Reto 2

Andrés Herrera Poyatos

26/10/2014

Contents

1 Introducción al problema de las cifras

Dado un conjunto de 6 enteros sacados aleatoriamente con remplaamiento del conjunto:

$$A = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 25, 50, 75, 100\}$$

se plantea conseguir otro entero aleatorio de 3 cifras usando para ello solo las operaciones de suma, resta, producto y división entera, teniendo en cuenta que solo se puede usar cada número (de los 6) como mucho una vez y que es posible no utilizar los 6 para conseguir el número de 3 cifras. Solo se permite utilizar la resta si el resultado es mayor que 0. Análogamente, solo se permite el uso de la división entera en el caso de ser esta exacta (su resto es 0).

2 Definiciones y resultados previos

2.1 Relaciones de Equivalencia

Las relaciones de equivalencia son un concepto matemático definido sobre un conjunto dado cualquiera. Como tantos otros conceptos matemáticos, está basado en una idea intuitiva, la representación de relaciones del tipo: ciudades en una misma región, alumnos de la misma clase, instrucciones dentro del mismo bloque de código, enteros con el mismo valor de módulo P , etc.

Definición 1. Sea A un conjunto. Una **relación binaria** $\sim$ definida sobre A es una regla que nos indican si dados dos elementos a y b pertenecientes a A están o no relacionados. En caso de estarlo se denota $a \sim b$. Si por el contrario a y b no están relacionados se denota $a \not\sim b$.

Definición 2. Sea A un conjunto. Una **relación de equivalencia** definida en A es una relación binaria $\sim$ de A que verifica las siguientes propiedades:

1. Reflexiva: $a \sim a \forall a \in A$
2. Simétrica: $a \sim b \Leftrightarrow b \sim a \ \forall a, b \in A$
3. Transitiva: $a \sim b \ y \ b \sim c \Rightarrow a \sim c \ \forall a, b, c \in A$

Definición 3. Sea A un conjunto, $a, b \in A$ y $\sim$ una relación de equivalencia sobre A. Para cada $a \in A$ se llama **clase de equivalencia de a**, y se denota por $[a]$, al conjunto:

$$[a] = \{b \in A : a \sim b\}$$

Teorema 1. Sea A un conjunto, $a, b \in A$ y $\sim$ una relación de equivalencia sobre A. Entonces, la intersección de dos clases de equivalencia es no vacía si y sólo si ambas clases coinciden:

$$[a] \bigcap [b] \neq \emptyset \Leftrightarrow [a] = [b]$$

Este último teorema permite escribir el conjunto A como unión disjunta de las clases de equivalencia del mismo. Es decir, el **conjunto cociente** (ver siguiente definición) es una partición de A.

Definición 4. Sea A un conjunto y $\sim$ una relación de equivalencia sobre A. Se llama **conjunto cociente** (y se denota $A/\sim$) al conjunto formado por las clases de equivalencia de A:

$$A/\sim = \{[a] : a \in A\}$$

2.2 El Problema de las Cifras

Definición 1. Sea C un conjunto finito de enteros positivos y x otro entero positivo. Una **solución** al **Problema de las Cifras asociado a C y x** es un par formado por una posible operación de elementos de C (combinación de elementos de C, sin repetición, y de las operaciones elementales –suma, producto, resta, división y posibles paréntesis– con las restricciones dadas en la introducción), y el resultado de la operación. La **mejor solución** al **Problema de las Cifras asociado a C y x** es la solución que miniminiza la distancia de su resultado a x. A x se lo denomina **valor objetivo** mientras que a los elementos de C se los denomina **elementos iniciales** o **soluciones iniciales**.

Por ejemplo, si $C = \{2, 6, 7, 9, 50, 75\}$ y *x=100*, una solución a su problema asociado es el par:

$$((6 * 7 - 50/2) * 9, 153)$$

La mejor solución al problema anterior responde a:

$$(2 * 50, 100)$$

Definición 2. Se considera el Problema de las Cifras asociado a C y x. Se dice que dos soluciones al problema, s y *s'*, son **semejantes** cuando están compuestas por los mismos elementos iniciales. Se denota $s \approx s'$. Por tanto, $\approx$ es una relación binaria en S, donde S es el conjunto de todas las soluciones al Problema de las Cifras asociado a C y x.

Proposición 1. $\approx$ es una relación de equivalencia sobre S.
Demostración
Basta ver que $\approx$ es reflexiva, simétrica y transitiva. En cualquier caso es dejarse llevar por la definición.

■

Consecuentemente, podemos dividir el conjunto S en clases de equivalencia a partir de $\approx$. El resultado es claro, una clase de equivalencia está formada por aquellas soluciones generadas por los mismos elementos iniciales. Basta "contar" cuantas combinaciones de los elementos iniciales existen para saber el número de clases de equivalencia de S.

Proposición 2. La relación de equivalencia $\approx$ divide al conjunto S en 2^n clases de equivalencia, donde n es el número de elementos iniciales o cardinal de C.
Demostración
Basta considerar el número de cadenas formadas por ceros y unos de longitud n. La razón es bien sencilla, si hay un uno en la posición i-ésima está el elemento inicial i-ésimo en la solución. Si hay un 0 no está. Luego cada cadena de unos y ceros caracteriza a una clase en particular y toda clase es caracterizada unívocamente de esta forma. Es claro que el número de cadenas formadas por ceros y unos de longitud n es 2^n (dos posibilidades por componente).

■

Nótese que estamos considerando la clase de aquellas soluciones compuestas por ningún elemento inicial. Es claro que esta clase es vacía, la obviaremos a partir de ahora. Contamos con $2^n - 1$ clases útiles, de las cuales n de ellas son aquellas cuyas soluciones solo han sido formadas por un elemento inicial, es decir, cada una contiene una única solución del tipo (c, c) con $c \in C$. Estas son las denominadas **clases iniciales** a partir de las cuales construiremos las demás.

En la demostracción se le asigna a cada clase una cadena de ceros y unos de longitud n. Esta cadena es característica de la clase. Podemos considerarla como un número en binario. Al número resultado de pasar la cadena de binario a decimal se lo denomina **clave** de la clase. Denotamos **clase**$[m]$ a la clase de clave m.

Por ejemplo, para $C = \{2, 6, 7, 9, 50, 75\}$, hay 64 clases cuyas claves son: $\{0, 1, 2, 8, ..., 32, 3, ..., 7, ..., 63\}$. Además:

- clase[1] = {(75,75)} (1 = 000001)
- clase[2] = {(50,50)} (2 = 000010)
- clase[4] = {(9,9)} (4 = 000100)
- clase[8] = {(7,7)} (8 = 001000)
- clase[16] = {(6,6)} (16 = 010000)
- clase[32] = {(2,2)} (32 = 100000)
- clase[3] = {Soluciones generadas por los elementos iniciales 50 y 75} (3 = 000011)
- clase[7] = {Soluciones generadas por los elementos iniciales 9, 50 y 75} (7 = 000111)
- clase[63] = {Soluciones generadas por todos los elementos iniciales} (63 = 111111)

Definición 3. Una clase de clave m es **menor** que otra clase de clave k si sus soluciones están generadas por menos elementos iniciales. Equivalentemente, una clase es menor que otra si el número de unos de su clave es menor. Se denota clase[m] < clase[k]. Es claro que ser menor es un relación de orden sobre el conjunto de las clases.

Definición 4. Una clase de clave m es una **subclase** de otra clase de clave k si clase[m] < clase[k] y los elementos iniciales que generan la clase[m] también generan (junto con otros) la clase[k], es decir, m & $k = m$, donde & es el operador lógico *AND*. Se denota $clase[m] \prec clase[k]$.

Caracterización 1. Sean $m, k \in 1, ..., 2^n - 1$. Los siguientes enunciados son equivalentes:
a) La clase de clave m es una **subclase** de la clase de clave k.
b) m & $k = m$ y k-$m > 0$.
Demostración
Veamos que a) implica b). Por definición, m & $k = m$. Como clase[m] está generada por más elementos iniciales que clase[k], k debe tener algún uno más a parte de los que comparte con m. Por tanto, k-$m > 0$. Para el recíproco basta ver que clase[m] < clase[k]. Efectivamente, como m & $k = m$, k tiene al menos los mismos unos en binario que m. Además, k-$m > 0$ lo que obliga a que k contiene algún uno más que m.

■

Definición 5. Dos soluciones s y s' se denominan **disjuntas** si no comparten ningún elemento inicial en sus combinaciones correspondientes. Análogamente, dos clases de clave k y m son disjuntas si k & $m = 0$, es decir, las soluciones de una clase son disjuntas con las de la otra.

Proposición 3. Sea una clase de clave k y una subclase de la misma de clave m. Entonces, la clase de clave k-m es una subclase de clase[k]. Además, las clases m y k-m son disjuntas.
Demostración
Utilizando la caracterización anterior, m & $k = m$. Entonces, *(k-m)* & m=0 y *(k-m)* $\oplus$ $m = k$. Utilizando la operación *AND* en la segunda igualdad: *(k-m)* & k = *(k-m)* & (*(k-m)* $\oplus$ m) = (*(k-m)*&*(k-m)*)$\oplus$(*(k-m)*&m) = *(k-m)*$\oplus$ 0 = *(k-m)* Por tanto, *(k-m)* & k = *(k-m)*. Además, k-(k-m) = m > 0, verificando la clase de clave k-m la caracterización de subclase, luego clase[k-m] $\prec$ clase[k]. De *(k-m)* & m=0 se tiene que las clases de claves k y m son disjuntas.

■

Definición 6. Dadas s = *(E,r)* y s' = *(E',r')* con $s \in clase[k]$ y $s' \in clase[m]$, clases disjuntas, donde E y E' son sus correspondientes expresiones y r y r' los resultados de las mismas. Entonces, definimos las soluciones ***s+s'*** = *(E+E',r+r')*, ***s*s'*** = *(E+E',r+r')*, ***s-s'*** = *(E-E',r-r')* (suponiendo $r > r'$) y ***s/s'*** = *(E / E',r / r')* (suponiendo que r' divide a r, es decir, r % r' = 0). Estas nuevas soluciones pertenecen a la clase de clave k+m. Llamamos soluciones generadas por las clases de claves k y m a aquellas soluciones que se pueden obtener a partir de aplicar una operación anterior a una solución de la clase de clave k y otra de la clase de clave m.

Por ejemplo, para $C = \{2, 6, 7, 9, 50, 75\}$ y s=(6+9,15), s'=(2*75,150) disjuntas se tiene que:

- s+s' = ((6+9)+(2*75), 165)

- s*s' = ((6+9)*(2*75), 2250)
- s'-s = ((2*75)-(6+9), 135)
- s'/s = ((2*75)/(6+9), 10)

Nótese que $s \in clase[20]$ y $s' \in clase[33]$ mientras que las 4 soluciones calculadas pertenecen a clase[20+33]=clase[53].

Podemos utilizar el concepto anterior para generar una clase a partir de sus subclases.

Proposición 4. Las soluciones de una clase de clave k coinciden con el conjunto de todas las soluciones generadas por cada subclase de clave i y la subclase de clave k-i, denotado B, (siendo ambas subclases disjuntas por la **proposición 3**).
Demostración.
La demostración se realiza por doble inclusión.

- Veamos que la clase de clave k está contenida en B: Si una solución s pertenece a la clase[k], por el carácter binario de los operadores +,*,-,/ su expresión E debe responder a dos expresiones *E'* y *E"* sobre las que se aplica uno de los operadores anteriores. Por tanto, *s = (E,r) = (E',r')operación(E'',r")* donde *r'* y *r''* son los resultados de evaluar sus correspondientes expresiones. Es claro que *(E',r')* y *(E'',r")* pertenecen a dos subclases disjuntas (de claves i y j) de la clase[k]. Se tiene i & $j = 0$ y i *OR* $j = k$. Luego $i+j = k$ y $j = k$-i. Por tanto, las subclases tienen claves i y k-i respectivamente como se quería.

- Veamos que B está contenido en la clase de clave k. Sea una subclase de clase[k] con clave i. Entonces las soluciones generadas por las subclases de claves i y k-i pertenecen a la clase de clave $i+(k-i) = k$ como se vio en la definición 6.

■

Proposición 5. Dada una clase de clave k presentando t elementos iniciales en sus soluciones, existen 2^t subclases de la misma (contando a la clase vacía y a sí misma).
Demostración
Basta ver una subclase (de forma análoga a la proposición 2) como una cadena de 0 y 1 de longitud t donde un 1 en la posición i-ésima indica que el elemento inicial correspondiente a dicha posición es usado en la subclase y un 0 indica que no. Hay 2^t cadenas posibles. Como una cadena caracteriza de forma unívoca a una subclase, se tiene que hay 2^t subclases posibles con la clase vacía (todos 0) y la clase[i] (todos 1).

Definición 7. Se dice que dos soluciones s y s', son **equivalentes** cuando están compuestas por los mismos elementos iniciales. Se denota $s \sim s'$.

3 Algoritmo

La idea para resolver el problema es construir una clase a partir de sus subclases utilizando la proposición 4. De esta forma no se recalcula cada solución, ya están realizadas las operaciones de las correspondientes subclases. Por ejemplo, para $C = \{2, 6, 7, 9, 50, 75\}$:

- clase[7] = {clase[1]+*-/clase[6], clase[2]+*-/clase[5], clase[4]+*-/clase[3]}

Cada clase se representa como un conjunto de soluciones. Por lo tanto, el algoritmo en primer lugar declara un vector con 2^n clases (**proposición 2**), donde n es el cardinal de C (6 en nuestro caso). Posteriormente, se ordenan las clases por la relación de orden **menor que** dada en la **definición 3**. De esta forma, el algoritmo recorre el vector de clases creando la clase actual a partir de las anteriores que sean subclases de la actual. La validez de este procedimiento la proporciona la **proposición 4** y el hecho de que las clases anteriores en

el vector (ya creadas) son todas las menores que la actual luego entre ellas se encuentran todas sus subclases. La **proposición 5** nos dice que hay 2^t subclases para cada clase generada por t elementos iniciales. Si aplicamos la **proposición 4** para su generación, se calcularan las soluciones para las $2^{t-1}-1$ posibles parejas de subclases disjuntas que generen soluciones de la clase actual. (No contamos la pareja formada por la clase vacía y la actual). Se muestra a continuación el pseudocódigo del algoritmo:

```
// Función que encuentra la mejor solución posible al problema de las
// cifras asociado a C y objetivo.
void resolver(){
    for (int i = 1; i < 2^n && mejor_solucion->resultado != objetivo; i++){
        generarClase(i);
    }
}
```

```
// Función que genera una clase a partir de sus subclases utilizando
// la proposición 4. Hay 2^{t-1}-1 parejas de subclases disjuntas
// que generen soluciones de la clase[i].
void generarClase(int i){
    int numero_subclases = 2^{t-1} - 1 // Razonamiento previo
    int subclases_usadas = 0;
    for (int j = 1; subclases_usadas < numero_subclases
    && mejor_solucion->resultado != objetivo; j++){
        if (clase[j] es subclase de clase[i] y no se ha utilizado){
            subclases_usadas++;
            // Se generan las soluciones posibles de la clase[i] a partir de las
            // subclases disjuntas clase[j] y clase[i-j]
            generarSoluciones(i, j);
        }
    }
}
```

La ventaja de este procedimiento para calcular todas las soluciones al Problema de las Cifras viene dada por la función **generarSoluciones(i,j)**. Una primera aproximación consiste en calcular para cada pareja de soluciones de las clases clase[j] y clase[i-j] las soluciones dadas en la **definición 6** y añadirlas a clase[i]. Sin embargo, el tamaño de cada clase aumentará enormemente al calcular "todas las ramas del árbol de soluciones". Podemos mejorar notablemente la escalabilidad del algoritmo si evitamos añadir soluciones equivalentes (**definición 7**) a la clase[i]. Por tanto, solo añadimos una solución de las calculadas a partir de las clases clase[j] y clase[i-j] si no existe otra con igual resultado en la clase[i] (en cuyo caso serían equivalentes). De esta forma reducimos notablemente el tamaño de las clases y con ello la eficiencia del algoritmo, calculando solo las soluciones útiles desde el punto de vista del resultado. Este proceso es lícito pues si tomamos dos soluciones equivalentes, *s'* y *s"* y otra disjunta con las anteriores, *s*, las soluciones obtenidas de operar *s'* con *s* y *s"* con *s* tendrán distinta expresión, pero igual resultado siendo inútil su cálculo. Así pues, obtenemos:

```
// Función que añade a la clase[i] las soluciones generadas por clase[j] y clase[i-j].
// max(s,s') devuelve la solución con mayor resultado. Análogo para min(s,s').
void generarSoluciones(int i, int j){
    for(int k = 0; k < clase[j].size(); k++){
        for(int m = 0; m < clase[i-j].size(); m++){
            s = solución k-ésima de la clase[j];
            s'= solución m-ésima de la clase[i-j];
            if (clase[i] no tiene una solución equivalente a s+s'){
                clase[i].push(s+s');
            }
```

```
            if (min(s,s') > 1 && clase[i] no tiene una solución equivalente a s*s'){
                clase[i].push(s*s');
            }
            if (clase[i] no tiene una solución equivalente a max(s,s')-min(s,s') ){
                clase[i].push(max(s,s')-min(s,s'));
            }
            if (min(s,s') > 1 && max(s,s')%min(s,s') == 0  && clase[i] no tiene una
            solución equivalente a max(s,s')-min(s,s') ){
                clase[i].push(max(s,s')-min(s,s'));
            }
        }
    }
}
```

Exigimos en la multiplicación y división que los resultados de ambas soluciones sean mayores que uno pues es inútil añadir una solución multiplicada o dividida por un uno. Las soluciones que en un futuro sean creadas por esta última ya habrían sido creadas (desde el punto de vista de su resultado) en una clase menor que tenga como subclase la clase[j] o la clase[i-j].

Nótese que utilizando esta última función estamos viendo una clase como un connjunto de soluciones semejantes pero no equivalentes. Es por tanto útil utilizar una adecuada implementación de conjunto para realizar este proceso.

4 Implementación

4.1 Interfaz de la implementación

Se proporcionan dos implementaciones diferentes para el problema utilizando las ideas expuestas anteriormente. La interfaz de ambas implementaciones es equivalente. Ambas constan de una carpeta llamada ProblemaCifras* con la correspndiente implementación. Dicha carpeta consta a su vez de las subcarpetas:

- bin : Aquí se guarda el ejecutable tras la compilación.
- doc : Aquí se guarda la documentación generada por Doxygen tras hacer make documentacion
- include : Aquí se encuentran los archivos .h
- src : Aquí se encuentran los archivos .cpp

Y los archivos:

- makefile : Compila el programa.
- parametros.par : Datos que se pasan como argumento al programa.

El archivo parametros.par tiene el siguiente contenido:

```
*************** Problemas De las Cifras ***************
NumeroDatos=6
Datos= 2 6 7 9 50 75
ValorObjetivo= 234
```

NumeroDatos es el cardinal del conjunto C y Datos los elementos del mismo. ValorObjetivo responde al valor objetivo del problema a resolver. La llamada al programa se realiza de la siguiente forma:

```
./bin/problemaCifras parametros.par
```

El resultado de la ejecución se muestra en la terminal. Por ejemplo, este sería el resultado obtenido para los datos anteriores usando solo vectores:

```
Tiempo de ejecución: 0.000574 segundos
Se ha conseguido el valor objetivo.
Operaciones realizadas:
2 * (75 + (6 * 7))
```

4.2 Representación de una solución

En el código nos desprendemos de la visión de solución como una expresión y el resultado de evaluar dicha expresión. La razón es simple: es costoso acarrear con una string que contenga la correspondiente expresión. Proponemos pues que una solución, dado el carácter binario de las operaciones, conste con los siguientes datos miembros:

- Resultado : Resultado de evaluar la expresión.
- Clase : Entero con la clave de la clase de equivalencia a la que pertenece.
- Padre1 y Padre2 : Punteros a las soluciones que la generaron al aplicarles a las mismas determinada operación binaria (+,*,- o /).
- Operación: Entero que indica la operación que se aplicó (1,2,3 y 4 respectivamente).

De esta forma, cuando se encuentre la solución que minimiza la distancia de su resultado al valor objetivo basta recorrer los punteros padres hasta llegar a las soluciones iniciales que la generaron mediante recursividad, imprimiento en cada paso la operación utilizada.

4.3 Implementación utilizando solo vectores: ProblemaCifrasVector

En este caso cada clase es un vector de la stl. El algoritmo funciona como se ha explicado anteriormente con la diferencia de que siempre se añade una solución obtenida a la clase (no disponemos de un mecanismo eficiente para ver si ya contiene una solución equivalente o no). Para disminuir el tamaño de las clases y aumentar la escalabilidad del algoritmo eliminamos las soluciones equivalentes una vez se ha generado la clase. Para ello basta ordenar la clase en función de los resultados de las soluciones con un *sort* de la STL de eficiencia $O(m \log m)$, donde m es el número de soluciones de la clase. A continuación, mediante un procedimiento lineal ($O(m)$) se recorre la clase asignando a las primeras componentes del vector las soluciones no equivalentes entre sí. Por último, se libera el espacio no usado del vector:

```
sort(clase[i].begin(), clase[i].end());
int lectura, escritura = 1;
for (lectura = 1; lectura < clase[i].size(); lectura++){
    if (! (clase[i][lectura] == clase[i][lectura-1])){
        clase[i][escritura] = clase[i][lectura];
        escritura++;
    }
}
for (int j = 0; j < lectura - escritura; j++){
    clase[i].pop_back();
}
```

4.4 Implementación utilizando la clase set: ProblemasCifrasSet

En este caso las clases están representadas por un *set* de la STL. Son pues un conjunto propiamente dicho donde el algoritmo funciona en su versión pura, solo se añade una solución a la clase si no hay ninguna otra equivalente (gracias a la función *set::insert()*). La ventaja de esta implementación es la escalabilidad, pues no es lo mismo no insertar soluciones de forma progresiva que eliminarlas de golpe después de crear la clase. El primer procedimiento es *O(m' log m')* y el segundo es *O(m log m)*, donde *m'* es el tamaño final de la clase (tras eliminar las equivalentes) y *m* el tamaño sin eliminar las equivalentes. Sin embargo, para problemas de 6 datos la diferencia es insignificante.

5 Ejercicio Extra: Combinaciones Mágicas

5.1 Explicación del algoritmo

Existen combinaciones de 6 números del conjunto A dado en la introducción con las que puede obtenerse cualquier número de 3 cifras. Un ejemplo es el siguiente conjunto:

$$A = \{2, 6, 7, 9, 50, 75\}$$

A estas combinaciones se las denominan **Combinaciones Mágicas**. Utilizo mi solución al Problema de las Cifras para calcular todas las Combinaciones Mágicas posibles. Para generalizar el problema, propongo un algoritmo que encuentre aquellas combinaciones (con repetición) de números de un conjunto dado tales que generen todos los valores entre un intervalo acotado de números enteros positivos dado por su máximo y mínimo. El programa se adjunta en la carpeta **CombinacionesMagicas**, que sigue la misma estructura de directorios que los dos programas anteriores. Para su uso basta compilarlo y llamarlo de la siguiente forma:

```
./bin/combinacionesMagicas parametros.par
```

En este caso el archivo parámetros responde a:

```
*************** Problemas De las Cifras, Combinaciones Mágicas ****************
NumeroDatos= 14                              // Número de datos de donde obtener combinaciones
Datos= 1 2 3 4 5 6 7 8 9 10 25 50 75 100     // Datos de donde obtener las combinaciones
LongitudCombinacion= 6                       // Longitud de las combinaciones a obtener
LimiteInferior= 100                          // Mímimo del intervalo
LimiteSuperior= 999                          // Máximo del intervalo
```

Estos datos son los propuestos en el reto. Para los mismos obtengo **1248 Combinaciones Magicas** en 655.395 segundos de cómputo. La ejecución se ha realizado en un ordenador portátil con 8 GB de RAM y procesador Intel i5 a 2,5 GH.

El algoritmo utilizado consiste en comprobar para cada posible combinación con repetición, de longitud dada (LongitudCombinacion), de los números dados como Datos si es o no Combinación Mágica para el intervalo dado. Para esto útimo basta ejecutar el algoritmo propuesto para resolver el Problema de las Cifras sin objetivo, calculando todas las clases y manteniendo la cuenta de las soluciones cuyo resultado está dentro del intervalo pedido. Si al final del algoritmo se han conseguido todos los números del intervalo entonces ¡es una Combinación Mágica!.

Para calcular todas las combinaciones con repetición de longitud LongitudCombinación basta usar el siguiente algoritmo recursivo llamándolo con un vector de espacio reservado y los demás argumentos a cero:

```
void calcularCombinacionesMagicas(vector <int> elementos_iniciales, int ultimo_elemento, int etapa){
    if (etapa == longitud_combinacion){
        Algoritmo solver(elementos_iniciales, limite_inferior, limite_superior);
        solver.resolver();
    }
    else{
        for (int i = ultimo_elemento; i < num_elementos; i++){
            elementos_iniciales[etapa] = elementos[i];
            calcularCombinacionesMagicas(elementos_iniciales, i, etapa+1);
        }
    }
}
```

El funcionamiento es el siguiente. Cada combinación con repetición está unívocamente determinada como un vector ordenado. Basta calcular todos los vectores ordenados con valores dentro del conjunto dado como Datos y de tamaño *longitud_combinación* para tener todas las posibles combinaciones con repetición. Este hecho consiste en asignar progresivamente las componentes de forma ordenada y con todos los elementos posibles. Por ejemplo, para un vector de longitud tres sería:

```
for (int i1 = 0; i1 < num_elementos; i1++){
    for (int i2 = i1; i2 < num_elementos; i2++){
        for (int i3 = i2; i3 < num_elementos; i3++){
            elementos_iniciales[0] = elementos[i1];
            elementos_iniciales[1] = elementos[i2];
            elementos_iniciales[2] = elementos[i3];
            Algoritmo solver(elementos_iniciales);
            solver.resolver();
        }
    }
}
```

La función recursiva no es más que una generalización de este algoritmo para cualquier longitud de vector. Nótese que las combinaciones con repetición de n elementos tomados de r en responden a:

$$CR(n,r) = \frac{(n+r-1)!}{r!(n-1)!}$$

En nuestro caso, para los valores $n = 14$ y $r = 6$ hay un total de 27132 combinaciones con repetición, de las cuales solo 1248 son mágicas.

5.2 Las 60 primeras combinaciones mágicas

```
Combinación mágica 1: 1 2 3 4 5 100
Combinación mágica 2: 1 2 3 4 9 75
Combinación mágica 3: 1 2 3 4 10 75
Combinación mágica 4: 1 2 3 4 10 100
Combinación mágica 5: 1 2 3 5 9 100
Combinación mágica 6: 1 2 3 6 9 100
Combinación mágica 7: 1 2 3 7 9 100
Combinación mágica 8: 1 2 3 7 10 75
Combinación mágica 9: 1 2 3 7 10 100
Combinación mágica 10: 1 2 3 8 10 50
Combinación mágica 11: 1 2 3 8 10 100
```

```
Combinación mágica 12: 1 2 3 9 10 75
Combinación mágica 13: 1 2 3 10 75 100
Combinación mágica 14: 1 2 4 5 7 100
Combinación mágica 15: 1 2 4 5 8 100
Combinación mágica 16: 1 2 4 5 9 100
Combinación mágica 17: 1 2 4 5 50 75
Combinación mágica 18: 1 2 4 6 7 75
Combinación mágica 19: 1 2 4 6 9 50
Combinación mágica 20: 1 2 4 6 9 100
Combinación mágica 21: 1 2 4 7 9 50
Combinación mágica 22: 1 2 4 7 9 75
Combinación mágica 23: 1 2 4 7 9 100
Combinación mágica 24: 1 2 4 7 10 100
Combinación mágica 25: 1 2 4 9 10 75
Combinación mágica 26: 1 2 4 9 10 100
Combinación mágica 27: 1 2 4 9 25 75
Combinación mágica 28: 1 2 4 9 25 100
Combinación mágica 29: 1 2 4 9 50 75
Combinación mágica 30: 1 2 4 9 75 100
Combinación mágica 31: 1 2 4 10 25 100
Combinación mágica 32: 1 2 4 10 75 100
Combinación mágica 33: 1 2 5 6 8 75
Combinación mágica 34: 1 2 5 6 8 100
Combinación mágica 35: 1 2 5 6 9 100
Combinación mágica 36: 1 2 5 6 25 75
Combinación mágica 37: 1 2 5 7 8 75
Combinación mágica 38: 1 2 5 7 8 100
Combinación mágica 39: 1 2 5 7 9 100
Combinación mágica 40: 1 2 5 7 10 75
Combinación mágica 41: 1 2 5 7 25 75
Combinación mágica 42: 1 2 5 7 75 100
Combinación mágica 43: 1 2 5 8 8 100
Combinación mágica 44: 1 2 5 8 9 50
Combinación mágica 45: 1 2 5 8 9 100
Combinación mágica 46: 1 2 5 8 50 100
Combinación mágica 47: 1 2 5 8 75 100
Combinación mágica 48: 1 2 5 9 25 50
Combinación mágica 49: 1 2 5 9 25 100
Combinación mágica 50: 1 2 5 9 50 100
Combinación mágica 51: 1 2 5 9 75 100
Combinación mágica 52: 1 2 6 7 8 100
Combinación mágica 53: 1 2 6 7 9 100
Combinación mágica 54: 1 2 6 7 10 75
Combinación mágica 55: 1 2 6 7 10 100
Combinación mágica 56: 1 2 6 8 9 50
Combinación mágica 57: 1 2 6 8 9 100
Combinación mágica 58: 1 2 6 8 10 100
Combinación mágica 59: 1 2 6 8 25 75
Combinación mágica 60: 1 2 6 8 25 100
```