Teoría de la Recursividad
Capítulo 1: Relaciones entre elementos de un conjunto

La recursividad es la herramienta por excelencia para construir diferentes combinaciones de elementos de determinado tipo, con el fin de establecer aquellas que cumplen con ciertos requisitos prefijados, o que optimizan en algún sentido un aspecto de nuestro interés, como pudiera ser mayor productividad, o menor distancia, o costo mínimo, o similares.

Cuando no se encuentra una vía o algoritmo apropiado para llegar en forma más o menos "directa" al objetivo buscado, surge la necesidad de generar todas las variantes o "combinaciones" posibles en la situación dada.

En lo que sigue nos acercaremos a esta problemática, proponiendo situaciones sencillas en las que se hace necesario encontrar parejas, tríos, u otras agrupaciones de elementos de un conjunto que cumplan determinadas condiciones, o que optimicen un aspecto dado.
Como utilizaremos los elementos propios de cada situación propuesta, en cualquier orden - no necesariamente el de su lectura -, será muy común que haya que almacenar los mismos en un arreglo. Esto, con ser una característica muy general, no excluye la existencia de casos particulares en que no se hace necesario el empleo de arreglos.

Para "combinar" elementos de un conjunto dado, habitualmente basta plantear ciclos "`for`..." anidados convenientemente - en general, tantos ciclos como elementos se desea relacionar - para lograr el objetivo con relativa facilidad. A veces habrá que emplear ciertas técnicas o tener en cuenta determinadas consideraciones. Algunas de estas cuestiones "colaterales" se presentaran en los ejemplos que siguen.

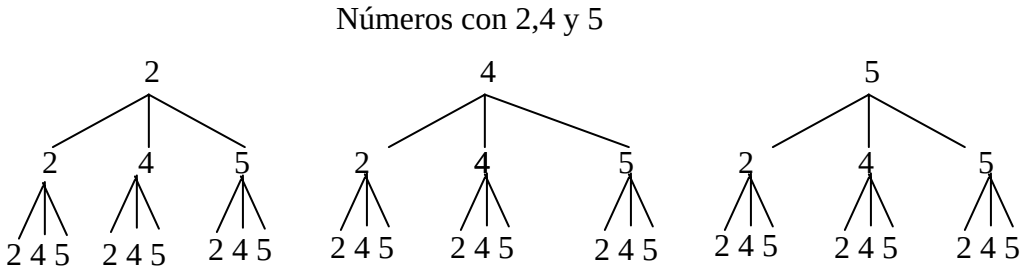
Ejemplos

1. Para concretar ideas, propongámonos obtener todos los números de tres cifras que sean divisibles por 17, y que se puedan construir a partir de tres dígitos dados (supongamos que los mismos son 2, 4, y 5). Entre los números que se busca, se admite la repetición de dígitos.

Si intentamos un enfoque "matemático" del problema, buscando las condiciones que debe reunir un número de tres cifras para que sea divisible por 17, chocaremos con la dificultad de que el criterio de divisibilidad por este número es bastante "complicado".

Es más sencillo optar por un criterio de "tanteo": construir todos los números posibles con los dígitos dados, y luego comprobar cu les cumplen la condición de divisibilidad.

La generación de los números posibles, la podemos esquematizar así:



[Un esquema de este tipo se denomina “árbol”. Bajando por las “ramificaciones” se obtiene todos los números]

Así, los números de tres cifras son:

222, 224, 225, 242, 244, 245, 252, 254, 255, 422, 424, 445, 442, 444, 445, 452, 454, 455, 522, 524, 545, 542, 544, 545, 552, 554, 555

Probando los divisibles por 17, se obtiene 255, 425, 442 y 544 que son los únicos, entre todos los posibles, que cumplen todas las condiciones exigidas.

En la computadora, guardando los dígitos en un arreglo “d”, y recordando que
abc = a*100 + b*10 + c, el problema se resuelve muy fácilmente:

```
for (int i=0; i<3; i++)
  for (int j=0; j<3; j++)
    for (int k=0; k<3; k++)
     {
           num= d[i]*100 + d[j]*10 + d[k];
           if (num % 17= =0) Console.WriteLine(num);
     }
```

Anidando ciclos `for`..., se obtiene de manera muy rápida y sencilla las combinaciones deseadas de los elementos del conjunto dado.

Pero puede hacer falta agregar otras consideraciones. Veamos los siguientes ejemplos:

2.- Supongamos que tenemos una lista de N números naturales positivos y deseamos saber cuantos divisores tiene cada número en la lista.

Como es natural, la lista estará almacenada en un arreglo (que llamaremos a). Debemos recorrer toda la lista tomando un elemento por vez para analizarlo (luego, hace falta un ciclo). Pero, una vez “fijado” el elemento a analizar en cierto momento, hay que realizar otro recorrido de la lista, para probar con cada elemento de la misma si es o no divisor del número que estamos analizando.

De ahí la necesidad de un segundo ciclo, “interior” al primero.

Resulta de ello una disposición de dos ciclos anidados, apropiada para “generar” todas las combinaciones posibles. Auxiliándonos de un segundo arreglo (c, de contadores) que previamente habremos inicializado con ceros, se llega al siguiente fragmento de programa:

```
for (int i=0; i<a.Length; i++)
    for (int j=0; i<a.Length; j++)
        if (a[i] % a[j]= =0) c[i]++;
```

Cuyo efecto es contar cuantos divisores tiene, en la propia lista, cada uno de sus elementos.

3.- Supongamos ahora que tenemos otra lista de N números no repetidos y que nos interesa obtener todas las sumas posibles de tres sumandos diferentes cada una. Pero, si ya hemos considerado la suma x + y + z, queremos evitar considerar las equivalentes z + x + y, o y + x + z. Es decir, en este caso, el ORDEN de los elementos no determina que dos tríos sean diferentes.

Para evitar repeticiones, basta ir relacionando cada elemento de la lista solamente CON LOS QUE LE SIGUEN. (Si nos imaginamos los números de la lista dispuestos horizontalmente, de izquierda a derecha, diremos también: basta ir relacionando cada elemento de la lista solamente con los que están A SU DERECHA. Usaremos indistintamente una u otra expresión en este texto).

En principio, la formulación de los ciclos parece ser

```
for (int i=0; i<a.Length; i++)
  for (int j=i+1; j<a.Length; j++)
    for (int k=j+1; k<a.Length; k++)
    {
         suma=a[i]+a[j]+a[k];
         Console.WriteLine(a[i]+"+"+a[j]+"+"+a[k]+"="+suma);
    }
```

Observamos que cada ciclo interior comienza a partir de la posición de la lista SIGUIENTE a la del sumando seleccionado en el ciclo previo, y tomar sucesivamente todos los elementos de la lista desde dicha posición hasta el final.

Aunque esto funciona bien en C# (que controla antes de la ejecución de un ciclo si este debe ejecutarse o no), podemos ser más exactos al programar. En efecto, el primer sumando debe tomarse a lo sumo hasta la posición n-2, pues los dos restantes deben tomarse de posiciones posteriores a aquella. Y por similar razón, el segundo sumando solo debería tomarse, como máximo, hasta la posición n-1.

Luego, una formulación más adecuada es la siguiente:

```
for (int i=0; i<a.Length-2; i++)
  for (int j=i+1; j<a.Length-1; j++)
    for (int k=j+1; k<a.Length; k++)
    {
         suma=a[i]+a[j]+a[k];
         Console.WriteLine(a[i]+"+"+a[j]+"+"+a[k]+"="+suma);
    }
```

Destacamos que los ciclos anidados definidos como se muestra en la forma precedente, garantizan la obtención de todas las combinaciones posibles, de tres elementos distintos por vez, sin repetir ninguna, en los casos en que se desee EVITAR incluir los MISMOS elementos en diferente orden.

4.- Modifiquemos en parte el enunciado anterior, para permitir que los tres sumandos no sean necesariamente elementos distintos. Es decir, admitimos ahora expresiones del tipo x+x+z, x+x+x o x+y+y.

No obstante, mantenemos la restricción de no considerar una expresión que contenga los mismos sumandos (aunque en distinto orden) que otra ya analizada previamente. O sea, si ya hemos analizado x+x+z no queremos imprimir también x+z+x o z+x+x.

Para este caso, nos bastará con introducir una mínima modificación a la disposición anterior:

```
for (int i=0; i<a.Length-2; i++)
   for (int j=i; j<a.Length-1; j++)
      for (int k=j; k<a.Length; k++)
      {
          suma=a[i]+a[j]+a[k];
          Console.WriteLine(a[i]+"+"+a[j]+"+"+a[k]+"="+suma);
      }
```

5. En ocasiones, el hecho de que el conjunto esta o no ordenado determina una u otra forma de disponer los ciclos.

Veamos el siguiente ejemplo ilustrativo en ese sentido:

Se tiene una lista de N números positivos. Se desea imprimir todos los “tríos pitagóricos” que se pueda obtener con elementos de la lista. (“trío pitagórico” es cualquier conjunto de tres números positivos - a veces se agrega la condición de que sean naturales - tales que el cuadrado de uno de ellos sea igual a la suma de los cuadrados de los otros dos. Ej.: 3, 4,5).

Caso a: La lista no está ordenada.

En esta situación, hay que analizar todos los tríos posibles, y de ahí surge la disposición.

```
for (int i=0; i<a.Length; i++)
  for (int j=0; j<a.Length; j++)
    for (int k=0; k<a.Length; k++)
      if ( (a[i]*a[i]) = = (a[j]*a[j])+(a[k]*a[k]) )
      { ... }
```

En este caso, podemos comprobar que obtenemos todos los tríos pitagóricos pero "duplicados": si en la lista estaban los números 6, 8, 10, nuestro programa imprimiría 10, 8, 6 y también 10, 6, 8.

Generalmente, una duplicación de este tipo no es deseable, y para evitarla, habría que agregar alguna otra condición, como por ejemplo:

```
if  (( a[j]>a[k] ) && ( (a[i]*a[i]) = = (a[j]*a[j])+(a[k]*a[k]) )) { ... }
```

Caso b: La lista está ordenada en forma descendente (o la ordenamos previamente en esa forma).

En tal caso, basta relacionar cada elemento con los que están a su derecha, y los ciclos serían:

```
for (int i=0; i<a.Length-2; i++)
    for (int j=i; j<a.Length-1; j++)
        for (int k=j; k<a.Length; k++)
            if ( (a[i]*a[i]) == (a[j]*a[j])+(a[k]*a[k]) ) { ... }
```

Con lo que obtenemos todos los tríos posibles; y sin repeticiones.

6. De hecho, estamos incursionando en una parte de las Matemáticas que se conoce como Teoría Combinatoria, que estudia los diferentes agrupamientos que se puede obtener tomando K elementos por vez de un conjunto de N elementos (N mayor o igual que K). En algunos casos, no se admite la repetición de elementos pero en otros sí; en ciertos casos interesa el orden de los elementos, en otros no.

Puesto que el tratamiento más detallado de esta cuestión queda completamente fuera de los objetivos de este trabajo, nos permitimos sugerir al lector que estudie con detenimiento las características de cada uno de los tres modelos presentados, para lograr captar en cada caso la correspondencia entre las restricciones planteadas y la apropiada formulación de los ciclos.

7. No siempre los elementos a relacionar surgen de una lista de datos, ni necesariamente tienen que estar almacenados en un arreglo. Veamos un ejemplo:

Se tiene tres números reales cualesquiera, x, y, z; (datos). Con los dos primeros, se desea realizar una cualquiera de las siguientes operaciones: adición, sustracción o multiplicación. Con el resultado obtenido y el tercer número, se quiere efectuar también una de las operaciones mencionadas (ambas operaciones pueden repetirse, o no). Se desea encontrar una combinación de operaciones mediante las cuales se obtenga, en valor absoluto, el mayor resultado final posible.

Queda claro que no se trata aquí de obtener combinaciones de números, sino una secesión “óptima” de operaciones aritméticas. Por otra parte, el enunciado no excluye la repetición de operaciones (Ej.: +,+), y admite las mismas operaciones en distinto orden (Ej.: +,* y *,+; ya que (x+y)*z no tiene por que conducir al mismo resultado que (x*y)+z).

Por último, si bien de hecho hay una “lista” (+,-,*), no hace falta esta vez almacenarla en un arreglo: Basta con “asociar” cada operación a uno de los números 1, 2, 3, y ya estamos en condiciones de obtener todas las variantes posibles mediante dos ciclos anidados.

A continuación presentamos el programa completo, en el que podrá apreciarse la utilización de la misma técnica básica del primer ejemplo, aunque con elementos complementarios de programación que no fue necesario utilizar en aquella ocasión:

```
using System;
namespace Ejemplo_7
{
   class Ejemplo_7
   {
      static double x,y,z,r,rf,max;
      static char p,prim,s,seg;      //  <---  para  almacenar símbolos
                                     //        de 1ra. y 2da. operación
      static void Main()
      {
            Console.WriteLine("Deme los 3 numeros: ");
            x=double.Parse(Console.ReadLine());
            y=double.Parse(Console.ReadLine());
            z=double.Parse(Console.ReadLine());
            max=0;

            for(int i=1; i<=3; i++)
            {
                  if (i==1) { r=x+y; prim='+'; }
                  if (i==2) { r=x-y; prim='-'; }
                  if (i==3) { r=x*y; prim='*'; }
                  for(int j=1; j<=3; j++)
                  {
                        if (j==1) { rf=r+z; seg='+'; }
                        if (j==2) { rf=r-z; seg='-'; }
                        if (j==3) { rf=r*z; seg='*'; }

                        if (rf>max) { max=rf; p=prim; s=seg; }
                  }
            }
            Console.WriteLine("Resultado maximo: "+x+p+y+s+z+ "=" + max);
            Console.ReadLine();
            }
      }
}
```

8. En ocasiones, hace falta recurrir a determinadas técnicas para lograr exactamente el resultado deseado:

Supongamos que tenemos una lista de N números cualesquiera. Los números pueden aparecer una o más veces en la lista, en cualquier orden. Nos interesa saber cuantas veces aparece cada número en la lista.
Sea la lista de 6 elementos 4, 9, 4, 5, 5, 4
Nos gustaría luego de analizarla, lograr la siguiente impresión:

Apariciones: 4 - 3 veces
9 - 1 vez
5 - 2 veces

Es obvio que tenemos que comparar los elementos de la lista, e ir contando las repeticiones. En este caso, bastaría comparar cada número con los que aparecen a su derecha. Pero también es evidente que, por ejemplo, una vez analizado el primer 4, contando todas sus apariciones en la lista, no sólo no es necesario sino que es incorrecto analizar los otros 4 que aparecen en la misma.

Expondremos brevemente distintas técnicas posibles para evitar analizar dos veces el mismo número:

a) Si no interesa conservar la lista original, sino obtener la lista “depurada” de repeticiones, podemos realizar las comparaciones a medida que vamos leyendo la lista. Necesitaremos el arreglo <a> para la lista “depurada” y un arreglo auxiliar (cant), cuyos elementos se inicializan con 1 a medida que van siendo necesarios, para llevar la cuenta de la cantidad de veces que aparece cada número. Los números repetidos simplemente no se incluyen en la lista, pero se cuentan.
El correspondiente fragmento de programa podría ser así:

```
// n: total de elementos de la lista original
// m: cantidad de elementos en la lista “depurada”
a[0] = int.Parse(Console.ReadLine());
cant[0]=1;
m=0;
int j=0;
for (int i=1; i<=n; i++)
{
     int x = int.Parse(Console.ReadLine());
     j=-1;
     do
     {         // Búsqueda de x en la lista "depurada"
           j++;
     }while ((j<m) && (x!=a[j]));
     if (x == a[j]) cant[j]++;
     else
     {
           m++;              //    inclusión de x
           a[m]=x;           //    en la lista
           cant[m]=1;
     }
}
```

En este fragmento, se lee el primer elemento de la lista sobre a[0] y se inicializa con 1 las variables cant[0] y m.

A partir de este instante, los restantes elementos de la lista se leen sobre la variable x y posteriormente se buscan entre los elementos ya almacenados en el arreglo (ciclo "`do-while`..."). Este ciclo termina cuando se de una de dos condiciones:

- O bien el número ya estaba en el arreglo, es decir, x=a[j] para algún valor de j; luego, se trata de un número repetido y en ese caso solo se incrementa el contador correspondiente.

- O bien, j recorre los m elementos del arreglo sin encontrar a x. En este caso, se incrementa m y se guarda x en el arreglo (hay un nuevo elemento en la lista); y como es natural, se indica que la cantidad de apariciones del "nuevo" es 1 (hasta ese momento).

Observamos que en ocasiones, es más apropiado utilizar un ciclo `do-while` (también podría ser `while`...) que un ciclo `for`... pero la idea básica del "anidamiento" es la misma.

b) Si nos interesa mantener la lista original, entonces podemos

b.1) En un nuevo arreglo auxiliar, ir almacenando cada número que analizamos. Antes de analizar un número de la lista, habrá que buscarlo en el arreglo auxiliar. Si no está, es que se presenta por primera vez, y procedemos en la forma indicada, realizando luego todas las comparaciones del mismo con los que le siguen en la lista original. Si el número esta en la lista auxiliar, no lo analizamos nuevamente.

b.2) "Marcar" las repeticiones que nos vamos encontrando, y no analizar los elementos "marcados".

Para "marcar" un elemento, a veces basta con hacerle una pequeña transformación (cambiarle el signo, adicionarle una constante conveniente, etc.), que nos permita diferenciar fácilmente este elemento de los restantes (no "marcados"), y, a la vez, en el instante oportuno, recuperar el número original realizando la transformación inversa a la de la marca.

En los casos en que esto no sea fácilmente realizable, es preferible utilizar otro arreglo (usualmente de tipo `bool`), en el que un valor convencional definido por el programador indica si el elemento debe analizarse o no.

En los capítulos 4 y 5 se muestra programas que requieren el empleo de algunas de estas técnicas.

Ejercicios Propuestos:

1- Dados N números enteros diferentes, y otro número natural K, con 0 < K < N imprima el único numero de la lista que tiene exactamente K menores que ,l en la misma. Encuentre la solución sin ordenar la lista.

2- Imprima en orden ascendente todos los números de 4 cifras que es posible construir empleando los dígitos 4, 9 y 3. Se admite la repetición de dígitos.

3- Se tiene una lista de N números naturales, positivos y no repetidos. Imprima la fracción más próxima a 1, pero diferente de 1, que se puede construir con elementos de la lista.

4- Dada una lista de nombres masculinos de una sola palabra (Ej.: Carlos, Alberto, Roberto) obtener todos los nombres compuestos diferentes que sea posible (cada nombre compuesto se formar con 2 nombres simples), con la condición de que ambos nombres no comiencen con la misma letra.

5- Dada una lista de N números, en la que uno o más pueden estar repetidos, encontrar las posiciones más alejadas entre sí que contienen un mismo número.

6- Dada una oración, imprimir el "tramo" más largo de la misma en que no se repite ningún carácter.

7- Dada una lista de 8 números, subdivídala en dos sub-listas de 4 elementos cada una, de modo que las sumas de ambas se diferencien entre sí lo menos posible.

8- Dados 4 dígitos diferentes y mayores que cero, imprima todos los números de 4 cifras que es posible formar con ellos, con la condición de que ninguno de los dígitos puede aparecer más de dos veces en un mismo numero.

9- Halle todas las soluciones de la ecuación: ((A op1 B) op2 C) op3 D = E en la que A, B, C, D, E son números reales que se lee como datos y op1, op2, op3 simbolizan en cada caso cualquiera de las operaciones aritméticas fundamentales (+,-,*, /). Se admite la aparición repetida de operaciones.

10- N ciudades se comunican entre sí por carretera. Sobre una misma carretera pueden estar varias de las N ciudades, de modo que la conexión entre dos ciudades puede ser directa, o a través de ciudades intermedias. Puede haber, además, más de una vía de comunicación entre dos ciudades. Llamaremos "tramo" a los segmentos de carretera que unen directamente dos ciudades (sin que haya ciudades intermedias).
Dados los datos N: cantidad de ciudades, T: cantidad de tramos y una lista de los T tramos, definido cada uno de ellos por un trío de la forma

No. de ciudad, distancia en Km., No. de ciudad

se desea encontrar el camino m s corto (de uno, dos o tres tramos) entre dos ciudades dadas A y B.

Impresión: Supongamos que se pide el camino entre las ciudades 4 y 9.
Debe imprimir, según sea el caso, una de estos 4 tipos de variantes:

- No hay camino de tres tramos o menos entre 4 y 9
- Camino mas corto: 4-9
- Camino mas corto: 4-11-9
- Camino mas corto: 4-2-10-9

11- Dados N números naturales (N>5), imprima sin repeticiones todos los tríos posibles tales que la suma de dos de esos números, divididos por un tercero, también de la lista, den un resultado mayor que 1 y menor que 2. Cada número puede aparecer a lo sumo una vez en un trío. Los tríos deben imprimirse en el orden 1er.sumando, 2do sumando, divisor. A los efectos de este problema, a, b, c y b, a, c se consideran EL MISMO trío (porque a + b es "lo mismo" que b + a), pero a, b, c y a, c, b son TRÍOS DIFERENTES.

12- Dada una oración, imprima en pantalla y separando las letras por un espacio, la palabra más larga. (Si hubiera dos o más de longitud máxima, tome la situada más a la derecha en la oración).

A continuación, seleccione de las restantes palabras, aquellas que podrían "cruzarse" con la más larga seleccionada, si se imprimieran en forma vertical. En esa selección, tienen prioridad las palabras más largas. Dos palabras "verticales" no se pueden cruzar sobre letras consecutivas de la palabra horizontal.

Imprima como lo muestra el ejemplo:

Oración dada: este es un ejemplo sencillo
Impresión: s e n c i l l o
2 3 1

1 – ejemplo
2 – este
3 – un
No se pudo cruzar: es

Observe que "es" no se puede "cruzar" con la "e" de sencillo, porque al lado cruza "este", que tiene prioridad por ser más larga.