Aufgabe 2: Rechenrätsel

Teilnahme-Id: 60660

Bearbeiter/-in dieser Aufgabe:
Raphael Gaedtke

19. April 2022

Inhaltsverzeichnis

1 Lösungsidee

Definition 1. *Ein zu erzeugendes Rechenrätsel habe n Operatoren (Die Zahl n wird durch den Nutzer vorgegeben).*
Die Operatoren werden von $\circ_1$ bis $\circ_n$ und die Operanden von z_0 bis z_n durchnummeriert. Die Operanden sind dabei allesamt einzelne, positive Ziffern zwischen 1 und 9. Das Ergebnis sei die positive ganze Zahl e. Das Rechenrätsel hat dann folgende Form:

$$z_0 \circ_1 z_1 \circ_2 ... \circ_{n-1} z_{n-1} \circ_n z_n = e \tag{1}$$

Um möglichst interessante und unterschiedliche Rechenrätsel zu erstellen, werden die Ziffern $z_0, z_1, ..., z_n$ zu Beginn mithilfe eines Zufallsgenerators gewählt. Basierend auf dieser Auswahl von Zahlen wird im weiteren Verlauf eine Zahl e ermittelt, für die das Rechenrätsel eindeutig lösbar ist.

1.1 Berechnung möglicher Ergebnisse

Um eine passende Zahl e zu wählen, werden in $(n + 1)$ Schritten alle möglichen Werte von e bestimmt. Dabei wird im i-ten Schritt berechnet, welche Ergebnisse sich mit den ersten i Zahlen und den $(i - 1)$ zwischen diesen Zahlen liegenden Operatoren $\circ_1, \circ_2, ..., \circ_i$ erzeugen lassen, wenn diese Operatoren beliebige, mit Regel f) vereinbare Werte annehmen.
Nach $(n + 1)$ Schritten liegen dann alle Ergebnisse vor, die sich mit allen Zahlen und Operatoren erreichen lassen. Aus diesen Ergebnissen wird dann wiederum mithilfe eines Zufallsgenerators die Zahl e ausgesucht.

Die Ergebnisse, die sich mit den ersten i Zahlen erzeugen lassen, werden dabei in der Menge M_i gespeichert, die möglichen Ergebnisse des gesamten Rechenrätsels also in der Menge M_{n+1}. Es gilt zudem

$$M_1 = \{z_0\}, \tag{2}$$

weil sich mithilfe der Zahl z_0 ohne Operatoren bzw. Verknüpfungen mit anderen Zahlen kein anderes Ergebnis erreichen lässt.
Um M_i für ein i mit $1 < i \leq (n+1)$ zu berechnen, wird die durch Regel d) vorgegebene Punkt- vor Strich-Rechnung angewandt: Ist in einer Lösung für eine Zahl k in M_i der Operator o_j mit $j \leq i$ der letzte Operator vor z_i, der entweder ein $+$ oder ein $-$ ist, so ist k die Summe oder Differenz eines Elements aus M_j und Zahlen $z_j, z_{j+1}, ..., z_i$, die nur durch $*$ oder : miteinander verknüpft sind.
Um diese Überlegung auszunutzen, werden im i-ten Schritt noch die Mengen $S_0, S_1, ..., S_i$ gespeichert. Die Menge S_j für $0 \leq j$ soll dann die Menge aller Zahlen speichern, die sich mit den Ziffern $z_j, z_{j+1}, ..., z_i$ erzeugen lassen, wenn die zwischen diesen Zahlen liegenden Operatoren alle entweder * oder : sind.
Außerdem muss noch berücksichtigt werden, dass es Elemente $k \in M_i$ geben kann, deren Lösung ausschließlich Multiplikation und Division erfordern. Dies lässt sich ausdrücken durch die Aussage

$$S_0 \subset M_i. \tag{3}$$

Für jedes i ergibt sich daraus die rekursive Vorschrift

$$M_i = S_0 \cup \left(\bigcup_{j=0}^{i} \{(m+s), (m-s) \mid \forall m \in M_j, s \in S_j\} \right). \tag{4}$$

Diese Vorschrift lässt sich algorithmisch umsetzen, indem über i, für jedes i über j und für jedes j über alle möglichen Kombinationen von Elementen aus M_j und S_j iteriert wird.
Die Mengen $S_0, S_1, ..., S_i$ werden im $(i+1)$-ten Schritt ebenfalls aus diesen Mengen für den i-ten Schritt abgeleitet. Dazu wird in jeder dieser Mengen jedes Element mit z_i multipliziert und -insofern dies unter Beachtung von Regel f) möglich ist- durch z_i geteilt. Die Ergebnisse dieser Rechnungen bilden die aktualisierte Menge. Außerdem ist immer

$$S_i = \{z_i\}. \tag{5}$$

Dabei muss angemerkt werden, dass die Mengen M_i und S_i keine konstanten Mengen im mathematischen Sinne sind. In der vorliegenden Dokumentation wird mit einer „Menge“ vielmehr eine zustandsabhängige Datenstruktur bezeichnet, die so wie eine Menge Elemente enthält. In dieser Datenstruktur können dann neue Elemente hinzugefügt und enthaltene gelöscht werden.

Beispiel 1. *Die Berechnung eines eindeutigen Ergebnisses e wird am in der Aufgabenstellung gegebenen Beispiel* $4 \circ 4 \circ 3 = 13$ *nachvollzogen. Hier wäre also* $n = 2$ *und die Verwendung eines Zufallsgenerators hätte die Werte* $z_0 = 4$, $z_1 = 4$ *und* $z_2 = 3$ *ergeben.*

1. *Zunächst ist* $S_0 = M_1 = \{z_0\} = \{4\}$.
2. *Im nächsten Schritt ist* $S_1 = \{z_1\} = \{4\}$ *und die Menge* S_0 *wird aktualisiert zu* $S_0 = \{1, 16\}$, *weil* $4 * z_1 = 4 * 4 = 16$ *und* $4 : z_1 = 4 : 4 = 1$ *gilt. Daraus ergibt sich* $M_2 = \{0, 1, 8, 16\}$.
3. *Im letzten Schritt ist* $S_2 = \{3\}$. *Die Menge* S_0 *und* S_1 *werden aktualisiert zu* $S_0 = \{3, 48\}$ *und* $S_1 = \{12\}$. *Es ergibt sich* $M_3 = \{-3, -2, 3, 4, 5, 11, 13, 16, 19, 48\}$. *Aus den positiven, eindeutigen Einträgen dieser Menge wird per Zufall die Zahl 13 ausgewählt und es ergibt sich das obige Rechenrätsel.*

Im Folgenden wird beschrieben, wie mithilfe dieses Verfahrens auch die Eindeutigkeit eines Ergebnisses und die Lösungen der resultierenden Rechenrätsel bestimmt werden. Außerdem werden Laufzeitverbesserungen durch die Verwendungen von Hashtabellen für das Speichern alle Mengen M_i und S_i und das Ausschließen bestimmter Ergebnisse beschrieben.

1.2 Eindeutigkeit möglicher Ergebnisse

Um die Eindeutigkeit eines Ergebnisses festzuhalten, wird für jedes Element der Mengen M_j und S_j eine boolesche Variable gespeichert, die genau dann wahr ist, wenn dieses Element eindeutig ist. Eindeutig bedeutet hier, dass die entsprechende Zahl mit genau einer Auswahl von Operatoren erreicht werden kann.

Dieser Wert ist immer wahr für das einzige Element von $M_1 = \{z_0\}$, da dieses nicht auf mehrere Arten dargestellt werden kann. Außerdem ist dieser Wert für die Zahl z_i, die im $(i+1)$-ten Schritt zur bis dahin leeren Menge S_{i+1} hinzugefügt wird, aus dem gleichen Grund immer wahr. Beim Aktualisieren einer Menge S_j kann der korrespondierende bool-Wert einer Zahl auf False gesetzt werden, wenn sie beim Aktualisieren mehrmals erreicht wird.
Es gibt dann verschiedene Fälle, in denen ein Element einer Menge M_j nicht eindeutig ist und in denen die korrespondierende Variable somit auf False gesetzt wird:

1. Ein Element einer Menge M_j, das als Summe bzw. Differenz eines Elements aus einer anderen Menge M_k und der Menge S_k darstellbar ist, ist dann nicht eindeutig, wenn das entsprechende Element aus M_k nicht eindeutig ist.

2. Eine solche Summe bzw. Differenz ist auch dann nicht eindeutig, wenn das entsprechende Element aus S_k nicht eindeutig ist.

3. Ein Element einer Menge M_j ist dann nicht eindeutig, wenn es auf zwei verschiedene Arten als Summe bzw. Differenz von Elementen von Mengen M_k und S_k dargestellt werden kann.

Die ersten beiden Bedingungen können direkt beim Auffinden eines neuen Elements einer Menge M_j überprüft werden. Um die dritte Bedingung zu überprüfen, wird beim Auffinden eines solchen Elements festgestellt, ob die entsprechende Zahl schon in M_j enthalten ist.
In diesen drei Fällen wird der entsprechende boolesche Wert auf False gesetzt. In allen anderen Fällen ist die Zahl in der entsprechenden Menge eindeutig dargestellt und der entsprechende boolesche Wert deswegen True.

1.3 Lösung resultierender Rechenrätsel

Die Lösung des Rechenrätsels für spezifische Ergebnisse wird auf eine der Bestimmung der Eindeutigkeit ähnliche Art und Weise ermittelt.
Dafür wird für jedes Element einer Menge M_i oder S_i eine Zeichenkette gespeichert. Eine Zeichenkette für die Menge M_i enthält $(i-1)$ Zeichen, von denen jedes einem der Symbole $+$, $-$, $*$ oder : entspricht. Das k-te dieser Zeichen speichert den für das Erreichen der aktuellen Zahl als Ergebnis nötigen Zustand des k-ten Operators.
Analog enthält eine Zeichenkette für ein Element einer Menge S_i nur die Zeichen $*$ und :, die den notwendigen Zustand der entsprechenden Operatoren beschreiben. Wird eine einzelne Ziffer der Menge M_1 oder einer Menge S_i hinzugefügt, ist die dazugehörige Zeichenkette zunächst leer.
Wird ein Element einer Menge S_i aktualisiert, wird an die dazugehörigen Zeichenkette entweder ein $*$ oder ein : angehängt. Wird eine Zahl $m \pm s$ einer Menge M_i hinzugefügt, wird an die Zeichenkette für m ein $+$ oder ein $-$ angehängt und an das Ergebnis dieser Konkatenation die Zeichenkette für s.
Kann ein Element einer Menge M_i oder S_i mit mehreren Verknüpfungen von Operatoren erreicht werden, enthält die dazugehörige Zeichenkette eine beliebige dieser Möglichkeiten. Dies ist unschädlich, weil diese Zeichenketten ohnehin nicht ausgelesen werden: Nach dem Berechnen und Auswählen eines eindeutigen Ergebnisses e wird nur dessen zugehörige Zeichenkette verwendet, um die Lösung des Rechenrätsels zu rekonstruieren.

1.4 Existenz eines eindeutigen Ergebnisses

Die Zahlen $z_0, z_1, ..., z_n$ müssen zu Beginn so gewählt werden, dass immer ein eindeutiges Ergebnis vorliegt, dass am Ende als die Zahl e ausgewählt werden kann. Es wird gezeigt, dass es immer eine solche Zahl e gibt, wenn keine der Ziffern eine 1 ist und auch keine 2 auf eine 2 folgt:

Satz 1. ***Es sei eine Folge von Ziffern*** $z_0, z_1, ..., z_n$ ***so gewählt, dass keine dieser Ziffern eine 1 ist und auch keine 2 auf eine 2 folgt. Dann ist das Produkt***

$$p = \prod_{i=0}^{n} z_i \tag{6}$$

durch keine andere Wahl von Operatoren mit diesen Ziffern darstellbar. Es wird zudem gezeigt, dass p ***das maximale Ergebnis des Rechenrätsels ist.***
Beweis: Die Behauptung wird durch Widerspruch bewiesen. Es gebe also eine Auswahl von Operatoren $\circ_0, \circ_1, ..., \circ_n$*, für die es ein* i *gibt mit* $\circ_i \neq *$ *und für die die Verknüpfung der Ziffern mit diesen Operatoren*

das Ergebnis p hat. Es wird dann gezeigt, dass die Zahl $b = z_0 \circ_1 z_1 \circ_2 ... \circ_{n-1} z_{n-1} \circ_n z_n$ *kleiner als p ist, was einen Widerspruch zur Annahme* $b = p$ *ergibt.*
Zunächst wird in der neuen Auswahl von Operatoren jedes : *und jedes* $-$ *durch ein* $*$ *ersetzt. Der Wert von b bleibt bei diesen Änderungen gleich oder wird größer.*
Danach gibt es unter den neu ausgewählten Operatoren, mit denen die Zahl b erzeugt wird, nur noch $+$ *und* $*$*. Es kann dann nicht* $b > p$ *gelten, weil keine der Ziffern eine 1 ist. Das Produkt zweier Zahlen, die größer als 1 sind, ist aber immer mindestens genauso groß wie deren Summe. Somit wird b nicht kleiner, wenn nacheinander alle Operatoren auf* $*$ *gesetzt werden.*
Es muss also nur noch der Fall $b = p$ *ausgeschlossen werden. In diesem Fall sei ein Operator* $\circ_k = +$ *und der Wert aller Ziffern auf der linken Seite dieses Operators a und der Wert aller Ziffern und Operatoren auf der rechten Seide d. Dann gilt*

$$p = b = a + d, \tag{7}$$

nach der Definition von p aber auch

$$p = ad \tag{8}$$

und somit

$$a + d = ad. \tag{9}$$

Daraus folgt $a \mid (a + d)$ *und somit* $a \mid d$ *bzw.* $d \mid (a + d)$ *und* $d \mid a$*.* $a \mid d$ *und* $d \mid a$ *gilt aber nur für* $a = d$*. Dann ist also* $(a + d) = 2a$ *und* $ad = a^2$*, was nach Gleichung 9 insgesamt*

$$a^2 = 2a \tag{10}$$

ergibt. Die einzige Lösung dieser Gleichung ist $a = 2$*, woraus nach Gleichung 9 aber auch* $d = 2$ *bedeutet. Es kann aber nicht* $a = d = 2$ *sein, da keine zwei aufeinanderfolgenden Ziffern 2 sein dürfen und somit* $a > 2$ *oder* $d > 2$ *gilt. a und d sind nämlich aufgebaut aus je mindestens einer Ziffer, die je mindestens die den Wert 2 haben. Es kann also nur* $a = 2$ *gelten, wenn a aus nur einer Ziffer mit dem Wert 2 berechnet wird.*
Damit ist p nur darstellbar, wenn jeder Operator ein $*$ *ist. Diese Darstellung ist eindeutig.*

□

1.5 Laufzeitverbesserung durch Hashtabellen

Um die beschriebene Lösungsidee umzusetzen, muss für gegebene Zahlen immer wieder abgefragt werden, ob diese in einer Menge M_i oder einer Menge S_i enthalten sind. Werden diese Mengen in einem einfachen Container gespeichert, benötigt eine solche Abfrage im schlechtesten Fall lineare und das Hinzufügen eines neuen Elementes am Ende dieses Containers konstante Zeit. In anderen Datenstrukturen wie etwa binären Suchbäumen lässt sich die Komplexität beider Vorgänge bis hin zu logarithmischer Laufzeit verringern.
Durch die Verwendung von Hashtabellen bzw. Hashmaps kann logarithmische Laufzeit allerdings noch unterschritten werden. Bei diesen handelt es sich um Datenstrukturen, die das Einfügen und Suchen von Elementen durch die Verwendung einer Hashfunktion in konstanter Zeit erlauben (Im schlechtesten Fall ist diese Laufzeit tatsächlich ebenfalls linear. In der vorliegenden Dokumentation und sämtlichen darin enthaltenen Laufzeitüberlegungen wird diese allerdings als konstant betrachtet, da dies beispielsweise auch in den Lösungshinweisen für die 2. Runde des 38. BwInf[1] der Fall ist).
Das Programm muss im Laufe der Berechnung allerdings auch über die Elemente von Mengen M_i und S_i iterieren können, was Hashtabellen nur in mehr als linearer Zeit unterstützen. Deswegen wird der Speicherbedarf erhöht, um die Laufzeit zu verbessern:
Die Einträge solcher Mengen werden doppelt gespeichert: In einem dynamischen Array werden alle Elemente gespeichert, um zeitsparend über diese iterieren zu können, während eine Hashtabelle die gleichen Elemente enthält. Mit dieser Hashtabelle wird dann ermittelt, ob eine Zahl schon in der Menge enthalten ist.
Auch die bool-Werte und Zeichenketten für die Eindeutigkeit und Operatorwerte einer Zahl werden in Hashmaps gespeichert, die eine solche Zahl diesen Werten zuordnet.

1.6 Laufzeitverbesserung durch Ausschluss von Ergebnissen

In den Mengen M_i werden während der Ausführung der Lösungsidee auch negative Ziffern zwischengespeichert, da diese als Zwischenergebnisse laut Aufgabenstellung nicht verboten sind. Dabei ist es allerdings

[1] https://bwinf.de/fileadmin/bundeswettbewerb/38/loesungshinweise382.pdf, Seite 37, zuletzt abgerufen am 21.03.2022

nicht sinnvoll, alle diese Zahlen zu speichern, da für die Ausgabe $e \geq 1$ gelten muss.
Deswegen muss eine negative Zahl nur dann gespeichert werden, wenn die Summe dieser Zahl und der maximale Wert einer Kombination der auf dieses Zwischenergebnis folgenden Ziffern (also das maximale Ergebnis e mit dieser Zahl als Zwischenergebnis) mindestens 1 ist.
Die maximale Kombination der auf dieses Zwischenergebnis folgenden Ziffern ist nach Satz 1 aber das Produkt dieser Ziffern. Also werden negative Zahlen in einer Menge M_i nur dann gespeichert und weiterverwendet, wenn der Betrag dieser Zahlen kleiner ist als das Produkt aller Ziffern, die auf dieses Zwischenergebnis folgen. Auf ähnliche Weise kann das Ergebnis e nach oben beschränkt werden:

Definition 2. *Es sei o eine vom Nutzer vorgegebene Zahl mit $o \geq 1$. Das Programm soll dann, insofern dies möglich ist, für die Zahl e eine Zahl auswählen, die eindeutig als Ergebnis des Rechenrätsels darstellbar ist und für die $1 \leq e \leq o$ gilt. o ist also eine obere Schranke für e. In dieser Dokumentation und insbesondere allen Quelltexten zu dieser Aufgabe wird o auch als „Obergrenze“ bezeichnet.*

Bei diesem Verfahren muss allerdings auch sichergestellt werden, dass es noch eine eindeutige Lösung für e gibt. Deswegen wird zusätzlich zu den für o erlaubten Zwischenergebnissen auch noch das Produkt p aller Ziffern gespeichert, das nach Satz 1 eindeutig darstellbar ist.
p wird allerdings nur dann ausgegeben, wenn es entweder kleiner als o ist und zufällig gewählt wird oder es keine eindeutigen Ergebnisse gibt, die kleiner als o sind. Ist e nämlich das Produkt aller Ziffern, wird das Rechenrätsel insgesamt weniger spannend.

1.7 Laufzeitüberlegungen

Bei der Laufzeitanalyse der oben beschriebenen Lösungsidee müssen verschiedene Operationen berücksichtigt werden:

- Das Einfügen in die Mengen M_i und S_i
- Das Einfügen in die Container, die die bool-Werte und Zeichenketten für die Eindeutigkeit und Lösung der Elemente dieser Mengen speichern
- Die Abfrage, ob eine Menge M_i oder S_i eine bestimmte Zahl enthält
- Das Auswählen eines eindeutigen Elements aus der Menge M_{n+1} als Ergebnis e, wofür alle eindeutigen Elemente aus dieser Menge bestimmt bzw. alle nicht eindeutigen gelöscht werden müssen

Die Ein- und Ausgabe und das zufällige Auswählen der Ziffern z_0, z_1, ..., z_{n+1} werden bei der Laufzeitanalyse vernachlässigt. Gleiches gilt für die anfängliche Initialisierung der Datenstrukturen für die Mengen M_i und S_i.
Es wird davon ausgegangen, dass die oben genannten Operationen auf den Mengen M_i uns S_i, also das Einfügen, Löschen und Abfragen von Elementen, in konstanter Zeit laufen. Dies ist möglich aufgrund der in Kapitel 1.5 dargestellten Verwendung von Hashtabellen. Zur Bestimmung der Laufzeit muss also abgeschätzt werden, wie oft diese Operationen in Abhängigkeit von n und o ausgeführt werden.
Dazu soll zunächst angenähert werden, wie viele Elemente die Mengen S_i am Ende des Verfahrens enthalten. Eine obere Schranke dafür ist die Anzahl von willkürlichen Kombinationen der Operatoren $*$ und :. Für k Operatoren ergeben sich so 2^k Möglichkeiten, insgesamt haben die Mengen S_i also $2^1 + 2^2 + ... + 2^n \approx 2^{n+1}$ Elemente.
Jedes dieser Elemente wird in mindestens einem der $(n + 1)$ Schritte des Verfahrens einmal eingefügt, wofür sein Wert einmal abgefragt wird. Nach Kürzen aller konstanten Faktoren ergibt sich daraus allein für das Berechnen und Aktualisieren der Mengen S_i eine Laufzeit von $O(n \cdot 2^n)$.
Um auch die Mengen M_i zu berechnen, werden Kombinationen von Elementen aus Mengen M_i und S_i berechnet. In einer Menge M_i können sich dabei nur Elemente befinden, die mit den folgenden Zahlen noch in ein Ergebnis e zwischen 1 und e resultieren können. Die Anzahl dieser Zahlen hängt proportional von n und o ab und kann deshalb durch no angenähert werden.
Das Durchgehen der Kombinationen geschieht wiederum in $(n + 1)$ Schritten, wobei jede dieser Kombinationen abgefragt und eventuell eingefügt werden muss. Für die Laufzeit des Berechnens aller Mengen M_i ergibt sich als grobe Abschätzung somit $O(on^2 \cdot 2^n)$. Dieser Wert ist das Produkt aus der Anzahl der Elementen der Mengen S_i, der Anzahl an Schritten n und der Größe der Mengen M_i.
Abschließend muss noch einmal die Menge M_{n+1} durchgegangen werden, um alle nicht eindeutigen Elemente zu löschen. Unter der Annahme des Löschens von Elementen in konstanter Zeit ergibt sich dafür eine Laufzeit von o.

Die Laufzeit des Verfahrens ist im schlechtesten Fall als Summe aller Teilschritte $O(n \cdot 2^n + on^2 \cdot 2^n + o)$ bzw. $O(on^2 \cdot 2^n + o)$, was allerdings eine sehr grobe Abschätzung ist - so führt beispielsweise die Anwendung von Regel f) zu starken Laufzeitverbesserungen. Die vorliegenden Überlegungen haben hauptsächlich den Nutzen, zu zeigen, dass ein Rechenrätsel in exponentieller Laufzeit erzeugt wird. Für kleine Werte von n liegt die Laufzeit trotzdem in einem akzeptablen Rahmen.

1.8 Überlegungen zu Komplexität und Speicherbedarf

Der Speicherbedarf des Programms ist proportional zur Größe der Mengen M_i und S_i und deswegen nach Kapitel 1.7 exponentiell. Da die Laufzeit des Verfahrens insgesamt aber von der Anzahl der Kombinationen von Elementen je einer Menge M_i und S_i abhängt, wächst der Speicherbedarf mit wachsender Eingabegröße weniger schnell als die Laufzeit des Verfahrens.
Das angegebene Lösungsverfahren kann das Problem nicht in Polynomialzeit lösen. Da für die Erzeugung eines eindeutigen Rechenrätsels beispielsweise alle Elemente der Mengen S_i berücksichtigt werden müssen, lässt sich vermuten, dass das Entscheidungsproblem, ob ein Rechenrätsel eindeutig ist, nicht in Polynomialzeit lösbar bzw. NP-schwer ist.

2 Umsetzung

Die Lösungsidee wird in C++ umgesetzt.
Programmintern werden zu Beginn Container initialisiert, in denen das Rechenrätsel und die Zwischenergebnisse der Berechnungen zwischengespeichert werden. Das Rechenrätsel selbst wird als Container vom Typ `vector<long long>` dargestellt, der die Ziffern auf der linken Seite des Rechenrätsels enthält und direkt nach dem Einlesen der Eingabe mithilfe eines Zufallsgenerators in einer for-Schleife initialisiert wird.
Die Mengen M_i und S_i werden in Containern des Typs `vector<vector<pair<long long, bool>>>` abgelegt - ein solcher Container enthält alle Mengen M_i bzw. alle Mengen S_i. Eine einzelne Menge M_i bzw. S_i wird repräsentiert durch einen Container von Variablen, die aus jeweils einem Integer und einem bool-Wert bestehen, wobei der erste Teil die Zahl selbst und der zweite die Eindeutigkeit dieser Zahl speichert. Diese Container heißen `m` für die Mengen M_i und `s` für die Mengen S_i.
Um zu überprüfen, ob eine Zahl bereits Element einer solchen Menge ist, werden alle vorhandenen Elemente durch Hashmaps auf ihre Indizes im entsprechenden Container abgebildet. Diese Zuordnung wird in Containern des Typs `vector<unordered_map<long long, long long>>` gespeichert. Analog wird durch Container des Typs `vector<unordered_map<long long, string>>` jede Zahl auf ihre als Zeichenkette abgelegte Lösung abgebildet.
Diese Hashmaps heißen für M_i bei der Position der Zahl `mPos` und bei der Lösung `mLoes`, für die Mengen S_i sind diese Bezeichnungen `sPos` und `sLoes`.

In einer `for`-Schleife werden dann nach und nach alle Mengen M_i intialisiert. In einem solchen Schritt werden dabei zuerst in zwei weiteren, ineinander verschachtelten `for`-Schleifen die Elemente der Mengen S_i durch Multiplizieren und Dividieren mit der neuen Ziffer aktualisiert.
Zu Beginn der Programmausführung werden in einem Container `postProdukt` zudem $(n+1)$ Werte gespeichert, von denen der k-te das Produkt aller Ziffern, die auf z_k folgen, einliest. Dieser Container wird verwendet, um zu große oder zu kleine Zahlen auszuschließen.
Auf diese Berechnungen folgen drei ineinander verschachtelte `for`-Schleifen, in denen eine Menge M_i initialisiert wird. Dazu werden in der äußersten dieser `for`-Schleifen alle Mengen M_j mit $j < i$ durchgegangen. In den beiden inneren Schleifen werden alle Kombinationen von Elementen in M_j und S_j durchgegangen und die Container `m`, `mPos` und `mLoes` aktualisiert.
Durch diese drei `for`-Schleifen wird gewissermaßen die in Gleichung 4 beschriebene Rekursionsvorschrift für die Mengen M_i umgesetzt.
Nach dem Berechnen aller Mengen M_i werden aus dem letzten Container M_{n+1} alle nicht eindeutigen Elemente gelöscht. Hat dieser Container nach dem Löschen all dieser Elemente immer noch mindestens zwei Einträge, wird das Produkt aller Ziffern des Rechenrätsels gelöscht, wenn dieses über der Obergrenze o liegt.
Aus dem verbleibenden Container wird per Zufallsgenerator ein Ergebnis ausgewählt und mitsamt der die Lösung beschreibenden Zeichenkette an die Ausgabefunktion übergeben.

3 Beispiele

Die Ausgabedatei des Programms enthält genau zwei Zeilen, wobei in der ersten Zeile das Rechenrätsel selbst und in der zweiten dessen Lösung steht. Die Leerstellen, in die Gwendoline Operatoren einfügen muss, werden durch Unterstriche dargestellt.

Rechenraetsel mit 1 Operator: 6 _ 3 = 3
Loesung dieses Rechenraetsels: 6 - 3 = 3

Rechenraetsel mit 2 Operatoren: 2 _ 5 _ 2 = 8
Loesung dieses Rechenraetsels: 2 * 5 - 2 = 8

Rechenraetsel mit 3 Operatoren: 3 _ 8 _ 4 _ 4 = 19
Loesung dieses Rechenraetsels: 3 + 8 + 4 + 4 = 19

Rechenraetsel mit 4 Operatoren: 8 _ 5 _ 4 _ 5 _ 2 = 400
Loesung dieses Rechenraetsels: 8 * 5 * 4 * 5 : 2 = 400

Rechenraetsel mit 5 Operatoren: 5 _ 9 _ 9 _ 4 _ 9 _ 6 = 14586
Loesung dieses Rechenraetsels: 5 * 9 * 9 * 4 * 9 + 6 = 14586

Rechenraetsel mit 6 Operatoren: 3 _ 4 _ 6 _ 7 _ 2 _ 8 _ 4 = 224
Loesung dieses Rechenraetsels: 3 * 4 : 6 * 7 : 2 * 8 * 4 = 224

Rechenraetsel mit 7 Operatoren: 6 _ 9 _ 4 _ 2 _ 7 _ 5 _ 7 _ 7 = 418
Loesung dieses Rechenraetsels: 6 * 9 * 4 * 2 + 7 * 5 - 7 * 7 = 418

Rechenraetsel mit 8 Operatoren: 3 _ 7 _ 3 _ 3 _ 2 _ 4 _ 7 _ 2 _ 7 = 259
Loesung dieses Rechenraetsels: 3 * 7 * 3 * 3 + 2 * 4 * 7 + 2 * 7 = 259

Rechenraetsel mit 9 Operatoren: 9 _ 8 _ 7 _ 4 _ 8 _ 9 _ 9 _ 7 _ 2 _ 7 = 9121
Loesung dieses Rechenraetsels: 9 * 8 * 7 : 4 * 8 * 9 + 9 * 7 - 2 * 7 = 9121

Rechenraetsel mit 10 Operatoren: 6 _ 9 _ 5 _ 6 _ 2 _ 8 _ 3 _ 5 _ 3 _ 4 _ 9 = 2874
Loesung dieses Rechenraetsels: 6 + 9 - 5 * 6 + 2 * 8 * 3 * 5 * 3 * 4 + 9 = 2874

Rechenraetsel mit 11 Operatoren: 2 _ 5 _ 9 _ 7 _ 5 _ 2 _ 9 _ 6 _ 2 _ 9 _ 4 _ 2 = 5767
Loesung dieses Rechenraetsels: 2 - 5 + 9 * 7 * 5 * 2 * 9 + 6 * 2 * 9 - 4 * 2 = 5767

Rechenraetsel mit 12 Operatoren: 3 _ 5 _ 9 _ 8 _ 2 _ 6 _ 7 _ 5 _ 7 _ 7 _ 8 _ 6 _ 6 = 5630
Loesung dieses Rechenraetsels: 3 * 5 + 9 * 8 * 2 * 6 * 7 - 5 - 7 * 7 * 8 - 6 * 6 = 5630

Rechenraetsel mit 13 Operatoren: 7 _ 5 _ 4 _ 9 _ 9 _ 7 _ 4 _ 5 _ 4 _ 3 _ 3 _ 9 _ 6 _ 2 = 4414
Loesung dieses Rechenraetsels: 7 + 5 - 4 + 9 * 9 + 7 + 4 * 5 * 4 * 3 : 3 * 9 * 6 - 2 = 4414

Rechenraetsel mit 14 Operatoren: 7 _ 5 _ 6 _ 3 _ 6 _ 6 _ 8 _ 9 _ 7 _ 2 _ 4 _ 6 _ 5 _ 4 _ 6 = 51116
Loesung dieses Rechenraetsels: 7 * 5 * 6 : 3 * 6 * 6 * 8 * 9 : 7 * 2 - 4 - 6 * 5 * 4 * 6 = 51116

Rechenraetsel mit 15 Operatoren: 9 _ 2 _ 8 _ 6 _ 9 _ 5 _ 9 _ 6 _ 9 _ 4 _ 8 _ 8 _ 7 _ 9 _ 5 _ 7 = 63120
Loesung dieses Rechenraetsels: 9 - 2 + 8 * 6 * 9 - 5 * 9 + 6 + 9 * 4 * 8 * 8 * 7 : 9 * 5 * 7 = 63120

Rechenraetsel mit 16 Operatoren: 4 _ 8 _ 3 _ 7 _ 3 _ 2 _ 3 _ 4 _ 5 _ 2 _ 6 _ 7 _ 4 _ 7 _ 9 _ 4 _ 2 = 41947
Loesung dieses Rechenraetsels: 4 - 8 - 3 * 7 - 3 * 2 * 3 * 4 * 5 - 2 + 6 * 7 * 4 * 7 * 9 * 4 - 2 = 41947

Rechenraetsel mit 17 Operatoren: 6 _ 9 _ 8 _ 5 _ 9 _ 5 _ 5 _ 2 _ 6 _ 5 _ 5 _ 2 _ 8 _ 5 _ 7 _ 7 _ 6 _ 3 = 88980
Loesung dieses Rechenraetsels: 6 - 9 - 8 * 5 * 9 * 5 * 5 - 2 - 6 + 5 * 5 * 2 * 8 * 5 * 7 * 7 - 6 - 3 = 88980

Rechenraetsel mit 18 Operatoren: 3 _ 5 _ 5 _ 8 _ 4 _ 3 _ 8 _ 4 _ 9 _ 4 _ 9 _ 2 _ 3 _ 7 _ 2 _ 6 _ 7 _ 6 _ 7 = 45536
Loesung dieses Rechenraetsels: 3 * 5 - 5 - 8 + 4 * 3 * 8 * 4 * 9 * 4 + 9 * 2 * 3 * 7 * 2 * 6 * 7 - 6 * 7 = 45536

Rechenraetsel mit 19 Operatoren: 5 _ 8 _ 7 _ 5 _ 3 _ 5 _ 4 _ 7 _ 7 _ 3 _ 8 _ 8 _ 2 _ 6 _ 3 _ 4 _ 8 _ 5 _ 5 _ 4 = 66866
Loesung dieses Rechenraetsels: 5 * 8 * 7 * 5 * 3 + 5 * 4 * 7 * 7 * 3 * 8 * 8 * 2 : 6 + 3 - 4 * 8 - 5 - 5 * 4 = 66866

Rechenraetsel mit 20 Operatoren: 2 _ 5 _ 3 _ 5 _ 3 _ 6 _ 4 _ 7 _ 7 _ 5 _ 6 _ 7 _ 2 _ 5 _ 4 _ 2 _ 6 _ 9 _ 8 _ 5 _ 4 = 95259
Loesung dieses Rechenraetsels: 2 - 5 * 3 * 5 + 3 + 6 * 4 * 7 * 7 * 5 * 6 * 7 * 2 : 5 - 4 * 2 * 6 * 9 * 8 + 5 - 4 = 95259

Rechenraetsel mit 21 Operatoren: 5 _ 3 _ 3 _ 7 _ 8 _ 8 _ 7 _ 5 _ 2 _ 5 _ 3 _ 7 _ 6 _ 4 _ 4 _ 7 _ 4 _ 5 _ 3 _ 6 _ 5 _ 3 = 82051
Loesung dieses Rechenraetsels: 5 * 3 : 3 * 7 * 8 * 8 * 7 * 5 + 2 * 5 * 3 * 7 : 6 * 4 * 4 * 7 - 4 + 5 - 3 * 6 * 5 * 3 = 82051

Rechenraetsel mit 22 Operatoren: 5 _ 5 _ 3 _ 6 _ 5 _ 9 _ 2 _ 9 _ 3 _ 2 _ 3 _ 3 _ 9 _ 2 _ 8 _ 8 _ 2 _ 9 _ 5 _ 5 _ 4 _ 3 _ 4 = 990572
Loesung dieses Rechenraetsels: 5 + 5 * 3 * 6 - 5 * 9 * 2 * 9 * 3 * 2 + 3 * 3 + 9 * 2 * 8 * 8 * 2 * 9 * 5 : 5 * 4 * 3 * 4 = 990572

3.1 Laufzeit und Obergrenzen

Insgesamt kann das Programm Rechenrätsel mit bis zu 23 Operatoren berechnen. Diese obere Grenze ist tatsächlich nicht ausschließlich durch die Laufzeit, sondern auch durch das Abspeichern der Zahlen im Programm ergeben.
Für die oben beschriebenen Zahlen muss, um die Existenz eines eindeutigen Ergebnisses zu gewährleisten, das Produkt aller Ziffern gespeichert werden, was in einer Variable des Datentyps `long long` geschieht. Dieser Datentyp ist in C++ der größte Datentyp mit Vorzeichen und kann Zahlen mit einem Betrag von etwa $9 \cdot 10^{18}$ oder weniger speichern (Es wird ein Datentyp mit Vorzeichen benötigt, da das Programm auch negative Zahlen in den Mengen M_i zwischenspeichert).
Für $n > 23$ überschreitet das Produkt von $(n+1)$ zufällig gewählten Ziffern aber häufig die Beschränkung dieses Datentyps. Ohne eine Implementierung von Big-Nums (oder der Verwendung des in g++ verfügbaren Datentyps `__int128_t`) kann ein Programm in C++ Operatorenzahlen in dieser Größenordnung auch unabhängig von der Laufzeit häufig nicht behandeln.
Auf einem durchschnittlichen Computer liegt die Laufzeit des Programms für $n \leq 14$ stabil unter 10 Sekunden und für $15 \leq n \leq 21$ stabil unter 100 Sekunden, wenn dabei $o = 10^5$ gewählt wird. Hierbei ist zu beachten, dass die Laufzeit des Programms für eine Operatorzahl aufgrund des zufälligen Auswählens von Ziffern nicht immer gleich ist.
Für $n > 21$ liefert das Programm für $o = 10^5$ häufig das Produkt aller Ziffern für die Zahl e zurück. Für $o = 10^6$ benötigt das Programm für $n = 22$ circa 50 Minuten und für $n = 23$ circa 107 Minuten. Die Laufzeit des Programms ist somit insgesamt exponentiell, liegt für $n \leq 23$ aber noch in einem akzeptablen Bereich.

4 Quellcode

Es folgt die Definition von Makros, die die Lesbarkeit des Quellcodes durch die Abkürzung längerer Datentypen erhöhen. Auch die Obergrenze o wird als Makro und nicht durch eine direkte Eingabe des Nutzers definiert, weil diese Obergrenze bei mehrmaligem Gebrauch des Programms seltener modifiziert werden muss als etwa die Anzahl der Operatoren (In der Einsendung sind mehrere Programme verfügbar, die alle Werte von o in den Beispielen abdecken oder eine entsprechende Eingabe des Nutzers unterstützen). Statt der Anzahl der Operatoren wird die Anzahl der Ziffern in der Variable `zahlen` gespeichert:

```
#define ll long long
#define plb pair<ll, bool>
#define OBERGRENZE 100000
```

Das Festlegen der Ziffern des Rechenrätsels durch einen Zufallsgenerator:

```
vector<ll> raetsel(zahlen);
//Dieser Container enthaelt die Ziffern auf der linken Seite des Rechenraetsels
//Randomisiertes Initialisieren des Containers raetsel:
for(int i = 0; i < zahlen; i++)
{
    if(i == 0 || raetsel[i-1] != 2)
    {
        //Die Zahl 2 darf fuer die aktuelle Ziffer verwendet werden
        raetsel[i] = 2+rand()%8;
    }
    else
    {
        //Die Zahl 2 darf fuer die aktuelle Ziffer nicht verwendet werden
        raetsel[i] = 3+rand()%7;
    }
}
```

Initialisierung der Container `preProdukt` und `postProdukt`

```
vector<ll> preProdukt(zahlen); //Der i-te Wert ist das Produkt der ersten i Zahlen
vector<ll> postProdukt(zahlen);
//Der i-te Wert ist das Produkt aller Zahlen, die auf die i-te folgen (ohne diese selbst)
preProdukt[0] = raetsel[0];
for(int i = 1; i < zahlen; i++)
{
    preProdukt[i] = preProdukt[i-1]*raetsel[i];
}
postProdukt[zahlen-1] = 0;
postProdukt[zahlen-2] = raetsel[zahlen-1];
for(int i = zahlen-3; i >= 0; i--)
{
    postProdukt[i] = postProdukt[i+1]*raetsel[i+1];
    //Abfangen von Ueberlaeufen in dieser Zahl:
    if(postProdukt[i]/raetsel[i+1] != postProdukt[i+1])
    {
        //Abbruch des Programms mit einer informativen Fehlermeldung
    }
}
```

Die Definition der für das Speichern der Mengen M_i und S_i nötigen Datenstrukturen:

```
vector<vector<plb>> m(zahlen, vector<plb> (0));
//Container fuer die mit den ersten i Zahlen erzeugbaren Ergebnisse
vector<vector<plb>> s(zahlen, vector<plb> (0));
//Container fuer die durch Multiplikaton und Division erzeugten Summanden fuer jede Zahl
//In den obigen Containern ist der zu einer Zahl gehoerige bool-Wert genau dann True,
//wenn sie eindeutig darstellbar ist

vector<unordered_map<ll, ll>> mPos(zahlen);
//Hashtabellen mit den Indizes bestimmter Elemente in m
vector<unordered_map<ll, ll>> sPos(zahlen);
//Hashtabellen mit den Indizes bestimmter Elemente in s
vector<unordered_map<ll, string>> mLoes(zahlen);
//Hashtabellen mit den Operatoren-loesungen fuer Elemente von m
vector<unordered_map<ll, string>> sLoes(zahlen);
//Hashtabellen mit den Operatoren-loesungen fuer Elemente von s
```

Der nächste Abschnitt enthält die Initialisieren der Container `m` und `s`. Gezeigt werden sollen dabei die Strukturen insbesondere der beteiligten for-Schleifen. Die eigentlichen Anweisungen für das Modifizieren dieser Container wurden ausgelassen bzw. durch Kommentare ersetzt, um die Lesbarkeit des Codes zu erhöhen:

```
for(int i = 0; i < zahlen; i++) //Berechnen von m[i] fuer jede Ziffer im Rechenraetsel
{
    //Aktualisieren der Summanden:
    for(int j = 0; j < i; j++)
    {
        //Aktualisieren des Containers s[j]
        ll anzahl = (ll) s[j].size(); //Anzahl der bisher gefundenen Summanden
        for(ll k = 0; k < anzahl; k++) //Multiplikation mit der neuen Zahl:
        {
            //Eintraege des Containers werden aktualisiert
        }
        for(ll k = 0; k < anzahl; k++) //Division durch die neue Zahl
        {
            //Eintraege des Containers werden aktualisiert
        }
    }
    //Aktualisieren des Containers s[i]
    s[i].push_back({raetsel[i], true});
    sPos[i][raetsel[i]] = 0;
    sLoes[i][raetsel[i]] = "";

    //Berechnen aller moeglichen Ergebnisse fuer die ersten i Zahlen:
    //Alle bisherigen Zahlen bilden einen einzigen Summanden:
    for(plb p : s[0])
    {
        if(p.first - postProdukt[i] < OBERGRENZE || p.first == preProdukt[i])
        {
            //Die Zahl ist klein genug oder die groesstmoegliche
            m[i].push_back({p.first, p.second});
            mPos[i][p.first] = (ll) m[i].size() - 1;
            mLoes[i][p.first] = sLoes[0][p.first];
            //Diese Anweisung bleiben bestehen als Beispiel fuer das Einfuegen eines neuen Elements
        }
    }
    //Summanden werden zu bisherigen Ergebnissen addiert
    for(int j = 1; j <= i; j++) //Erste Zahl des letzten Summanden
    {
        for(plb p : m[j-1]) //Durchgehen aller vorherigen Ergebnisse
        {
            for(plb a : s[j]) //Durchgehen aller Summanden zu diesem Ergebnis
            {
                //Addition des Summanden unter Beruecksichtigung der Eindeutigkeit
                //Einfuegen in den Container M[i]

                //Gleiches fuer Subtraktion
            }
        }
    }
}
```

Auswählen einer Zahl e als Ergebnis

```
//Nun Auswaehlen eines Ergebnisses aus dem letzten Container m
//Zunaechst Loeschen aller mehrdeutigen Elemente:
for(ll i = 0; i < (ll) m[zahlen-1].size(); i++)
{
    if(!m[zahlen-1][i].second)
    {
        m[zahlen-1].erase(m[zahlen-1].begin()+i);
        i--;
    }
}
//Wenn moeglich, Loeschen des groessten Elements:
if(m[zahlen-1].size() > 1)
{
    m[zahlen-1].erase(max_element(m[zahlen-1].begin(), m[zahlen-1].end()));
}

//Nun zufaelliges Auswaehlen eines Ergebnisses:
ll ergebnis = m[zahlen-1][rand()%m[zahlen-1].size()].first;
string loesung = mLoes[zahlen-1][ergebnis];
```