Reto 2

Pablo Baeyens Fernández
José Manuel Muñoz Fuentes
Darío Sierra Martínez

Estructura de Datos

1. Introducción

1.1. Problema

Definición. Sea C un multiconjunto. Se dice que $n \in \mathbb{N}$ es **generable por** C si $n \in C$ o existen a, b generables por C (generadores de n) tales que:

- $n = a \circ b$ para cierta operación $\circ \in \{+, -, /, \cdot\}$.
- Si $a, b \in C$, $a \neq b$ contando repetición y en otro caso sus generadores son distintos.

El conjunto de los elementos generables por C lo notaremos C^*.

El problema consiste en describir el procedimiento de un algoritmo con la siguiente entrada y salida:

Entrada: La entrada consiste de:

- Multiconjunto C de 6 elementos de $C_T = ([1, 10] \cap \mathbb{N}) \cup \{25, 50, 75, 100\}$ no necesariamente distintos.
- Entero $S \in [100, 999]$ llamado **solución**.

Salida: Lista de operaciones binarias básicas (suma $+$, resta $-$, producto $\cdot$, cociente $/$) tales que:

1. Cada operación usa sólo elementos generables por C que o bien estaban en C o bien se han obtenido a través de una operación anterior en la lista.
2. Cada número se utiliza como máximo una vez (contando repeticiones).
3. El último resultado es $T \in C^*$ tal que $|S - T| = \min_{N \in C^*} |S - N|$
4. Todo número utilizado en las operaciones es un entero no negativo.

Así, el algoritmo deberá llegar a un número lo más cercano posible a S efectuando operaciones con elementos de C, pudiendo usar un número solo una vez por cada aparición del mismo y pudiendo emplear números obtenidos mediante operaciones previas.

Dado que solo utilizamos estructuras de datos lineales, llamaremos L a la lista que se identifica con el multiconjunto C.

1.2. Operaciones posibles

Podemos notar que para cumplir las condiciones de la salida del algoritmo, dados 2 elementos $a, b \in \mathbb{N}^*$ existen cómo máximo 4 operaciones válidas:

1. $a + b$. Basta considerar un orden ya que es conmutativa.
2. $a \cdot b$. De nuevo, basta considerar un orden ya que es conmutativa.

3. $|a-b|$. Esta operación consiste en la resta del menor al mayor. No es necesario considerar la otra resta porque $a < b \implies a - b < 0$ y $a = b \implies a - b = b - a = 0$ (en este caso ninguna resta es posible).

4. $\text{máx}\{a,b\}/\text{mín}\{a,b\}$. Esta operación es el cociente del mayor entre el menor, y solo es posible si el menor divide al mayor y el menor no es igual a 0. Solo hay como máximo un orden posible porque $a < b \implies b \nmid a$ y $a = b \implies a/b = b/a$.

Sea $a \circ b = c$ una cierta operación. Podemos considerar algunos criterios para reducir el número de operaciones:

- Si $c = a$ o $c = b$ la operación puede omitirse de cualquier solución, dado que habrá otra solución que omite este paso y simplemente usa a o b, respectivamente, en lugar de c.
- Si $c = 0$ la operación puede omitirse de cualquier solución, ya que de un 0 solo puede obtenerse otro 0 o un valor ya existente. Podemos redefinir la división como 0 cuando no es válida y entrará en ese caso.
- Si a surgió de la misma operación que se está realizando $(d \circ e = a, (d \circ e) \circ b = c)$, puede omitirse porque:
  - Si $\circ$ es conmutativa ($+$ o $\cdot$) y por tanto asociativa, $(d \circ e) \circ b = c$ equivale a $d \circ (e \circ b) = c$. Podemos hacer solo este segundo caso y omitir el primero.
  - Si $\circ$ no es conmutativa ($-$ o $/$) y es válida, $(d \circ e) \circ b = c$ equivale a $d \circ (e \circ^{-1} b) = c$ siendo $\circ^{-1}$ la operación opuesta. Podemos hacer solo este segundo caso y omitir el primero.
- Si b surgió a partir de la misma operación $\circ$ que se está realizando $(d \circ e = b, a \circ (d \circ e) = c)$, en algunos casos también es posible descartar la operación; a saber:
  - Si $\circ$ no es conmutativa y es válida, la expresión $a \circ (d \circ e) = c$ equivale a $(a \circ^{-1} e) \circ d = c$. Podemos hacer solo este segundo caso y omitir el primero.
  - Si $\circ$ es conmutativa y la posición de d en L es anterior a la de a, $a \circ (d \circ e) = c$ equivale a $d \circ (a \circ e) = c$. Podemos hacer solo este segundo caso y omitir el primero.

Decimos que una operación es **válida** si no puede omitirse por lo expuesto anteriormente.

2. Primer algoritmo

2.1. Almacenamiento y presentación de resultados

Este algoritmo tiene como objetivo construir todos los caminos posibles que pueden seguirse usando los números de C y las operaciones disponibles, dando una solución exacta en cuanto se encuentre o el camino con una solución lo más cercana posible a S. Para ello, ampliará L (que era la lista que se identificaba con C) con listas de 4 elementos $[n, i, j, op]$, donde n es un entero obtenido a partir de los elementos que ocupan las posiciones i y j en L mediante la operación op, según se definieron las operaciones en la introducción.

Por comodidad, definiremos L_i como el i-ésimo elemento de L, haremos referencia a los elementos de L_i con $L_i.[n/i/j/op]$, y "el valor de L_i" hará referencia a $L_i.n$ si L_i es lista o a L_i si es un entero. Esta estructura de L nos permitirá obtener la secuencia de pasos que se han seguido hasta llegar a cualquier elemento de L mediante el siguiente procedimiento recursivo:

Entrada: L y N, con $N = [n, i, j, op]$ o $N = n$
Salida: Las operaciones con elementos de L para llegar a n
si $N \neq n$ **entonces**
  Aplica este algoritmo con entrada L, L_i;
  Aplica este algoritmo con entrada L, L_j;
  si *op no es conmutativa y* $L_i.n < L_j.n$ **entonces**
    intercambia i y j;
  fin
  devolver `a operador b = n`, *con* $a = L_i.n$, $b = L_j.n$, `operador` *una representación de op*
fin

Algoritmo 1: Obtención de operaciones

Se observa que este procedimiento conmuta correctamente los operandos en el caso de las restas y los cocientes a la hora de mostrar la operación, por lo que el algoritmo no tendrá que preocuparse por almacenar el orden en el que se encontraban los operandos (aunque sí por efectuar correctamente cada operación).

2.2. Algoritmo

Nota: en esta sección se definen funciones para poder describir el algoritmo. Estas funciones no tienen por qué existir en un hipotético programa que ponga en práctica este algoritmo. De hecho, ofrecería un mejor resultado introducir los valores que toman estas funciones para cada L_i como un elemento más en la lista que lo constituye.

Definición. Sea $L_k \in L$:

- L_k es de **primera generación**, $\text{gen}(L_k) = 1$ si $L_k \in C$ (es decir, si es uno de los 6 elementos iniciales).
- L_k es de n**-ésima generación**, $\text{gen}(L_k) = n$ si $\text{gen}(L_{L_k.i}) + \text{gen}(L_{L_k.j}) = n$.

Los **miembros** de una generación i son $\text{miembros}(i) = \{L_k : gen(k) = i\}$.

La generación representa el número de elementos que han sido necesarios para obtener L_k, o también el número de operaciones que se han requerido más 1. El algoritmo se ejecutará de forma que, en cada iteración, se obtendrán elementos que serán exclusivamente de una generación en particular, controlando la generación de los dos elementos de los que procede cada uno.

Los conjuntos de miembros nos permiten diferenciar los elementos de L según en qué momento de la ejecución del algoritmo han sido obtenidos y determinar el inicio y el final de los bucles que recorren los elementos con los que se operará en cada paso. Como el algoritmo generará los elementos de generación en generación, será posible determinar la generación de un elemento por su posición.

Definición. Sea $L_k \in L$. Los **números usados por** L_k, $U(L_k)$ son:

- $U(L_k) = \{L_k\}$ si $L_k \in C$
- $U(L_k) = U(L_{L_k.i}) \cup U(L_{L_k.j})$, en otro caso.

Dos elementos de L, L_a y L_b, **se solapan** si $U(L_a) \cap U(L_b) \neq \emptyset$.

De esta forma, $U(L_k)$ representa el conjunto de números de C que han intervenido para obtener L_k, y dos elementos se solaparán cuando tengan algún elemento de C en común. Esto será crucial para poder comprobar a cada paso del algoritmo si es posible operar con dos elementos entre sí: cada par de elementos que se usen para añadir uno nuevo deben no solaparse.

Se almacenará en N la mejor aproximación de S. Cuando termine el algoritmo, se consultará N y se generarán las operaciones realizadas para obtener N con el algoritmo anterior.

Finalmente, el algoritmo queda así:

Entrada: $C = \{c_1, c_2, c_3, c_4, c_5, c_6\}$, $S \in [100, 999]$
Salida: L y N con $|N - S|$ mínima.
$L = [c_1, c_2, c_3, c_4, c_5, c_6]$;
$N = c_6$;
para todo g **desde** 2 **hasta** 6 **hacer**
  para todo i **tal que** $2 \cdot \text{gen}(L_i) \leq g$ **hacer**
    para todo $j \in miembros(g - \text{gen}(L_i))$ **tal que** $j > i$ **hacer**
      si $U(L_i) \cap U(L_j) = \emptyset$ **entonces**
        para todo $op \in \{+, -, \cdot, /\}$ **hacer**
          si $[n, i, j, op]$, *con* $n = L_i.n$ *op* $L_j.n$, *es* ***válida*** **entonces**
            Añadir $[n, i, j, op]$ a L;
            si $|n - S| < |N.n - S|$ **entonces**
              $N.n \leftarrow [n, i, j, op]$;
              si $N.n = S$ **entonces**
                devolver L, N
              fin
            fin
          fin
        fin
      fin
    fin
  fin
fin
devolver L, N

Algoritmo 2: Obtención de la solución

Puede observarse que, cuando $g = 6$, si el nuevo n no está más cerca de S que $N.n$, no es necesario añadir $[n, i, j, op]$ a L. Añadir esta condición oscurece el algoritmo, pero se tendrá en cuenta en la implementación.

Para definir el inicio y el final de los bucles que recorren L, se empleará una lista G de 6 elementos, que indicará en la posición G_i el primer elemento de L que es de i-ésima generación. Definiremos también G_7 como la posición siguiente a la del último elemento de L. La función miembros(i) podría redefinirse aprovechando G como $\{L_k : k \geq G_i, k < G_{i+1}\}$.

Aprovechando G, podremos determinar con precisión el inicio y el final de los bucles que recorren los elementos del vector. Se explorará L de forma que cada par de elementos considerados sumará g en todo momento y la generación de L_i será menor o igual que la de L_j.

El bucle que recorre los i debe terminar al final de la última generación $\bar{g}$ tal que $2 \cdot \bar{g} \leq g$, que es la posición anterior a la que comienza la generación posterior. Tal posición es $G_{((g+1)/2+1)} - 1$, usando la división entera (con truncamiento).

El bucle que recorre los j debe recorrer la generación $\bar{g}$ que $\bar{g}+gen(L_i) = g$; pero si ambas fuesen la misma, debería empezar después de i. Así, el inicio estará en máx$\{i+1, G_{(g-gen(L_i))}\}$ y el final en $G_{(g-gen(L_i)+1)} - 1$.

Con los bucles escritos de esta forma, el algoritmo queda así:

Entrada: $C = \{c_1, c_2, c_3, c_4, c_5, c_6\}$, $S \in [100, 999]$
Salida: L y N con $|N - S|$ mínima.
$L = [c_1, c_2, c_3, c_4, c_5, c_6]$;
$N = c_6$;
para todo g **desde** 2 **hasta** 6 **hacer**
  para todo i **desde** 1 **hasta** $G_{((g+1)/2+1)} - 1$ **hacer**
    para todo j **desde** máx$\{i + 1, G_{(g-gen(L_i))}\}$ **hasta** $G_{(g-gen(L_i)+1)} - 1\}$ **hacer**
      si $U(L_i) \cap U(L_j) = \emptyset$ **entonces**
        para todo $op \in \{+, -, \cdot, /\}$ **hacer**
          si $[n, i, j, op]$, *con* $n = L_i.n$ *op* $L_j.n$, *es* ***válida*** **entonces**
            Añadir $[n, i, j, op]$ a L;
            si $|n - S| < |N.n - S|$ **entonces**
              $N.n \leftarrow [n, i, j, op]$;
              si $N.n = S$ **entonces**
                devolver L, N
              fin
            fin
          fin
        fin
      fin
    fin
  fin
fin
devolver L, N

Algoritmo 3: Obtención de la solución (bucles for)

2.3. Características del algoritmo

Este algoritmo explora todas las posibles operaciones válidas. Por ello, garantiza encontrar la solución más cercana posible a la pedida. Además, dado que construyen las generaciones una a una y en orden ascendente, se garantiza que el número de operaciones usadas para obtener un valor que se distancie lo menos posible a la solución será el mínimo, tanto en el caso de resultado exacto como en caso de valor aproximado.

Este algoritmo destaca por el bajo tiempo de ejecución de su implementación. Usando la versión modificada para hallar combinaciones mágicas de este algoritmo, se han obtenido todas las combinaciones mágicas en $52,1$ segundos.

El algoritmo no ordena los operandos en cada resta o cociente aunque sea necesario. En cualquier caso, como siempre hay un único resultado válido, el algoritmo que presenta la solución se encargará de mostrar los operandos en el orden correcto.

2.4. Implementación

Se usarán dos tipos de datos `Nodo` y `VectorNodos` que representarán a cada L_k y a L respectivamente.

2.4.1. Tipo de dato para L_k

Cada L_k debe estar representado por una estructura que incluya, como mínimo, tipos de datos que se identifiquen con la descripción que se dio anteriormente ($[n, i, j, op]$).

n será un tipo de dato entero con signo de 32 bits. Es importante que el tamaño sea de al menos 32 bits, dado que si fuese de 16 bits sería posible salirse del rango y obtener resultados falsos o perder resultados correctos.

- Un ejemplo de resultado falso: si trabajamos con enteros de 16 bits con signo, $100^2 \cdot 75$ daría como resultado 29104, que podría dar lugar a una solución falsa. Con $C = \{4, 6, 8, 75, 100, 100\}$, $S = 911$ (ejemplo de problema para el que no existe una solución exacta) se obtendría esta falsa solución: $100 \cdot 100 = 10000; 10000 \cdot 75 = 29104; 29104/8 = 3638; 3638 + 6 = 3644; 3644/4 = 911$.
- Un ejemplo de problema que no podría resolverse sería $C = \{3, 3, 25, 50, 75, 100\}$, $S = 996$. Toda solución a este ejemplo pasa por obtener el número 99600, que está fuera del rango de los enteros de 16 bits con o sin signo (si se ejecuta el algoritmo haciendo que se rechace todo resultado temporal igual a 99600 de forma similar a como se hace con el 0, no se obtiene solución exacta). Una solución (podría ser la única) es: $3 \cdot 50 = 53; 25 \cdot 53 = 1325; 3 + 1325 = 1328; 75 \cdot 1328 = 99600; 99600/100 = 996$.

Con un entero de 32 bits también es posible salirse del rango (con 100^5, por ejemplo), pero si el entero tiene signo, al provocar desbordamiento obtendremos un número negativo, por lo que se cancelará y nunca obtendremos un resultado falso; y por otro lado se requiere operar con todos los números menos uno para llegar a provocar desbordamiento, pero, en tal situación, al dividir por un número que como máximo será 100 en ningún caso el resultado será menor que 1000, por lo que tampoco es posible que se pierda una solución correcta.

i y j serán dos enteros de 32 bits que indicarán la posición de los L_i y L_j de los que se obtuvo el nodo anterior (como se verá en la siguiente sección, el tamaño de L es menor que 2^{31}). Dado que los L_k deben tener todos la misma estructura, y en lo propuesto los seis primeros elementos (que procedían de C) estaban compuestos solo por un número, para esos seis elementos iniciales podrían tomar un valor arbitrario (por ejemplo, el valor inválido -1, lo que requeriría enteros con signo), aunque en la implementación no se optará por ello.

op representará la operación con la que se obtuvo el elemento. Podría representarse con cualquier tipo de dato; optaremos por un entero de 16 bits. Para los seis primeros elementos, podría tomar un valor arbitrario, preferiblemente inválido.

De cara al algoritmo que genera las operaciones con las que se llegó a un elemento, interesará que, en los seis primeros elementos, al menos uno de los tres últimos elementos que componen esta estructura tuviese un valor inválido, dado que el algoritmo debe no hacer nada cuando está siendo ejecutado con uno de los seis primeros elementos como entrada. En la implementación optaremos por darle un valor inválido al campo *op*. Esto facilitará la labor a la función que comprueba si uno de los operandos procede de la misma operación.

Además de los datos anteriores, que son indispensables, probablemente interesará hacer (y en la implementación se ha hecho) lo siguiente:

- Incluir un entero que indique la generación del elemento podría reducir el tiempo de ejecución del algoritmo respecto del tiempo que necesitaría si emplease una función que calculase la generación de un elemento explorando los elementos de los que procede. Un entero de 16 bits que tomase el valor de la suma de las generaciones de los elementos de los que procede y que tomase el valor 1 para los elementos iniciales sería suficiente para ello.
- Sería particularmente interesante incluir un entero sin signo de al menos 16 bits que indique en el i-ésimo bit menos significativo si se ha usado el i-ésimo elemento del multiconjunto de números disponibles C. Tal bit sería 1 si lo ha usado, o 0 si no. Esto supondría que la comprobación de si dos elementos se solapan sería tan simple como comprobar si la operación lógica AND de ambos es un valor no nulo. Al añadir nuevos elementos a L, este dato tomaría el valor de la operación lógica OR sobre los datos en los dos elementos de los que se obtiene el nuevo. Por ejemplo, si el valor de este dato para dos elementos fuese 001011 y 110000, el elemento que se obtendría al operar con ambos tendría en este dato el valor 111011. Este dato tomaría, para el elemento inicial i-ésimo, el valor 2^{i-1} o $1 << i$. (1 para el primero, 2 para el segundo, 4 para el tercero...)

2.4.2. Tipo de dato para L

L debe estar representado por una clase que permita, al menos, añadir elementos L_k. Dado que el número de elementos es demasiado como para almacenarlos en la pila, se optará por una estructura lineal situada en memoria dinámica. El tipo de dato reservará espacio para un número suficiente (según se calculará a continuación) de elementos del tipo de dato descrito anteriormente, y además almacenará las posiciones en las que se inicia cada generación de elementos, para controlar dónde deben comenzar y acabar los bucles que recorren los elementos.

Convendrá que esta clase disponga de métodos para añadir elementos, consultar los elementos actuales, obtener la posición de inicio de cada generación y marcar como final de la generación el último elemento actual, además de un destructor que libere la memoria reservada. Para las combinaciones mágicas, se modificará ligeramente esta estructura.

2.4.3. Tamaño de L

El número de elementos de L para los que debe reservarse memoria viene determinado por la suma de elementos de cada generación:

- Para la primera generación tenemos seis elementos: los seis números iniciales.
- La segunda generación estará conformada por los resultados de operar con pares de elementos de la primera generación.
- La tercera reunirá resultados de operar con un elemento de la primera y otro de la segunda.
- La cuarta estará constituida tanto por operaciones con un elemento de la primera y otro de la tercera como por operaciones con pares de elementos de la segunda.
- La quinta tendrá elementos que procederán de otros previos de las generaciones 1 y 4 o 2 y 3.
- La sexta surgirá al operar elementos de las generaciones 1 y 5, 2 y 4 o 3 y 3. Pueden descartarse los elementos de esta generación que no se acerquen al objetivo más que otros elementos previos.

En general, los elementos de la generación n, para $n > 1$, proceden de pares de elementos de las generaciones i, j con $i \leq n/2$ tales que $i + j = n$.

Si decidimos no añadir los elementos de la sexta generación a no ser que tomen un valor más cercano al objetivo S que todos los demás, en esta generación habrá menos de 994 elementos, dado que con cualquier combinación de 5 números iniciales es posible llegar a, al menos, el valor 6 ($(1+1+1)*(1+1)$), y aun en tal caso es imposible que haya 994 elementos que, uno detrás de otro, sean mejores que el anterior: si hubiese 993 seguidos, el último sería ya la solución para cualquier solución entre 100 y 999.

Sea $T(i)$ el número de elementos que tendremos de la generación i-ésima. El número de espacios que deberán reservarse en memoria será $T = \sum_{i=1}^{6} T(i)$, o $T = \sum_{i=1}^{5} T(i) + 993$ si excluimos la sexta generación. Ahora calcularemos una cota superior para T suponiendo que todas las operaciones son válidas (es decir, que podremos hacer para cada pareja que no se solape las cuatro operaciones).

A la hora de operar con elementos de dos generaciones i, j no necesariamente distintas, para cada elemento de la i-ésima generación el número de elementos que puede obtenerse de él será igual al número de elementos de j-ésima generación que pueden combinarse con tal elemento de i (sea esto $P(i, j)$), 4 veces (una por operación, dado que suponemos que todas son válidas), es decir, $4 \cdot T(i) \cdot P(i, j)$.

Sea L_i un elemento de i-ésima generación. Si entendemos los elementos usados por L_i como una lista de elementos 1 o 0 que indican si el número inicial que se encuentra en esa posición ha sido usado o no (es

prácticamente lo que se hace en la estructura de L_k usando los bits de un entero como booleanos), tal lista tendrá i unos y $6-i$ ceros, dado que L_i habrá usado i números iniciales. Para poder combinarse con L_i, los elementos de j-ésima generación L_j deberán cumplir que los números usados por L_i no lo estén. Por tanto, los j números usados por L_j estarían repartidos por los no usados por L_i. Dado que hay C_n^k formas de seleccionar k elementos de un grupo de n, y puesto que en cada generación el número de elementos cuya lista de elementos sea una en particular es el mismo para cualquier lista válida, la cantidad de elementos de j-ésima generación disponibles para combinar con L_i será $P(i,j) = T(j) \cdot \dfrac{C_{6-i}^j}{C_6^j}$.

Por ello, si las generaciones i y j son distintas, el número de elementos que pueden obtenerse a partir de elementos de ambas es $4 \cdot T(i) \cdot P(i,j) = 4 \cdot T(i) \cdot T(j) \cdot \dfrac{C_{6-i}^j}{C_6^j}$. Si $i = j$, tal número se reduce a la mitad, puesto que solo hay una forma de combinar cada par de números. Por comodidad definiremos la función $I(i,j) = 1/2 \; si \; i = j, \; 1 \; si \; i \neq j$. La función que determina cuántos elementos hay en una generación (a la que llamamos previamente $T(i)$) queda así:

$$T(1) = 6; \; T(n) = \sum_{i=1}^{n/2} 4 \cdot T(i) \cdot T(n-i) \cdot \frac{C_{6-i}^{n-i}}{C_6^{n-i}} \cdot I(i, n-i) \;\; \forall n \in \{2,3,4,5,6\}$$

Ya podemos obtener el tamaño máximo. Si incluye toda la sexta generación, $T = \sum_{i=1}^{6} T(i) = 1\ 144\ 386$, o si no, $T = \sum_{i=1}^{5} T(i) + 993 = 177\ 699$. Podremos fijar el tamaño de la estructura que representa a L con ese valor (o cualquier valor mayor, pero eso supondría un desperdicio de memoria).

Cabe observar que podría acotarse más el tamaño si se tienen en cuenta las dos últimas condiciones que debía cumplir una operación para ser válida (que el primer elemento no proceda de la misma operación y que el segundo elemento no proceda de la misma operación en ciertos casos). Sin embargo, dado que en la implementación no se observa diferencia en el rendimiento si se reserva memoria para $10T$ espacios en lugar de T, aceptaremos la cota T.

3. Segundo Algoritmo

Mientras que el algoritmo anterior construía todos los caminos empezando por los menos profundos y los exploraba a cada vez mayor profundidad, este algoritmo optará por explorar los caminos en profundidad. Esto tendrá la ventaja de que no será necesario mantener en memoria todos los caminos en ningún momento, pero tendrá el problema de que, si el resultado exacto no se encuentra, habremos perdido el camino por el que se llegó al mismo. El procedimiento será recursivo.

Tomaremos L como entrada. Combinamos pares de elementos de L con las operaciones según se describieron, obteniendo 60 elementos. Comprobamos si hemos llegado al objetivo: si el elemento está entre estos elementos, lo hemos encontrado y podremos reconstruir la secuencia de operaciones. De lo contrario, para cada elemento obtenido (o hasta que encontremos el número), creamos un vector $\bar{L}$ con este elemento y los elementos de L que no se han usado para crearlo y repetimos el proceso con $\bar{L}$ como entrada.

El algoritmo es el siguiente:

Entrada: L,S
Salida: Operaciones hasta llegar a S
Sean $b_0, .., b_m$ resultados de efectuar todas las operaciones posibles sobre pares L_p y L_q con $p < q$;
para todo i *desde* 0 *hasta* m **hacer**
  si $b_i = S$ **entonces**
    devolver *Devuelve los* L_j, L_k *que se usaron para crear el* b_i
  en otro caso
    Repetir el algoritmo con $\{L_j : L_j$ no fue usado para obtener $b_i\} \cup b_i$;
  fin
fin

3.1. Características del algoritmo

La principal característica de esta implementación es que solo tenemos en memoria aquellos elementos necesarios para seguir avanzando. Una vez que ya hemos recorrido todos los posibles elementos asociados al elemento b_i, estos dejan de ser necesarios y se eliminan de la memoria.

Sin embargo, el principal problema es que, tal cual está ahora, no permite hallar la mejor aproximación.

3.2. Mejor Aproximación

El problema de encontrar la mejor aproximación es bastante sencillo una vez ya creado el algoritmo de búsqueda, basta con crear un campo nuevo en el que vamos guardando la mejor aproximación, M_a, que debe ser externo a la función:

Entrada: L,S
Salida: Operaciones hasta llegar a S
Sean $b_0, .., b_m$ resultados de efectuar todas las operaciones posibles sobre pares L_p y L_q con $p < q$;
para todo i *desde* 0 *hasta* m **hacer**
  si $b_i = S$ **entonces**
    devolver *Devuelve los* L_j, L_k *que se usaron para crear el* b_i
  en otro caso
    si $|b_i - M_a| < |M_a - S|$ **entonces**
      $M_a \leftarrow b_i$;
    fin
    Repetir el algoritmo con $\{L_j : L_j$ no fue usado para obtener $b_i\} \cup b_i$;
  fin
fin

Ahora podemos identificar el resultado aproximado más cercano. Sin embargo sigue sin ser posible construirlo; tendremos que llamar de nuevo al algoritmo con el resultado aproximado más cercano para poder construir las operaciones.

4. Tercer algoritmo

4.1. Almacenamiento y presentación de resultados

Este tercer y último algoritmo recorre por profundidad todos los caminos posibles utilizando un vector ordenado. Su principal ventaja es su **reducido uso de memoria**. Las estructuras de datos que utiliza el

algoritmo son 3:

Operaciones Una operación es una tripleta $(a, b, \circ)$ que representa la operación $a \circ b$.

Pila El algoritmo utiliza una pila de operaciones que usará, como máximo un total de 6 elementos. Notaremos la pila como P y sus elementos (al tratarla como lista) como P_i. La pila **almacena las operaciones realizadas**.

Lista ordenada Esta lista **ordenada** contiene **los elementos con los que se opera** en cada iteración. Debido a las transformaciones realizadas su tamaño variará entre **2** y **6** elementos. La notaremos como L y sus elementos como L_i.

Para la representación de resultados debemos acceder a la pila como si fuera un vector para hacer que el algoritmo sea más eficiente. Describimos un algoritmo recursivo que nos permite imprimir las operaciones hasta llegar al final. Utilizamos 3 funciones auxiliares que dependen de la implementación y que son:

- $R((a, b, \circ)) = a \circ b$
- $\text{Arg1}((a, b, \circ)) = a$
- $\text{Arg2}((a, b, \circ)) = b$

Con estas operaciones podemos describir el algoritmo:

```
Imprime(U, n):
Entrada: U, vector que nos marca los elementos usados, n posición a imprimir
si R(P_n) ∈ L y no ha sido usado entonces
    Márcalo como usado ;
    Termina;
fin
para cada i ∈ [0, n − 1] hacer
    si R(P_i) = Arg1(P_i) entonces
        Imprime(U, i);
        Sal del bucle;
    fin
fin
para cada i ∈ [0, n − 1] hacer
    si R(P_i) = Arg2(P_i) entonces
        Imprime(U, i);
        Sal del bucle;
    fin
fin
Imprimir P_n;
```

De esta forma, sólo tendremos que llamar `Imprime`(v, n) con n el último índice de la pila y v un vector de 6 elementos booleanos falsos para imprimir la solución.

4.2. Caso general

El algoritmo es un algoritmo recursivo y presentamos de forma separada el caso base y el caso general. En el caso general recorremos L y realizamos para cada pareja válida cada operación. Reducimos por tanto

el tamaño de L a exactamente un elemento menos y llamamos recursivamente al algoritmo:

```
Cifras(L, S, M_a, P):
Entrada: L, lista con n > 2 elementos, S, M_a, P
Salida: Si se ha conseguido llegar al objetivo
para cada ∘ ∈ {+, −, ·, /} hacer
    si ∘ usa 1 entonces
        i := 0;
    fin
    en otro caso
        i := mín_{k>i} L_k ≠ 1;
    fin
    mientras i < |L| hacer
        j := i + 1;
        mientras j < |L| hacer
            R := L_j ∘ L_i;
            si R no es válido entonces Continuar;
            Poner (L_j, L_i, ∘) en P;
            Borrar L_i, L_j de L;
            Insertar R en L;
            si |R − S| < |M_a − S| entonces
                M_a := R;
                si M_a = S entonces devolver true;
            fin
            si Cifras(L, S, M_a, P) entonces devolver true;
            Quitar (L_j, L_i, ∘) de P;
            Borrar R de L;
            Insertar L_i, L_j en L;
            j := mín_{k>j} L_k ≠ L_j;
        fin
        i := mín_{k>i} L_k ≠ L_i;
    fin
fin
devolver false
```

Algoritmo 4: Caso general del tercer algoritmo

Aprovechamos que el algoritmo está ordenado para evitar comprobar casos duplicados y evitar productos y divisiones por 1. La pila puede almacenar resultados basura que no contribuyan a llegar al resultado, por lo que debemos imprimir sólo las que necesitamos de acuerdo al algoritmo de impresión visto en la sección anterior.

4.3. Caso base

El **caso base** sucede cuando $|L| = 2$ es decir, tenemos una lista de la forma $[L_0, L_1]$ con $L_0 \leq L_1$. En tal caso recorremos todas las combinaciones y si alguna llega a la solución, la devolvemos:

Entrada: L, lista con 2 elementos y S
Salida: Si se ha conseguido llegar al objetivo
para cada $\circ \in \{+, -, \cdot, /\}$ **hacer**
  $R := L_1 \circ L_0$;
  si R *no es válido* **entonces**
    Continuar;
  fin
  si $|R - S| < |M_a - S|$ **entonces**
    $M_a := R$;
    si $M_a = S$ **entonces**
      Poner $(L_1, L_0, \circ)$ en P;
      devolver *true*
    fin
  fin
fin
devolver *false*

La operación no será válida si no cumple las condiciones indicadas en la introducción, es decir, el resultado es uno de los operandos o es 0.

4.4. Características del algoritmo

Cada operación realizada sobre la lista L antes de la llamada recursiva borra dos elementos e inserta uno, lo que reduce el tamaño de ésta en 1. Además, cuando restauramos el estado inicial de la lista, esta no puede aumentar su tamaño. Es decir, la lista **no puede tener nunca más de 6 elementos**, la cantidad de elementos que tiene inicialmente.

Por otra parte, la pila P sólo puede aumentar de tamaño al reducirse el de L, lo que limita su tamaño también a **un máximo de 6 elementos** (cada uno de ellos con 3 enteros). Es decir, además de las variables locales sólo necesitamos tener en memoria un total de **24 enteros** para ejecutar el algoritmo.

Esto nos limita a la hora de ofrecer la solución cuando no llegamos a la mejor aproximación, ya que la pila estará vacía en este caso. Para obtener la solución en este caso deberíamos llamar al algoritmo de nuevo con la mejor aproximación.

La otra gran ventaja del algoritmo es que nos permite podar gran cantidad de casos al estar L ordenada. Algunas de las optimizaciones que podemos realizar son:

- Saltar elementos repetidos.
- Evitar repeticiones por conmutatividad y casos no válidos como resultados negativos o divisiones que nunca serán válidas.
- Evitar utilizar unos cuando la operación sea producto o suma.
- Podríamos, utilizando el algoritmo de búsqueda binaria, tratar la división como un caso especial y buscar sólo los múltiplos del número más pequeño, lo que sería útil en tamaños de entrada mayores.

4.5. Implementación

Para la implementación definimos clases `Pila` y `Vector` que se utilizan para manejar P y L respectivamente. Como vimos en la sección anterior, su tamaño está acotado y variará entre 0 y 6 elementos, así que optamos por implementarlos utilizando **vectores estáticos** de 6 elementos.

La clase `Pila` guarda `structs` que representan cada operación con 3 enteros: los operandos y la operación. Implementamos los métodos habituales y el algoritmo de impresión descrito en la primera parte. Dado su reducido tamaño, implementar la pila como un vector estático es ventajoso para simplificar el código y nos permite acceso aleatorio en la parte final.

La clase `Vector` guarda enteros. Consta de un método de inserción y borrado ordenados y de métodos que nos devuelven la primera posición que no tiene un uno y que avanzan hasta el siguiente elemento distinto.

El algoritmo se implementa más fácilmente utilizando bucles `for` que iteran sobre los elementos no repetidos.

5. Combinaciones mágicas

Definición. Una combinación C se dice **mágica** si $[100, 999] \subseteq C^*$.

No conocemos propiedades de las combinaciones mágicas que nos permitan descartar combinaciones sin comprobarlas. Proponemos una refinación de la fuerza bruta, que compruebe todas las combinaciones una a una en busca de las que sean mágicas.

En primer lugar calcularemos el número de combinaciones posibles, que son los conjuntos C posibles a escoger de C_T. Como el número de elementos de C_T es $|C_T| = 14$, el número de elementos de C es $|C| = 6$, y podemos repetir elementos, el número de elementos se corresponderá con las combinaciones con repetición de 14 elementos tomados de 6 en 6:

$$\mathrm{CR}_{14}^{6} = 27\ 132\ \textit{combinaciones posibles}$$

5.1. Versión 1

Podría ocurrírsenos ejecutar 900 veces el algoritmo con cada combinación para ver si encuentra todos los números desde 100 hasta 999. Teniendo en cuenta que el tiempo de ejecución del peor caso (que se da cuando el algoritmo no para antes de que termine el bucle, es decir, cuando el número no se puede obtener exacto) para el algoritmo de menor tiempo de ejecución (el algoritmo 1) en nuestra máquina ronda las tres milésimas, el tiempo de ejecución total estaría acotado superiormente por:

$$0{,}003\ \frac{\textit{segundos}}{\textit{ejecución}} \cdot 27\ 132\ \textit{combinaciones} \cdot 900\ \frac{\textit{ejecuciones}}{\textit{combinación}} = 73\ 256{,}4\ s \approx 20\ \textit{horas}$$

No es un tiempo demasiado elevado teniendo en cuenta que bastaría con obtener las combinaciones mágicas una vez. Sin embargo, es fácil reducirlo modificando ligeramente los algoritmos.

5.2. Versión 2

Hay una forma de reducir el tiempo de ejecución. Dado que cada vez que ejecutamos el algoritmo exploramos los mismos caminos, con la única diferencia de que se parará antes si se encuentra el objetivo, podemos modificar los algoritmos para que no paren hasta el final y vayan marcando un número entre 100 y 999 cada vez que lo encuentren en el proceso (por ejemplo, en una matriz de booleanos). Al final del algoritmo se comprobaría si todo número entre 100 y 999 aparece marcado, y si así es, la combinación es mágica. Esto solo requerirá para cada combinación un tiempo similar (en realidad, ligeramente inferior) al de comprobación de un único objetivo para la combinación que no tenga solución exacta.

5.2.1. Algoritmo de Verificación

Creamos un vector con 900 booleanos (o 1000, si queremos que se marque el número n en la posición n y no calcular $n - 100$ a cada paso) con valor **false**. Ejecutamos el algoritmo modificado, de manera que si un valor generado se encuentra entre 100 y 999 esa posición toma el valor **true**. Si al final de la ejecución todas las componentes desde 100 hasta 999 tienen el valor **true**, la combinación es mágica.

Teniendo en cuenta que de esta manera nos quitamos el factor 900, el tiempo total, usando 0,003 segundos como el tiempo en el peor caso, tenemos que:

$$0{,}003\ \frac{segundos}{ejecución} \cdot 27\ 132\ combinaciones \cdot 1\ \frac{ejecución}{combinación} = 81{,}396\ segundos$$

Parece que merece la pena haberse parado a pensar antes de poner el programa a procesar durante veinte horas. En el algoritmo 1, debido a que se reduce el número de operaciones en el bucle más interno (no hay que calcular la mejor aproximación, lo que evita tener que calcular dos diferencias a cada paso), el tiempo resultante es, en nuestra máquina, aproximadamente 53 segundos. Como muestra, en una ejecución realizada en el momento en que se redactó esto, el tiempo de ejecución ha resultado ser de **52,1 segundos**.

Para recorrer todas las combinaciones emplearemos seis bucles anidados que recorran los catorce números iniciales posibles de menor a mayor. Cada bucle empezará en el valor que toma la variable del bucle en el que está contenido, para considerar solo las combinaciones ordenados. De esta forma, se obtienen las **1242** combinaciones mágicas existentes en orden lexicográfico.

6. Implementación

En la carpeta `Implementaciones` pueden encontrarse implementaciones en C++ de los tres algoritmos y una modificación del primero que obtiene las combinaciones mágicas. Incluimos un `Makefile` para facilitar su compilación. Para compilar solo el algoritmo `n` basta con usar la orden `make Algoritmo[n].out` sin corchetes.

Para la ejecución de cada algoritmo existen dos opciones:

- Pasar **7** argumentos indicando los números iniciales y la solución (en este orden).
- No pasar ninguno (en cuyo caso se generará un ejemplo aleatorio).

Los programas muestran las operaciones hasta alcanzar la mejor aproximación (salvo el segundo, que si no encuentra el resultado exacto solo mostrará el resultado más cercano sin desplegar las operaciones que llevaron al mismo) y el tiempo invertido en hallar la solución.

El programa para obtener las combinaciones mágicas, que se compilará con el nombre `Algoritmo1magic.out`, las devuelve encerradas en llaves y separadas por saltos de línea en salida estándar (puede redirigirse la salida a archivo usando `Algoritmo1magic.out > nombrearchivo`) y adjunta el tiempo de ejecución.