? (estoy dentro del arreglo y no
                            // he estado antes en esa casilla)

                            if ( (k>=0) && (k<n) && (tab[k]>0) )
                            {
                                p=k;            // Nos movemos
                                tab[p]*=(-1);   // Marcamos
                                s[niv]=p;       // Guardamos el salto.

                                Movimiento(niv+1,tab,s); // Llamamos al sig.

                                tab[p]*=(-1);            // Desmarcamos
                            }
                      }
        }

        static void Main(string[] args)  // <-- programa principal
        {
              do
              {
                    Console.Write("Deme la cantidad de casillas (N>5): ");
                    n=int.Parse(Console.ReadLine());
              }while (n<=5);
              t=LeerLista(n);
              a = new int[n];
              for(int i=0; i<n; i++)
              {
                    a[0]=i;                  // Seleccionamos el comienzo
                    t[i]*=(-1);              // y lo marcamos
                    Movimiento(1,t,a);       // Desencadena la recursividad
                    t[i]*=(-1);              // Desmarcamos para usar otro
              }
              Console.WriteLine("Hay "+c+" caminos.");
              Console.ReadLine();          // Pausa antes de terminar;
        }
    }
}
```

(*) Nota:

Dejamos al lector que trate de implementar estos tres `if` de manera que queden más óptimos. Pista: Utilice un bloque Nested-IF o un `switch` como habíamos dicho anteriormente.

III. El problema "de las cajas chinas".

Sobre 10 cajas (o también "casillas") dispuestas linealmente una al lado de la otra, se colocan en cualquier orden cuatro láminas que tienen dibujado cierto símbolo (digamos, una letra "a") y otras tantas con otro símbolo (por ejemplo una letra "b"), dejando vacías las dos casillas centrales.

Dos letras que estén una al lado de la otra se pueden llevar (conservando el mismo orden que tenían) hacia las casillas vacías, originando una nueva posición de casillas desocupadas a las que, a su vez, puede llevarse otra pareja de letras. Y así sucesivamente.

Dada una configuración inicial cualquiera, se desea determinar la MENOR cantidad de "pasos" necesaria para que todas las "a" queden a la izquierda de cualquier "b", sin que importe en que posiciones finales queden las dos casillas consecutivas desocupadas.

Ejemplo: (Se indica con ** la pareja a mover en cada "paso")

```
Configuración inicial: abba--abab
                              **
                Paso 1: abbabaa--b
                          **
                Paso 2: a--abaabbb
                             **
                Paso 3: aaaab--bbb    (final)
```

Análisis:

1. No se puede descartar que sea posible crear un algoritmo no recursivo capaz de alcanzar en todos los casos la configuración final deseada. Eso NO BASTA. Se pide llegar a la misma en la MENOR cantidad de pasos. Esto solo se puede garantizar probando con TODAS las combinaciones. Por tanto, es necesario un proceso recursivo.

   Centremos entonces nuestro análisis, como en los casos anteriores, en el procedimiento recursivo, que en este caso se encargara de ir probando con todos los movimientos posibles de parejas de letras hacia los espacios vacíos.

   ¿Cuales son esos movimientos?
   Volvamos al ejemplo: a b b a - - a b a b
   Posiciones --> 0 1 2 3 4 5 6 7 8 9

   En estas condiciones, se puede "mover" hacia el "hueco" la pareja que "comienza" en posición 0 (ab), o en 1 (bb), o en 2 (ba), o en 6 (ab), o en 7 (ba) o en 8 (ab). Observe que NO ES POSIBLE MOVER la pareja de 3 (a-), ni la de 4 (--), ni la de 5 (-a), por que NO SON PAREJAS DE LETRAS. Por otra parte, obviamente, no hay pareja que comience en la décima posición (9). En general, si el "hueco" esta en las posiciones h y h+1, no se puede mover las parejas que comienzan en h-1, h y h+1.

Por tanto, se puede programar los movimientos posibles mediante un ciclo del tipo:

```
for(int i=0; i<9; i++)
    if ( (i<h-1) || (i>h+1) )
    {
        // mover pareja de i,i+1 hacia h,h+1
        // actualizar valor de h
        // pasar al siguiente nivel
    }
```

2. Surge una dificultad inesperada:
   Analicemos con la idea anterior algunos pasos "de tanteo" a partir de la configuración inicial que ya hemos visto:

```
Seria:      abba--abab
             **
Paso 1    --baababab
            **
Paso 2    ba--ababab
           **
Paso 3    --baababab
```

   No hace falta realizar muchos pasos para darnos cuenta que una posición dada puede repetirse, como se aprecia aquí después de los pasos 1 y 3. Cuando esto ocurre, no importa después de cuantos pasos, no solo hemos "perdido" todos los pasos intermedios, sino que, inevitablemente, caeremos en un "ciclo infinito", repitiendo una y otra vez toda la secuencia, sin poder alcanzar la solución. (De paso, sirva este ejemplo para mostrar la utilidad de realizar siempre algunas transformaciones "a mano", muy útiles no solo para detectar "problemas", sino incluso para familiarizar al programador con la situación planteada.)

3. De modo que hay que tomar alguna medida para evitarlo. Después de sopesar algunas posibilidades - por ejemplo, aquí no resultan útiles formas de "marcar" que hemos visto en ejemplos anteriores - nos inclinamos por GUARDAR cada una de las configuraciones que vamos logrando, para comprobar con ellas si la próxima configuración que obtengamos es o no una repetición. De ser así, habrá que rechazarla y probar con otra en el mismo nivel. Pero entonces, necesitaremos un arreglo (en este caso, de variables tipo `string`) para tal almacenamiento. Cualquier "nivel" tiene que tener acceso al mismo; por tanto, debe ser declarado global. Igual que en el caso anterior, el subíndice necesario para almacenar una configuración lo da el propio valor del nivel, en cada instante. Además, ya podemos ir pensando en escribir un método `bool EsNueva`, que se encargue de comparar cada nueva configuración con las obtenidas anteriormente.

4. Un aspecto nuevo: Como no sabemos cuantos son los pasos necesarios para alcanzar la solución, hay que realizar este control en cada "nivel". En nuestro caso, lo haremos al comenzar cada ejecución del procedimiento, aunque puede hacerse también en otros momentos.

5. Sea `min` una variable en la que almacenaremos la menor cantidad de pasos necesarios - hasta el momento - para llegar a la configuración final. Entonces:
   SI llegamos a una nueva solución en `k` pasos y `k < min`, tenemos una solución más óptima que la anterior. Luego:
   - Hay que almacenarla (para ello, hace falta OTRO arreglo);
   - Hay que actualizar el valor de `min`;
   - y debemos retornar al nivel anterior, para seguir intentando otras posibilidades.

   SI NO: No necesariamente pasaremos al nivel siguiente. Como estamos buscando la solución en menos pasos, no vale la pena seguir explorando una vía que sobrepase o aun iguale en cantidad de pasos a alguna que ya hayamos logrado.

   Uniendo esto con la parte final del punto 4, vemos la conveniencia de programar algo así:

```
if (EsNueva(niv)||(niv<min-1))
{
      //  "pasar al nivel siguiente"
}
```

6. Inicialmente, no hay ninguna solución en `min` pasos. ¿Con qué valor podemos inicializar `min`? Arbitrariamente, probamos con `min = 15`. Si no encontramos ninguna solución con menos de 15 pasos, probamos ahora con 20. Y así sucesivamente, hasta encontrar una inicialización satisfactoria. ¿Por que razón 15, 20, etc.? … ¡Por experiencia!
   Un proceso recursivo con 15 o 20 niveles es razonablemente rápido, si en cada nivel no hay una gran cantidad de cálculo. Con 40 niveles, ya se hace mucho más lento. Con 60, puede demorar HORAS...

7. Si se ha dado cuenta los pasos se irán almacenando en un arreglo que a la hora de crearlo no sabemos que tamaño tendrá e irá incrementándose según los niveles de la recursividad. Este es un problema que para resolverlo basta con crear el arreglo lo suficientemente grande (pero no tanto) e ir utilizando lo que necesitemos. Por ejemplo podemos decir que el arreglo es de largo 100 y utilizar solamente las 10 primeras casillas. En este caso nuestro arreglo tendrá de largo `min`. (claro no)

8. Resumen de ideas sobre métodos útiles:

   `static bool EsNueva(int niv)`
   - Comprueba si se ha repetido o no una configuración.

   `static bool EsSolucion(int niv)`
   - Se encarga de comprobar si todas las “a” están a la izquierda de cualquier “b”-> `true`.En caso contrario -> `false`

   `static void Intercambia(int niv)`
   - Será el procedimiento recursivo que debe:
   - Comprobar si en el nivel anterior se llegó a una solución más óptima y, en tal caso, almacenarla. NO: mover una pareja de letras al “hueco” y actualizar `h`.
   - Si corresponde, pasar al nivel siguiente.
   - Al retornar: - Restituir a `h` su “viejo” valor.

Como en ocasiones anteriores, definido el “núcleo” de la solución, es relativamente sencillo programar los restantes métodos y el programa principal.

```
using System;
namespace Cajas_Chinas
{
      class Principal
      {
            static int min=15;                  // <- Minimo de pasos
            static string[] a= new string[15];  // <- Arreglo de strings que
                                                //    contendrá los pasos
            static string[] s;                  // <- Solucion
            static int h=4;                     // <- Posicion del "hueco"

            static bool EsNueva(int niv)
            {
                  for(int i=0; i<niv; i++)
                        if (a[i]==a[niv]) return false;
                  return true;
            }

            static bool EsSolucion(int niv)
            {
                  int c=0;
                  int i=-1;
                  do
                  {
                        i++;
                        if (a[niv][i]=='a') c++;
                  }while (a[niv][i]!='b' && c!=4);
                  return (c==4)? true : false;
            }

            static void GuardaSolucion(int niv)
            {
                  min=niv;
                  s = new string[niv+1];
                  for(int i=0; i<=niv; i++) s[i]=a[i];
            }

            static void Imprimir()
            {
                  Console.WriteLine("Mejor solucion ("+min+" pasos): ");
                  Console.WriteLine();
                  Console.WriteLine("    "+s[0]);
                  Console.WriteLine();
                  for(int i=1; i<=min; i++)
                        Console.WriteLine(i+" - "+s[i]);
            }
```

```
            static void Intercambia(int niv)
            {
                  if (EsSolucion(niv-1)) GuardaSolucion(niv-1);
                  else for(int i=0; i<9; i++)
                        if ( i<h-1 || i>h+1 )
                        {
                              a[niv]="";  // -- Listo para construir la nueva
                              int hv=h;   // -- Salva posic. actual "hueco".

                              // Pasamos las cajas para el hueco
                              for(int j=0; j<a[niv-1].Length; j++)
                              {
                                    // Vea nota al final (*)
                                    if (j==h) a[niv]+=a[niv-1][i];
                                    else if (j==h+1) a[niv]+=a[niv-1][i+1];
                                    else if (j==i) a[niv]+="--";
                                    else if (j!=i+1) a[niv]+=a[niv-1][j];
                              }

                              h=i;        // -- Ahora el hueco esta en i

                              if ( EsNueva(niv) && niv+1<min )
                                    Intercambia(niv+1);

                              h=hv;       // -- Volvemos a poner el hueco
                                          //    donde estaba
                        }
            }

            static void Main(string[] args)   // <-- programa principal
            {
                  Console.Write("Configuracion inicial: ");
                  a[0]=Console.ReadLine();    // Vea nota (**)

                  Intercambia(1);

                  Imprimir();
                  Console.ReadLine();
            }
      }
}
```

(*) Esta estructura `if - else - if`... es conocida como Nested-IF o IFs anidados.
La cual es más eficiente que si usáramos los IFs separados.
(**)No es nuestro interés en este momento realizar la validación de los datos que se introduzcan por lo que se omite este paso, para así centrarnos en la nueva técnica que estamos introduciendo.