

Reto II: Problema de las cifras

Francisco David Charte Luque Ignacio Cordón Castillo
Mario Román García

1 Tipos de datos abstractos usados.

Se emplea la siguiente notación genérica para la representación de un TDA abstracto:

TDA MiTDA

MiTDA
<ul style="list-style-type: none">- datos privados- métodos privados
<ul style="list-style-type: none">+ datos accesibles a través de la interfaz+ métodos invocables desde la interfaz

- Descripciones sobre el TDA

Los TDA empleados en la resolución del problema de las cifras han sido:

TDA Cuenta

Cuenta
<ul style="list-style-type: none">+ primero+ segundo+ operador+ resultado

- **primero** Número entero, representando el primer operando.
- **segundo** Número entero, que representa el segundo operando.
- **operador** Carácter que corresponde a la operación realizada sobre los números.
- **resultado** Número entero, resultado de realizar la operación sobre **primero** y **segundo**.

TDA ProblemaCifras

ProblemaCifras
<ul style="list-style-type: none"> - numerosIniciales - numeros - operaciones - meta - operacionesPosibles
<ul style="list-style-type: none"> + opera() + resuelve()

- **numerosIniciales** Conjunto que almacena los enteros a partir de los que se pretende obtener **meta**
- **numeros** Lista sobre la que se realizarán todas las operaciones necesarias hasta llegar a una aproximación (o al número buscado exactamente) de **meta**
- **operaciones** Lista de objetos **Cuenta** en los que se almacenarán las operaciones realizadas hasta llegar a **meta**, o a una aproximación a **meta**
- **meta** Entero positivo de 3 cifras a aproximar, y en caso de ser posible, hallar de forma exacta mediante operaciones sobre las cifras dadas iniciales
- **operacionesPosibles** Conjunto que contiene todas las operaciones posibles aplicables {+,*,-,/}
- **opera()** Función que devuelve para dos operandos dados, el resultado de una operación determinada de entre **operacionesPosibles** para ellos
- **resuelve()** Función recursiva que selecciona parejas de cifras de **numeros**, que introduce operados (con **opera()**) en dicha lista, para llamarse a sí misma e intentar llegar a **meta**. Caso de no producir acierto, saca los números introducidos y devuelve los extraídos, y reitera con otra pareja

TDA Números

Números
- contenedorNumeros
+ insertarResultado(resultado) + retirarNumero(posición)

- **contenedorNumeros** Contenedor en el que se almacenarán los números, dependiendo de la implementación, podría ser un vector, una cola, una doble cola o un vector circular.
- **insertarResultado** Inserta un nuevo resultado en el contenedor de números, con el que se podrá operar posteriormente.
- **retirarNumero** Retira un número ya usado, que ya no se podrá usar en siguientes operaciones.

TDA Operación

Operación
+ funcionEnteros(int, int) + esPosible(int, int)

- **funcionEntreDosEnteros** Función $f : \text{int} \times \text{int} \rightarrow \text{int}$, en forma de función anónima lambda que representará una de las operaciones posibles entre dos números del conjunto. En nuestro caso, sumas, restas, multiplicaciones y divisiones.
- **esPosible** Verdadero si se cumplen las condiciones que permiten que la operación produzca un entero positivo.

2 Algoritmos.

2.1 Algoritmo principal.

El algoritmo propuesto para resolver el problema de las cifras es:

Algoritmo 1 ALGORITMO DE CÁLCULO DEL NÚMERO DE 3 CIFRAS

Entrada:

meta, número a aproximar
numeros, enteros aleatorios iniciales del conjunto
size, número de posiciones de la lista **números**

Salida:

true si logramos alcanzar exactamente **meta** o es una de las cifras de **numeros**
false si sólo logramos una aproximación **aprox** a **meta**

- 1: Inicializa **mejor_aprox** a -1.
- 2: **si** Hay al menos dos cifras que seleccionar **entonces**
- 3:
- 4: **para** cada pareja ordenada (**a**,**b**) en **numeros**
- 5:
- 6: **para** cada operación **op** en **[+,*,-,/]**
- 7: **si** **a (op) b** es posible **entonces**
- 8: Computa la **cuenta**
- 9: Almacena la cuenta en la pila de cuentas
- 10: Retira **a**,**b** del conjunto de números
- 11: Introduce **a (op) b** en el conjunto de números
- 12: **si** $|meta - a (op) b| < |meta - mejor_aprox|$ **entonces**
- 13: **mejor_aprox** := **meta**
- 14: **si** **mejor_aprox** == **meta** **entonces**
- 15: **devolver true**
- 16: **fin si**
- 17: **fin si**
- 18: **si** llamamos recursivamente al algoritmo sobre **números** y devuelve **true**
 entonces
- 19: **devolver true**
- 20: **fin si**
- 21: Retira la **cuenta** de la pila de cuentas
- 22: Retira **a (op) b** del conjunto de números
- 23: Reintroduce **a**,**b** en el conjunto de números
- 24: **fin si**
- 25: **fin para**
- 26: **fin para**
- 27: **en otro caso**
- 28: Devolver **false**
- 29: **fin si**

Llamando $T(n)$ a la función que da la eficiencia del algoritmo 1, en función de la longitud del vector **numeros**, esto es, de las cifras dadas para llegar a meta, se tiene: Román

- Desde las líneas 1 a 3, las operaciones realizadas son $\mathcal{O}(1)$
- La selección de parejas (**a**,**b**) de la línea 4 se hace mediante combinaciones $\binom{a}{b}$.
Explícitamente, podemos observar como en la implementación en **C++** adjunta, esto supone:

$$\begin{aligned}\sum_{i=0}^{n-1} (n-i) &= \sum_{i=0}^{n-1} n - \sum_{i=0}^{n-1} i = \frac{2 \cdot n(n-1)}{2} - \frac{(n-1) \cdot (n-1)}{2} = \\ &= \frac{n^2 - 1}{2} \text{operaciones}\end{aligned}$$

- La línea 8 es $\mathcal{O}(1)$
- Las líneas 9 y 10 dependiendo del lenguaje de programación y de la estructura elegida para almacenar las operaciones y el conjunto, podrían ser $\mathcal{O}(1)$ en caso de ser listas, y $\mathcal{O}(n)$ cada una en caso de tratarse de vectores como en **C++**. En la implementación aportada se emplea un **vector** de la **STL** de tamaño fijo, luego la línea 10 se computaría como $\mathcal{O}(1)$. Por simplicidad también consideraremos la línea 9 como $\mathcal{O}(1)$, dado que podría programarse una lista enlazada dotada del operador `[]` necesario para implementar el algoritmo 2 en **C++**.
- Desde las líneas 12 a 17, se trata de operaciones $\mathcal{O}(1)$
- De nuevo sobre las líneas 21, 22, 23 puede decirse lo mismo que sobre las 9,10. Aquí las consideraremos $\mathcal{O}(1)$
- En la línea 18 se llama recursivamente al algoritmo. Por tanto:

$$T(n) = \begin{cases} \frac{n^2 - 1}{2} \cdot T(n-1) & n > 2 \\ 1 & n = 1 \end{cases}$$

y se tiene que:

$$\begin{aligned}T(n) &= \frac{n^2 - 1}{2} \cdot T(n-1) = \frac{n^2 - 1}{2} \cdot \frac{(n-1)^2 - 1}{2} \cdot T(n-2) = \\ &= \dots = T(n-j-1) \cdot \frac{1}{2^{j+1}} \prod_{i=0}^j [(n-i)^2 - 1]\end{aligned}\tag{1}$$

Tomando $j = n - 3$ en (1), se tiene $T(n) = \mathcal{O}\left(\frac{(n!)^2}{2^n}\right)$

2.2 Algoritmo de normalización de operaciones

Puesto que en la lista de operaciones aparecen todas las operaciones necesarias para llegar a **meta** o a una aproximación a la misma, pero también pueden más operaciones que las estrictamente necesarias, se presenta a continuación, otro algoritmo para normalizar dichas operaciones en función del resultado obtenido por el algoritmo 1. Nótese que este algoritmo está destinado a mejorar una solución válida ya obtenida para hacer que se muestre con las mínimas operaciones posibles.

Algoritmo 2 ALGORITMO DE NORMALIZACIÓN DE OPERACIONES

```
1: si hay más de una Cuenta en la lista de operaciones entonces
2:   Llama al siguiente algoritmo, pasándole primero y segundo de la última Cuenta
     efectuada, y como posición de escritura la penúltima de operaciones (podría
     ser -1)
3: fin si
Entrada:
   unaCuenta, última cuenta necesaria en la lista
   pos_escribir, posición anterior a la última normalizada
4: La Cuenta a consultar es la que ocupa pos_escribir
5: mientras No se hallen primero y segundo de unaCuenta como resultado de otra
   Cuenta, y quede alguna por consultar
6:
7:   si el resultado de la Cuenta consultada es primero o segundo entonces
8:     Marcarlo como encontrado
9:     Intercambiar la Cuenta que ocupa la posición pos_escribir en operaciones
     por la Cuenta consultada
10:    Decrementa pos_escribir y llama al algoritmo para la última Cuenta con-
     sultada, y pos_escribir
11:    El índice a consultar es ahora pos_escribir
12:  en otro caso
13:    Decrementa el índice de la posición a consultar
14:  fin si
15: fin mientras
```

Una vez normalizadas las operaciones:

```
1: Se itera operaciones desde el principio hasta el final de la lista
2: imprimir Cuenta actual
```

2.3 Optimización

El algoritmo mostrado puede ser optimizado haciendo que no estudie los casos en los que la pareja ordenada (a,b) no cumpla unos determinados requisitos, y no compute el resultado en los casos en los que éste no aportara nada a la resolución del algoritmo. Las siguientes comprobaciones están diseñadas para aplicarse antes de la línea 8 del algoritmo principal, donde se calcula la operación. En caso de que se cumpla una de estas condiciones, puede pasarse a la siguiente iteración del bucle.

- El resultado $a \text{ (op) } b$ es igual a a o b , por lo que no aporta nada a la resolución.
- El resultado es igual a 0, $a \text{ (op) } b == 0$, pero las operaciones con 0 no aportan nada a la resolución.
- El resultado es negativo, como consecuencia del desbordamiento o de restas no válidas, no debe ser usado.
- Como optimización previa, los números están ordenados. Se previene la duplicación de parejas y de casos.
- Como optimización previa, se descartan divisiones no enteras o restas negativas, que no pueden usarse en la resolución.

2.4 Algoritmo alternativo

Se propone otro algoritmo destinado a dar una orientación alternativa a la solución del problema. El algoritmo principal se centra en buscar una solución siguiendo un sólo camino cada vez, y ahondando en la recursividad antes de volver a buscar por otro camino. Este algoritmo buscar avanzar de forma uniforme por todos los caminos posibles hacia la solución.

3 Implementación

Mostramos ahora cómo podría implementarse el algoritmo propuesto, aplicando sobre él las optimizaciones sugeridas. Con esta implementación hemos querido comprobar el correcto funcionamiento y eficiencia del algoritmo. Para la mayoría de los casos, el tiempo requerido no excede los pocos segundos, estando el máximo en torno a los 20 segundos.

```
001: #include "cifras.h"
002: using namespace std;
003: typedef int (*Operacion)(int a, int b);
004:
005:
006: Cifras::Cifras (vector<int> introducidos) {
007:     #ifndef GRUPOS
008:         // Primera aproximación
009:         mejor = -1;
010:     #endif
011:
012:     #ifdef GRUPOS
013:         // Números marcados
014:         total_encontrados = 0;
015:         vector<bool> encontrado_inicial(BUSCADOS);
016:         for (int i=0; i<BUSCADOS; ++i)
017:             encontrado_inicial[i] = false;
018:         encontrado = encontrado_inicial;
019:
020:         // El cero se marca por defecto
021:         marcar(0);
022:     #endif
023:
024:     // Introduce los números en la doble cola.
025:     vector<int> numeros;
026:     int size = introducidos.size();
027:     for (int i=0; i<size; ++i)
028:         numeros.push_back(introducidos[i]);
029:
030:     this->numeros = numeros;
031: }
032:
033: bool Cifras::resuelve (int meta) {
034:     // Empieza comprobando que el número buscado no esté entre los dados.
035:     for (vector<int>::iterator it=numeros.begin(); it != numeros.end(); ++it){
036:         #ifndef GRUPOS
037:             if (*it == meta) {
038:                 Cuenta encontrada = {meta, meta, ' '};
039:                 mejor_operaciones.push_back(encontrada);
040:             }
```

```

041:         return true;
042:     }
043: #endif
044:
045: #ifdef GRUPOS
046:     marcar(*it);
047: #endif
048: }
049:
050: // Resuelve de forma recursiva todas las posibilidades.
051: bool resuelto=resuelve_rec(meta, numeros.size());
052:
053: #ifndef GRUPOS
054:     normalizaOperaciones();
055: #endif
056:
057:     return resuelto;
058: }
059:
060: bool Cifras::resuelve_rec (int meta, int size) {
061:     // Operaciones
062:     Operacion calcula[] = {
063:         [](int a, int b){ return a-b; },
064:         [](int a, int b){ return a/b; },
065:         [](int a, int b){ return a+b; },
066:         [](int a, int b){ return a*b; }
067:     };
068:
069:     #ifndef GRUPOS
070:     Cuenta opActual;
071:     #endif
072:
073:     if (size < 2) return false;
074:
075:     // Toma el primer número disponible
076:     for (int i=0; i<size-1; ++i) {
077:         int a = numeros[i];
078:
079:         if (!(a==0))
080:             numeros[i]=numeros[size-1];
081:         else
082:             continue;
083:
084:         // Toma el segundo número disponible
085:         for (int j=i; j<size-1; ++j) {
086:             int b = numeros[j];
087:             if (b != 0)
088:                 numeros[j] = numeros[size-2];
089:             else

```

```

090:         continue;
091:
092:         // Y prueba sobre ellos todas las operaciones
093:         for (int op=0; op<NOP; ++op) {
094:             // Cogemos siempre c como el mayor de ambos
095:             int c=(a>b?a:b), d=(c==a?b:a);
096:
097:             // Comprueba que la operación sea válida
098:             bool indivisible = ((c%d != 0) and op==DIV);
099:             if (indivisible)
100:                 continue;
101:
102:             // Comprueba que la operación sea útil
103:             int resultado = calcula[op](c,d);
104:             bool trivial = (resultado == a or resultado == b);
105:             bool zero = (resultado == 0);
106:             bool overflow = (resultado < 0);
107:             if (trivial or overflow or zero)
108:                 continue;
109:
110:             // Calcula y guarda la operación.
111:             #ifndef GRUPOS
112:             opActual = {c, d, SIMBOLOS[op], resultado};
113:             operaciones.push_back(opActual);
114:
115:             // Intenta resolver o mejorar con el nuevo número, sin pasarse
116:             if (abs(meta-resultado) < abs(meta-mejor)) {
117:                 mejor = resultado;
118:                 mejor_operaciones = operaciones;
119:
120:                 if (resultado == meta)
121:                     return true;
122:             }
123:             #endif
124:
125:             // Marca el nuevo resultado y comprueba si están todos marcados
126:             #ifdef GRUPOS
127:             if (marcar(resultado))
128:                 return true;
129:             #endif
130:
131:             // Guarda el nuevo resultado y sigue buscando
132:             numeros[size-2] = resultado;
133:             if (resuelve_rec(meta,size-1))
134:                 return true;
135:
136:             #ifndef GRUPOS
137:             //Saca las operaciones
138:             operaciones.pop_back();

```

```

139:         #endif
140:     }
141:     numeros[size-2]=numeros[j];
142:     numeros[j]=b;
143: }
144:     numeros[i]=a;
145: }
146:     return false;
147: }
148:
149: #ifndef GRUPOS
150: void Cifras::escribeOperaciones() {
151:     vector<Cuenta>::iterator it;
152:
153:     for(it=mejor_operaciones.begin(); it!=mejor_operaciones.end(); it++){
154:         cout << *it << endl;
155:     }
156: }
157: #endif
158:
159: #ifdef GRUPOS
160: bool Cifras::marcar(int n) {
161:     if (n<1000 and !encontrado[n]) {
162:         encontrado[n] = true;
163:         --total_encontrados;
164:         if (total_encontrados == BUSCADOS)
165:             return true;
166:     }
167:
168:     return false;
169: }
170: #endif
171:
172: #ifdef GRUPOS
173: void Cifras::imprime_restantes () {
174:     for (int i=0; i<BUSCADOS; ++i)
175:         if (!encontrado[i])
176:             cout << i << ' ';
177:     cout << endl;
178: }
179: #endif
180:
181: #ifdef GRUPOS
182: bool Cifras::todos_marcados () {
183:     bool todos = true;
184:     for (int i=0; i<BUSCADOS and todos; ++i)
185:         todos = encontrado[i];
186:     return todos;
187: }

```

```

188: #endif
189:
190: #ifndef GRUPOS
191: void Cifras::normalizaOperaciones() {
192:     int size = mejor_operaciones.size();
193:     int pos_escribir = size - 2;
194:
195:     if (pos_escribir >= 0){
196:         buscaOperandos (mejor_operaciones[size-1],pos_escribir);
197:
198:         mejor_operaciones.erase(mejor_operaciones.begin(),
199:             mejor_operaciones.begin() + pos_escribir + 1);
200:     }
201: }
202:
203: void Cifras::buscaOperandos(Cuenta unaCuenta, int& pos_escribir){
204:     bool uno_encontrado=false, otro_encontrado=false;
205:     int j = pos_escribir;
206:     int un_operando = unaCuenta.primeroperando,
207:         otro_operando = unaCuenta.segundooperando;
208:
209:     while ((!uno_encontrado || !otro_encontrado) && j>=0){
210:         if ((mejor_operaciones[j].resultado == un_operando) ||
211:             (mejor_operaciones[j].resultado == otro_operando)){
212:
213:             if (uno_encontrado)
214:                 otro_encontrado=true;
215:             else
216:                 uno_encontrado=true;
217:
218:             Cuenta aux(mejor_operaciones[j]);
219:             mejor_operaciones[j]=mejor_operaciones[pos_escribir];
220:             mejor_operaciones[pos_escribir]=aux;
221:             pos_escribir--;
222:
223:             buscaOperandos(mejor_operaciones[pos_escribir+1], pos_escribir);
224:             j = pos_escribir;
225:         }
226:         else
227:             j--;
228:     }
229: }
230:
231:
232: std::ostream& operator<<(std::ostream& salida, const Cifras::Cuenta& operacion){
233:     if (operacion.operador != ' ')
234:         salida << operacion.primeroperando << operacion.operador <<
235:             operacion.segundooperando << '=' << operacion.resultado;
236:     else

```

```
237:         salida << operacion.primeros;
238:
239:         return salida;
240:     }
241: #endif
242:
```