Aufgabe 2: Geburtstag

Teilnahme-ID: 52570

Bearbeiter dieser Aufgabe:

Florian Rädiker (16 Jahre)

18. April 2020

Inhalt

1 Lösungsidee

Sind nur Addition und Multiplikation erlaubt, lässt sich die Darstellung für eine Zahl vergleichsweise einfach finden. Solange im Term keine Kommata und damit keine Nicht-Ganzzahlen vorkommen dürfen (und dies ist laut Aufgabenstellung der Fall), können mit Addition und Multiplikation nur Ganzzahlen dargestellt werden, weil das Ergebnis der Addition oder Multiplikation zweier Ganzzahlen immer ganzzahlig ist. Es existieren somit endlich viele Terme für eine Zahl, die Addition und Multiplikation verwenden und von denen der beste in endlicher Zeit gefunden werden kann. Beispielsweise ist die Primfaktorzerlegung von 2019 $3 \cdot 673$, dementsprechend gibt es für 2019 nur eine Zerlegung, die Multiplikation verwendet. Für die Addition reichen die Zerlegungen von $1 + 2018$ bis $1009 + 1010$. Die Operanden der Zerlegungen werden rekursiv nach demselben Schema zerlegt. Es entsteht ein Suchbaum, in dem mittels Tiefensuche der beste Term, zusammengesetzt aus rekursiv erstellten Zerlegungen, gefunden wird. Die Blätter des Suchbaums sind Zahlen, die nur aus der vorgegebenen Ziffer $d \in \{1 \dots 9\}$ bestehen.[1] Diejenige Zerlegung mit den Operanden, deren Terme in der Summe am wenigsten Ziffern benötigen, ist die beste Zerlegung für eine Zahl. Diese wird genutzt, um den optimalen Term zu bilden: Die beiden Werte (Summanden bzw. Faktoren) werden durch die gefundenen Terme ersetzt. Am Beispiel mit $d = 3$ wird die beste Zerlegung für 2019, `3+2016`, zu `3+((3+3)*(3+333))`, wobei der rekursiv gefundene Term für 2016 `(3+3)*(3+333)` ist. Die Zerlegung `3+2016` benötigt 7 Ziffern. Es ist nicht ausgeschlossen, dass es mehrere Zerlegungen gibt, die mit gleich vielen Ziffern auskommen; es reicht aber aus, eine davon zu finden.

Möchte man auch Subtraktion und Division zulassen, steht man vor dem Problem, dass es theoretisch unendlich viele Möglichkeiten gibt, eine Zahl a zu „zerlegen“:

$$a = (a + 1) - 1 = (a + 2) - 2 = \dots = \frac{2a}{2} = \frac{3a}{3} = \dots$$

Aus diesem Grund ist es offensichtlich sinnvoll zu überlegen, welches die größtmögliche Zahl a ist, die mit $n \in \mathbb{N}^+$ Ziffern nur mit der Ziffer d dargestellt werden kann. Darauf wird in Abschnitt 1.1 eingegangen.

Im Folgendem werden zwei Verfahren vorgestellt. Das eine Verfahren (Abschnitt 1.3) nutzt die angesprochene Tiefensuche und findet einen Term recht schnell, berücksichtigt dafür aber keine nichtganzzahligen[2] Teilterme (zur Verwendung von Nicht-Ganzzahlen in Termen siehe Abschnitt 1.2). Das andere Verfahren namens Termbildung erzeugt alle Terme bis zu einer vorgegebenen Ziffernanzahl und kann für eine Beschleunigung der Tiefensuche sowie für Potenzen und Fakultäten verwendet werden (Abschnitt 1.4). Es kann auch nicht-ganzzahlige Teilterme berücksichtigen.

1.1 Größtmögliche Zahlen mit Grundrechenarten

Von den Grundrechenarten ist offensichtlich die Multiplikation am besten geeignet, um große Zahlen mit möglichst wenig Ziffern zu erhalten. Eine weitere Möglichkeit ist, „Schnapszahlen“ zu bilden, indem d n-mal hintereinandergeschrieben wird. d ist eigentlich keine Schnapszahl, wird aber als solche betrachtet. Beide Möglichkeiten könnten dazu geeignet sein, die größtmögliche Zahl mit n Ziffern darzustellen. Um herauszufinden, mit welcher Darstellung größere Zahlen erreicht werden, wird $d \cdot d \cdot \dots \cdot d = d^n$ mit $dd \dots d$ verglichen, wobei $d_n d_{n-1} \dots d_1 = d \cdot \sum_{i=0}^{n-1} 10^i$, sodass sich für $d = 4$ beispielsweise $ddd = 444$ ergibt. Eine andere Darstellung als die Summenformel für das Hintereinanderschreiben von d ist allerdings sinnvoll, um die beiden Terme besser miteinander vergleichen zu können.

[1] Dass die Blätter immer Zahlen sind, die nur aus der vorgegebenen Ziffer bestehen, geht nur, weil diese Darstellung der Zahl (meist) optimal ist und am wenigsten Ziffern benötigt (siehe Abschnitt 1.1).
[2] „Nicht-Ganzzahl“ $\in \mathbb{Q} \setminus \mathbb{N}$

Allgemein kann eine Schnapszahl mit der Formel $d\frac{B^n-1}{B-1}$ berechnet werden,[3] wobei B die Basis repräsentiert, also berechnet sich eine Schnapszahl mit $B = 10$ aus $d\frac{10^n-1}{9}$.

Es soll nun bewiesen werden, dass die Schnapszahl mit derselben Ziffernanzahl immer größer ist als das Ergebnis der Multiplikation, außer für $n = 1$, da in diesem Fall beide Darstellungen dieselbe Größe (d) erreichen.

$$d^n \leq d\frac{10^n-1}{9} \Leftrightarrow 9 \cdot d^{n-1} \leq 10^n - 1$$

Am größten wird der linke Term mit $d = 9$

$$9^n \leq 10^n - 1$$

Eine Schnapszahl bestehend aus der vorgegebenen Ziffer ist also immer die größtmögliche Zahl, die mit der entsprechenden Ziffernanzahl dargestellt werden kann, da die Ungleichung für $n \geq 1$ wahr ist.

1.1.1 Berechnung der Mindestzifferzahl

Mit diesem Wissen kann für jede (Ganz-)Zahl bestimmt werden, wie viele Ziffern zur Darstellung mit Grundrechenarten mindestens benötigt werden („Mindestzifferzahl“). Bis zur ersten Schnapszahl bestehend aus Ziffer d mit Länge 1 ist die Mindestzifferzahl 1. Bis dd ist sie 2, von $dd + 1$ bis ddd sind mindestens drei Ziffern zur Darstellung notwendig. Es ergibt sich folgende Berechnungsvorschrift für die Mindestzifferzahl $m(a, d)$ der Zahl a mit Ziffer $d \in \{1 \dots 9\}$:

$$m(a,d) = \begin{cases} l(a), & a \leq r(d,l) \\ l(a)+1, & a > r(d,l) \end{cases}$$

$$l(a) = \lfloor \log_{10}(a) \rfloor + 1 \text{ (Anzahl der Ziffern von a im Dezimalsystem)}$$

$$r(d,l) = d\frac{10^l-1}{9}$$

Nimmt man auch Potenzen hinzu, unterscheidet sich die Mindestzifferzahl stark von der ohne Potenzen. Die größten Zahlen werden mit Potenzen erreicht, indem eine Ziffer mit einem möglichst großen Exponenten potenziert wird:

$d^{d^{\dots^d}} = d^{(d^{(\dots^d)})}$, wobei d n-mal verwendet wird

Damit verschiebt sich die Grenze für Mindestzifferzahlen allerdings so weit nach oben, dass eine sinnvolle Nutzung dieser Grenze für die Tiefensuche nicht mehr möglich ist. Zudem kann mit Fakultäten eine unendlich große Zahl mit nur einer Ziffer dargestellt werden:

$$\Big(\big((d!)!\big)! \dots\Big)!$$

1.1.2 Schnapszahlen sind (meistens) die optimale Darstellung

Die Tiefensuche baut darauf auf, dass Schnapszahlen, die aus der vorgegebenen Ziffer bestehen, immer optimal sind. Beweisen lässt sich das mit obigen Überlegungen, allerdings nur für die Grundrechenarten. Sobald auch Potenzen und Fakultäten erlaubt sind, könnte es durchaus sein, dass ein besserer Term für eine Schnapszahl als die Darstellung der Schnapszahl im Dezimalsystem existiert. Ob dies tatsächlich der Fall ist, wurde empirisch überprüft, siehe dazu Abschnitt 2.7.1. Das Ergebnis lautet: Bis zu einer sinnvollen maximalen Ziffernanzahl sind Schnapszahlen immer optimal und können auch mit Fakultäten und Potenzen nicht besser dargestellt werden.

[3] en.wikipedia.org/wiki/Repdigit

1.2 Sollten auch Nicht-Ganzzahlen betrachtet werden?

Zahlen, die keine Ganzzahlen sind, können – auch mit Fakultäten und Potenzen – nur mithilfe einer Division erreicht werden. Da das Ergebnis aber ganzzahlig sein soll, weil Jahreszahlen (normalerweise) ganzzahlig sind, stellt sich die Frage, ob nicht zumindest Teilterme nicht-ganzzahlig sein können. Mit der Tiefensuche können keine Nicht-Ganzzahlen berücksichtigt werden, mit dem zweiten Verfahren (Abschnitt 1.4 „Termbildung“) ist dies möglich. Termbildung erzeugt alle möglichen Terme bis zu einer vorgegebenen Ziffernanzahl. Um herauszufinden, ob Nicht-Ganzzahlen in optimalen Termen tatsächlich vorkommen und wenn ja wie häufig, wurden per Termbildung zunächst alle Terme (mit Fakultäten und Potenzen) für jede Ziffer bis zu einer vorgegebenen Ziffernanzahl (d=3 und d=4: bis Ziffernanzahl 6; d=1: bis Ziffernanzahl 8; ansonsten bis Ziffernanzahl 7)[4] erzeugt, und zwar ohne nicht-ganzzahlige Quotienten. Danach wurde die Suche wiederholt, diesmal aber auch mit nicht-ganzzahligen Divisionen. Für jeden Term, der eine Division enthält, selbst aber zu einer Ganzzahl auswertet, wurde überprüft, ob er mit derselben Ziffernanzahl durch die erste Suche gefunden wurde. Das Ergebnis: Es gibt optimale Terme, die nicht-ganzzahlige Teilterme verwenden. Die Ausgabe des Programms ist in `Aufgabe2/output_terms_with_fractions.txt` zu finden. Die Ausgabe wurde auf Terme beschränkt, die zu Ganzzahlen kleiner oder gleich 2980 auswerten. Insgesamt wurden 139 Zahlen gefunden, deren Terme durch Verwendung nicht-ganzzahliger Teilterme weniger Ziffern benötigen. Diese Terme treten erst ab einer Ziffernanzahl von 5 auf, bei den Ziffern 1, 2 und 9 erst ab 7 Ziffern.

Ziffer	1	2	3	4	5	6	7	8	9	Σ
Anzahl Terme	13	3	14	11	17	23	14	30	14	139

Anzahl der Zahlen ≤ 2980, die mit nicht-ganzzahligen Teiltermen besser dargestellt werden können

Für die vorgegebenen Zahlen benötigen durch die Berücksichtigung nicht-ganzzahliger Teilterme nur zwei Terme weniger Ziffern; beide Terme verwenden Fakultäten und Potenzen: Statt `2020=((77/7)+(7*((7*((7*7)-7))-7)))` wird `(((7+(7+7))*(((7!)/(7*7))-7))+7)` gefunden und `2980=(((7!)-((7+7)/7))-(7*(7*((7*7)-7))))` kann durch `((7+((((7!)+((7!)+7))/7)/7))*(7+7))` besser dargestellt werden (jeweils 8 statt 9 Ziffern). Das Finden der Terme für die vorgegebenen Ziffern mit der Berücksichtigung nicht-ganzzahliger Teilterme dauert jedoch fast 4-mal länger als ohne. Aus diesem Grund gibt es zwei Verfahren: Die Tiefensuche ist effizienter als die Termbildung, berücksichtigt dafür aber keine nicht-ganzzahligen Teilterme. Termbildung ist nicht so schnell, aber es können auch nicht-ganzzahlige Teilterme gefunden werden.

1.2.1 Nicht-ganzzahliger Exponent

Es könnte sein, dass Terme weniger Ziffern benötigen, wenn der Exponent einer Potenz ein nicht-ganzzahliger Bruch ist. Es gilt $x^{a/b} = \sqrt[b]{x^a}$. Der Term sollte eine Ganzzahl ergeben und a, b und c sollten Ganzzahlen sein. Am einfachsten ist es wohl, die Quadratwurzel zu berechnen, mit einem Exponenten von $\frac{1}{2}$. Der Exponent selbst benötigt auch Ziffern (bei $\frac{1}{2}$ sind es für jede Ziffer außer der 3 genau drei Ziffern ($\frac{d}{d+d}$), mit Ziffer 3 sind es zwei: $\frac{3}{3!}$). Natürlich sind auch andere Exponenten wie $\frac{2}{3}$ oder $\frac{10}{8}$ vorstellbar. Es stellt sich die Frage, ob im Term $number = x^{\frac{a}{b}}$ der Term auf der rechten Seite weniger Ziffern benötigt als $number$ ohne nicht-ganzzahlige Exponenten. Dafür müssen sowohl die Anzahl der Ziffern von x als auch die Anzahl der Ziffern des Exponenten berücksichtigt werden. Um das zu überprüfen, wurden für jede Ziffer bis zur Zahl 3000 fast alle Terme mit Grundrechenarten, Potenzen und Fakultäten berechnet (siehe Umsetzung, Abschnitt 2.7.2). Das Ergebnis ist: Mit den getesteten Zahlen

[4] Die Ziffernanzahl variiert, weil für jede Ziffer unterschiedlich viele Terme bis zu einer Ziffernanzahl gefunden werden, siehe auch das Diagramm auf Seite 15. Damit verändert sich auch die Laufzeit für dieselbe maximale Ziffernanzahl. Die Ziffernanzahlen wurden so gewählt, dass das Programm für jede Ziffer möglichst dieselbe Zeit benötigt und damit ähnlich viele Terme erzeugt.

gibt es keinen Fall, bei dem durch einen nicht-ganzzahligen Exponenten eine geringere Ziffernanzahl erzielt werden kann. Es wurde mit Werten für a von 1 bis 10 und Werten für b von 2 bis 10 getestet.

1.3 Rekursive Zerlegung und Tiefensuche

1.3.1 Algorithmus ohne Optimierungen

Der in der Einleitung beschriebene Lösungsansatz kann mit folgendem Algorithmus beschrieben werden.

1. Eingabe: Zahl, für die ein Term gesucht wird
2. Falls die Zahl eine Schnapszahl bestehend aus der vorgegebenen Ziffer ist: Es handelt sich um ein Blatt des Suchbaums. Die Zahl selbst ist nach Abschnitt 1.1 bereits der bestmögliche Term, gib sie zurück
3. Finde Zerlegungen für diese Zahl (Addition, Subtraktion, Multiplikation oder Division)
4. Setze die bisher gefundene beste Ziffernanzahl auf unendlich und den bisher besten gefundenen Term auf undefiniert
5. Für jede Zerlegung:
    a. Nutze diesen Algorithmus (ab Schritt 1), um für jeden der zwei Operanden rekursiv eine Suche nach einem Term durchzuführen
    b. Falls die Summe der Ziffernanzahlen der beiden Operanden besser (d.h. kleiner) ist als das bisher bestgefundene Ergebnis: Speichere diese Zerlegung mit den Termen für die beiden Operanden als bisher bestes Ergebnis
6. Ausgabe: bestes gefundenes Ergebnis

Dieses Vorgehen beachtet keine Sonderfälle und kann weiter optimiert werden. Dies wird in den folgenden Abschnitten besprochen.

1.3.2 Eine Zahl analysieren

Dieser Abschnitt befasst sich mit dem Finden der Zerlegungen für eine Zahl (Schritt 3). Wie eingangs beschrieben, gibt es mit Addition und Multiplikation endlich viele Zerlegungsmöglichkeiten. Bei der Addition reichen diese von $1 + (a - 1)$ bis $\left\lfloor \frac{a}{2} \right\rfloor + \left\lceil \frac{a}{2} \right\rceil$. Für die Multiplikation müssen alle ganzen Zahlen $1 < x \leq \lfloor \sqrt{a} \rfloor$ gefunden werden, für die $a \bmod x = 0$ gilt. x und a/x sind zwei ganzzahlige Faktoren, deren Multiplikation die gesuchte Zahl ergibt. Mangels eines effizienten Faktorisierungsverfahrens wird über alle Zahlen von 1 bis zur abgerundeten Quadratwurzel der Zahl iteriert. Gilt für eine Zahl die Modulo-null-Bedingung, ist sie einer der möglichen Werte für x. Etwas effizienter ist die Probedivision, bei der nur mit Primzahlen, beginnend bei 2, die Modulo-null-Bedingung überprüft wird. Sobald eine Primzahl p gefunden wurde, auf die die Bedingung zutrifft, (p ist also ein Teiler) wird nicht mehr a weiter zerlegt, sondern a/p. $a/p \bmod p$ könnte, ohne dass p auf die nächsthöhere Primzahl gesetzt wird, aber auch 0 ergeben, und darum wird dies geprüft, bevor p erhöht wird. Erst, wenn die aktuelle Zahl nicht durch p teilbar ist, wird p auf die nächsthöhere Primzahl gesetzt. Abgebrochen wird, sobald p größer als die Quadratwurzel der aktuellen Zahl ist. Diese Zahl ist dann entweder 1 oder eine Primzahl (die Primzahl ist ebenfalls ein Faktor in der Primfaktorzerlegung). Am Beispiel $a = 873$ ergibt sich: $873 \bmod 2 = 1$, also ist 2 kein möglicher Wert für x. $873 \bmod 3 = 0$, also ist 3 ein möglicher Teiler und es wird mit $873/3 = 291$ weitergerechnet. $291 \bmod 3$ ist aber auch 0 (in der Primfaktorzerlegung steht also 3^2), weshalb mit $291/3 = 97$ weitergerechnet wird. $97 \bmod 3$ ergibt 1. 97 ist auch nicht durch die nächsthöhere Primzahl teilbar ($97 \bmod 5 = 2$) und auch nicht durch 7 ($97 \bmod 7 = 6$). Die nächste Primzahl, 11, ist größer als die Quadratwurzel von 97, daher ist die Suche beendet und die Primfaktorzerlegung von 873 lautet $3^2 \cdot 97$. Daraus ergibt sich $x \in \{3, 9\}$ ($9 = 3^2$).

Subtraktion und Division benötigen im Gegensatz dazu wie eingangs beschrieben eine Grenze, sodass bekannt ist, wann eine Zerlegung das bisher beste gefundene Ergebnis nicht mehr verbessern kann.

Mit der Formel aus Abschnitt 1.1 lässt sich für jede Zerlegung ermitteln, wie viele Ziffern mindestens benötigt werden. Mit diesem Wissen müssen nicht mehr alle Zerlegungen betrachtet werden, weil damit eine obere Schranke für Zerlegungen mit Subtraktion und Division existiert. Zerlegungen mit Subtraktion und Division werden nach dem folgenden Schema und in der dargestellten Reihenfolge berechnet (siehe Einleitung).

$$a = (a+1) - 1 = (a+2) - 2 = \cdots = (a+n) - n$$

$$a = \frac{2a}{2} = \frac{3a}{3} = \cdots = \frac{na}{n}$$

Mit größerem n steigt auch die Anzahl der mindestens benötigten Ziffern. Die Mindestzifferzahl für eine Zerlegung berechnet sich aus der Summe der beiden Mindestzifferzahlen für die Operanden. Es müssen keine Zerlegungen mehr betrachtet werden, sobald die Mindestzifferzahl für eine Zerlegung gleich oder größer der bisher besten gefundenen Anzahl ist, denn dann kann mit der aktuellen Zerlegung oder den folgenden (mit größerem n) kein besseres Ergebnis gefunden werden.[5] Für Addition und Subtraktion gilt das übrigens nicht: Die Mindestzifferzahl nimmt nicht stetig zu oder ab.[6]

Am vielversprechendsten ist es, mit den Zerlegungen zu beginnen, die die kleinste Mindestzifferzahl haben. Daher findet der Algorithmus zunächst alle Zerlegungen mit Addition und Multiplikation und ordnet sie jeweils ihrer Mindestzifferzahl zu.[7] Danach werden Zerlegungen mit aufsteigender Mindestzifferzahl, angefangen bei 2,[8] betrachtet und die Operanden der Zerlegung wiederum zerlegt. Für eine Mindestzifferzahl werden alle Zerlegungen mit Addition und Multiplikation, aber auch alle mit Division und Subtraktion betrachtet. Die Mindestzifferzahl wird solange um 1 erhöht, bis es kein besseres Ergebnis geben kann, weil die Mindestzifferzahl gleich der besten gefundenen Anzahl an Ziffern ist.

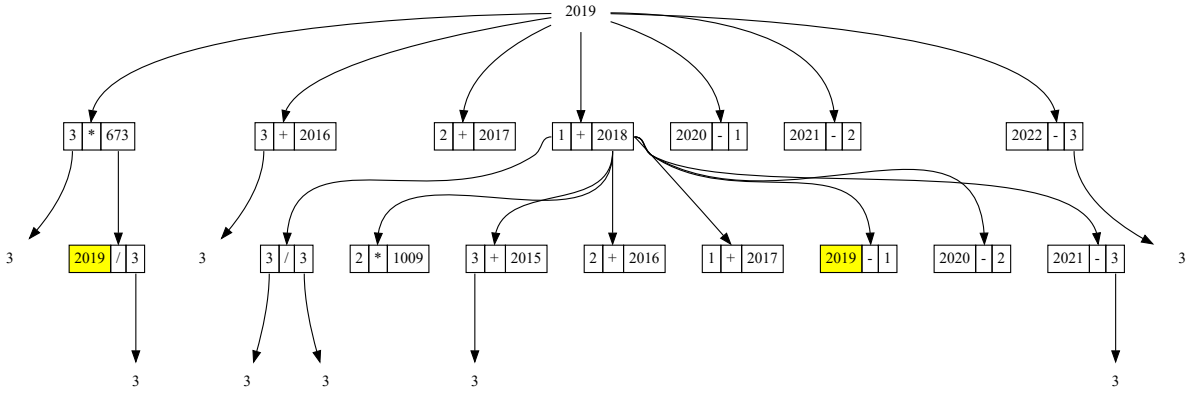

Abb. 1. Unvollständiger Suchbaum für die Zerlegung von 2019 mit Ziffer 3. Für 2019 sind alle Zerlegungen mit Mindestzifferzahl 5 dargestellt, für 2018 und 673 ist es nur eine Auswahl, um die Suche zu veranschaulichen. Für 2019 ist keine Zerlegung mit Division dargestellt, weil diese mindestens 6 Ziffern benötigte[9] (4038/2). Die Blätter werden ausschließlich durch die Ziffer 3 gebildet.

In Abb. 1 fällt auf, dass die gleichen Zahlen mehrfach Operanden sind, beispielsweise wird 1 von 1+2018, 1+2017, 2020-1 und 2019-1 gesucht (aus Gründen der Übersichtlichkeit ist „3/3“ nur einmal dargestellt, bei 1+2018). Auch 2016, 2017, 2020 und 2021 kommen mehrmals vor. Aus diesem Grund ist es sinnvoll, die Ergebnisse per Memoisation für spätere Berechnungen zu speichern, sodass keine

[5] Die maximalen Werte für n können sich für Subtraktion und Division unterscheiden.
[6] Zumindest nicht in der dargestellten Reihenfolge, in der die Zerlegungen für Addition und Multiplikation ermittelt werden.
[7] Wie die Mindestzifferzahl effizient für alle Grundrechenarten ermittelt werden kann, ohne dass sie für die zwei Operanden jeder Zerlegung neu berechnet wird, wird in Abschnitt 1.3.6 behandelt.
[8] Das muss nicht heißen, dass es immer eine Zerlegung mit mindestens zwei Ziffern gibt.
[9] Konjunktiv

Zerlegungen doppelt gesucht werden. Die Implementierung des Caches ist jedoch nicht trivial, weshalb in Abschnitt 1.3.5 auf die Memoisation eingegangen wird. Noch etwas fällt auf: Terme für eine Zahl werden gesucht, während ein Term für ebendiese Zahl gesucht wird (gelbe Hinterlegungen). Mit diesem Problem beschäftigt sich Abschnitt 1.3.4.

Wie sich aus der Abbildung erahnen lässt, kann es, sofern die Generierung von Zerlegungen zuerst die Addition 1+(a-1) liefert, am Anfang zum Versuch kommen, die gesuchte Zahl in eine Addition von 1en zu zerlegen (1+2018, 2018 wird zu 1+2017, 2017 wird zu 1+2016, …). Das ist der Grund dafür, dass für die aktuelle Mindestzifferzahl zuerst alle Multiplikationen und erst danach alle Additionen generiert werden. Dies entspricht der Reihenfolge in der Abbildung.[10] Es ist außerdem sinnvoll, pro Mindestzifferzahl erst Multiplikation und Addition zu betrachten und danach Division und Subtraktion: Da für Division und Subtraktion größere Zahlen benötigt werden, deren Terme tendenziell aus mehr Ziffern bestehen, ist davon auszugehen, dass Addition und Multiplikation in den optimalen Termen öfter verwendet werden.

1.3.3 Optimierung: Maximale Ziffernanzahl bei der Suche berücksichtigen

Die Tiefensuche kann weiter beschleunigt werden, wenn der Algorithmus zusätzlich zur Zahl, für die ein Term gefunden werden soll, auch eine maximale Ziffernanzahl als Eingabe erhält, die der resultierende Term maximal haben darf. Die bisher kleinste gefundene Ziffernanzahl sei c. Für eine Zerlegung werden die beiden Mindestzifferzahlen für die Operanden bestimmt, m_1 (erster Operand) und m_2 (zweiter Operand). Bei der Suche nach einem Term für den ersten Operanden darf der Term nicht mehr Ziffern als $c - m_2 - 1$ haben, ansonsten kann kein besseres Ergebnis erzielt werden. Wurde ein Term für den ersten Operanden mit der Ziffernanzahl l_1 gefunden ($l_1 \leq c - m_2 - 1$), darf der Term für den zweiten Operanden nicht mehr Ziffern als $c - l_1 - 1$ haben, damit das Ergebnis ein besseres ist. Wurde für den ersten Operanden kein Term gefunden, muss für den zweiten nicht mehr nach einem Term gesucht werden. Sollte eine maximale Ziffernanzahl übergeben worden sein, wird in Schritt 4 des Algorithmus die bisher beste gefundene Ziffernanzahl nicht mehr auf unendlich gesetzt, sondern auf *maximale Ziffernanzahl* + 1, obwohl der bisher beste gefundene Term eigentlich undefiniert ist. Diese Beschränkung wird dadurch an alle rekursiv berechneten Terme weitergegeben und hat zur Folge, dass die Rückgabe nur dann nicht undefiniert ist, wenn tatsächlich ein Term existiert, der weniger oder gleich viele Ziffern wie die übergebene maximale Ziffernanzahl hat. Diese Optimierung muss auch bei der Implementierung des Caches berücksichtigt werden, siehe Abschnitt 1.3.5.

Eine maximale Ziffernanzahl für die Operanden einer Zerlegung kann erst ermittelt werden, nachdem ein bisher bester Term gefunden wurde, da zu Beginn der Suche noch keine bisher beste Ziffernanzahl existiert. Das führt dazu, dass zu Beginn der Berechnung der Algorithmus im Baum der Zerlegungen sehr tief „wandert“. Durch eine Abschätzung der maximal benötigten Anzahl für den gesuchten Term kann das Verfahren beschleunigt werden. Das Ergebnis ist eine beschränkte Tiefensuche. Wenn der Algorithmus zu keinem Ergebnis kommt, war die maximale Anzahl zu klein und die Suche muss wiederholt werden (dies wird durch den Cache allerdings beschleunigt, siehe Abschnitt 1.3.5).

1.3.4 Tabuzahlen

Bei der Suche nach einem Term für eine Zahl kann es vorkommen, dass währenddessen ein Term für ebendiese Zahl gesucht wird. Das würde allerdings zu unerwünschten, theoretisch unendlich langen Rekursionen führen, weshalb Tabuzahlen eingeführt werden. Während der Tiefensuche wird eine Menge aller Zahlen gespeichert, für die gerade keine Zerlegung gesucht werden darf. Für die Zerlegungen von 673 in Abb. 1 gibt es zwei Tabuzahlen, nämlich 2019 und 673. Auf diese Weise wird

[10] In der Abbildung stehen rechts die Multiplikationen, dann folgen die Additionen usw. Der Grund dafür, dass die Additionszerlegungen für 2019 nicht mit „1+2018“, sondern mit „3+2016“ beginnen, liegt an der Reihenfolge, in der die Zerlegungen generiert werden. Siehe Abschnitt 1.3.6.

ausgeschlossen, dass als Zerlegung von 673 `2019/3` betrachtet wird. Im Cache kann durch dieses Verfahren allerdings ein falsches Ergebnis stehen, siehe den nächsten Abschnitt.

1.3.5 Optimierung: Memoisation

Der Cache ist ein Dictionary, in dem einer Zahl, sobald die Suche nach einem Term für diese Zahl beendet ist, der gefundene Term zugeordnet wird. Der Cache ist für Mehrfachverwendung ausgelegt und wird für alle Tiefensuchen mit der gleichen Ziffer verwendet.

Alle Informationen, die im Cache einer Zahl zugeordnet werden, sind:

1. Der beste gefundene Term. Der Term kann auch undefiniert sein, was bedeutet, dass aufgrund einer zu kleinen maximalen Anzahl kein Term gefunden wurde.
2. Die Ziffernanzahl des besten gefundenen Terms. Wenn der Term nicht undefiniert ist, entspricht diese Anzahl der tatsächlichen Ziffernanzahl. Wenn der Term allerdings undefiniert ist, ist diese Anzahl nicht unendlich, sondern *alte maximale Ziffernanzahl* + 1. Das liegt daran, dass wie in Abschnitt 1.3.3 beschrieben die beste bisher gefundene Ziffernanzahl auf *maximale Ziffernanzahl* + 1 gesetzt wird.
3. Die alte maximale Ziffernanzahl, mit der das gecachte Ergebnis berechnet wurde. Wird der Eintrag für eine Zahl aktualisiert, kann dieser Wert nur steigen, nicht kleiner werden. Wenn ein Term für eine Zahl nochmals gesucht wird, aber die maximale Ziffernanzahl im Cache ist größer als oder gleich groß wie die aktuelle maximale Ziffernanzahl, dann kann das Ergebnis im Cache nicht verbessert werden.
4. Die Menge der damaligen Tabuzahlen, mit denen das gecachte Ergebnis erreicht wurde.

Bevor ein Term für eine Zahl wie üblich gesucht wird, wird das Dictionary nach dieser Zahl durchsucht. Ist ein Eintrag vorhanden, kann auf das gecachte Ergebnis zurückgegriffen werden. Ob das gecachte Ergebnis direkt zurückgegeben werden kann, ist allerdings von verschiedenen Bedingungen abhängig.

Es kann nämlich vorkommen, dass im Cache wegen Tabuzahlen ein falsches Ergebnis steht. Das ist in der oben beschriebenen Konstellation der Fall: Wird im Anschluss an die Termsuche für 2019 eine weitere Termsuche für 673 durchgeführt, steht im Cache ein falsches Ergebnis, weil die beste Zerlegung für 673 `2019/3` ist (mit Ziffer 3). Diese Zerlegung wurde allerdings ausgeschlossen, weil 2019 eine Tabuzahl war. Für die erste Suche nach 673 (während der Termsuche nach 2019) war die Menge der Tabuzahlen noch $\{2019, 673\}$. Diese Tabuzahlen sind als alte Tabuzahlen im Cache gespeichert. Bei der folgenden, eigenständigen Termsuche für 673 sind die aktuellen Tabuzahlen allerdings nur noch $\{673\}$. Aus diesem Grund kann das gecachte Ergebnis auf keinen Fall zurückgegeben werden, wenn die alten Tabuzahlen keine Teilmenge der aktuellen Tabuzahlen sind. Wenn die Teilmengenbedingung jedoch wahr ist, kann das gecachte Ergebnis zurückgegeben werden, sofern es nicht undefiniert ist.

Ist die Teilmengenbedingung wahr und der gecachte Term ist undefiniert, liegt dies daran, dass die frühere maximale Ziffernanzahl zu niedrig war. Ist die aktuelle maximale Ziffernanzahl genau so groß oder kleiner, ist bekannt, dass auch mit der aktuellen maximalen Ziffernanzahl kein Ergebnis gefunden werden kann und es wird zurückgegeben, dass kein Ergebnis gefunden werden konnte. Ansonsten muss mit der gewöhnlichen Methode nach einem Term gesucht werden, allerdings gibt es einen kleinen Vorteil, wenn der gecachte Term nicht undefiniert ist: Dann nämlich ist bereits ein Ergebnis bekannt (auch, wenn es vielleicht nicht optimal ist), und es wird nun versucht, dieses Ergebnis zu verbessern, statt „von vorne“ anzufangen.

In der Darstellung der vollständigen Algorithmen (Abschnitt 1.3.7) sind die beschriebenen Schritte für den Cache übersichtlich dargestellt.

Am Ende der Suche nach einem Term wird das Ergebnis (auch, wenn kein Term gefunden wurde) im Cache gespeichert. Dabei werden auch die aktuellen Tabuzahlen gespeichert. Wurde die Zahl, die gerade gesucht wird, am Anfang im Cache gefunden, gab es alte Tabuzahlen. Beim Speichern der Tabuzahlen im Cache werden diese alten Tabuzahlen berücksichtigt. Nur eine Zahl, die sich in beiden Mengen (der Menge der alten Tabuzahlen und der Menge der aktuellen Tabuzahlen) befindet, wird in den Cache übernommen. Das geht, weil alle anderen Tabuzahlen aus den Mengen nur bei der einen, nicht aber bei der anderen Suche berücksichtigt wurden und dadurch alle Zerlegungen, die bei der einen Suche noch durch eine Tabuzahl nicht berücksichtigt werden konnten, bei der anderen Suche berücksichtigt wurden. An obigem Beispiel: Nach der eigenständigen Suche nach einem Term für 673 steht im Cache $\{673\}$ als die Menge alter Tabuzahlen. Alle Zerlegungen, die bei der ersten Suche nicht berücksichtigt werden konnten, weil 2019 eine Tabuzahl war, wurden bei der zweiten Suche berücksichtigt (denn dabei war 2019 keine Tabuzahl). Daher wird im Cache $\{2019, 673\} \cap \{673\} = \{673\}$ als alte Tabuzahlen gespeichert. Dies ist aber nur möglich, wenn die alte maximale Ziffernanzahl nicht kleiner als die aktuelle maximale Ziffernanzahl ist, weil ansonsten einige Zerlegungen bei der alten Suche aufgrund der geringeren maximalen Ziffernanzahl nicht berücksichtigt wurden.

Wenn der alte Eintrag im Cache überschrieben wird, kann es natürlich passieren, dass alte Informationen überschrieben werden, die eventuell „wertvoller“ waren, da sie beispielsweise eine größere maximale Ziffernanzahl verwendeten oder weniger Tabuzahlen hatten. Das ließe sich nur mit einem deutlich komplexeren Cache verhindern, der beispielsweise zu einer Zahl verschiedene Einträge speichert.

1.3.6 Optimierung: Effiziente Ermittlung der Mindestzifferzahlen für die Zerlegungen

Um während des Generierens der Zerlegungen mit Addition und Subtraktion die Mindestzifferzahl für jede Zerlegung zu erhalten, könnte die Mindestzifferzahl mittels der vorgestellten Formel für die beiden Operanden berechnet werden und die Summe ist dann die Mindestzifferzahl für die Zerlegung. Es fällt allerdings auf, dass sich bei einer Zerlegung mit Addition und Subtraktion die Mindestzifferzahlen der Operanden nur schrittweise verändern. Daher kann die Mindestzifferzahl, statt sie jedes Mal neu zu berechnen, beim Iterieren über die Werte für die Operanden erstellt werden. Dazu ist zunächst zu klären, wie die Zerlegungen einer Zahl a mit Addition, Subtraktion und Division generiert werden:[11]

Grundrechenart	Operand	Erste Zerlegung	Veränderung pro Schritt	Letzte Zerlegung
Addition	1. Summand	$\left\lfloor \frac{a}{2} \right\rfloor$	-1	1
	2. Summand	$\left\lceil \frac{a}{2} \right\rceil$	+1	$a - 1$
Subtraktion	Minuend	$a + 1$	+1	$a + n$
	Subtrahend	1	+1	n
Division	Dividend	$2a$	+a	na
	Divisor	2	+1	n

Für Addition, Subtraktion und Division ist dargestellt, welche Zerlegung als erstes („Erste Zerlegung“) und welche als letztes generiert wird. Die Spalte „Veränderung pro Schritt“ gibt an, welche Operation jeweils angewandt wird, um die nächste Zerlegung zu erhalten. Nach mehrmaliger Anwendung der Veränderung auf die Operanden gelangt der Algorithmus zur „Letzten Zerlegung“.

Für alle in der Tabelle dargestellten Operanden nehmen die Werte für jede neue Zerlegung in gleichmäßigen Schritten entweder zu oder ab (Spalte „Veränderung pro Schritt“). Die Werte sowie die

[11] Für Multiplikation wird die Optimierung nicht angewendet. Da durch alle möglichen Teiler iteriert wird, wäre eine Berechnung der Mindestzifferzahl auf die vorgestellte Weise allerdings auch möglich.

Mindestzifferzahlen für die Operanden werden per Generator (in der Implementierung ein Python-Generator) erzeugt, und zwar für jeden Operanden getrennt. Für alle in der Tabelle dargestellten Operanden der ersten Zerlegungen werden die Mindestzifferzahlen per Formel bestimmt. Der Generator berechnet dann, welche Schnapszahl als nächstes überschritten wird, und erzeugt bis zu dieser Zahl Werte für einen Operanden, entsprechend der Spalte „Veränderung“. Ist er bei der nächsten Schnapszahl angekommen, verändert er die Mindestzifferzahl entsprechend um 1 (+1 oder -1) und erzeugt Operanden, bis sich die Mindestzifferzahl das nächste Mal ändert. Der Generator erzeugt zu jedem Wert für den Operanden auch die Mindestzifferzahl. Eine Schleife führt die zwei Generatoren beider Operanden zusammen und berechnet für jede auf diese Weise entstehende Zerlegung die Mindestzifferzahl der Zerlegung als Summe der Mindestzifferzahlen der Operanden. Dass für jede Zerlegung die Summe gebildet werden muss, lässt sich vermeiden, indem die Generatoren nicht ihre eigene Mindestzifferzahl verändern, sondern die Mindestzifferzahl der Zerlegung.

Die Berechnung der Mindestzifferzahlen für die Zerlegungen unterscheidet sich zwischen den Grundrechenarten leicht. Bei der Addition sind beide Generatoren irgendwann beendet, bei Subtraktion und Division ist das nicht der Fall, sondern sie würden theoretisch unendlich viele Operanden mit Mindestzifferzahlen erzeugen. Bei der Division funktioniert die Erzeugung der Werte für den Divisor wie beschrieben per Generator, aber der Dividend verändert sich nicht um 1, sondern um a. Auch für den Dividend wäre die Nutzung eines Generators möglich, jedoch ist es einfacher, den Dividenden für jeden Divisor neu zu berechnen und für den Dividenden für jede Zerlegung zu prüfen, ob die nächste relevante Schnapszahl überschritten wurde. Ist das der Fall, wird die Mindestzifferzahl um 1 erhöht. Dass bei einer Veränderung des Divisors um 1 der Dividend zwei Schnapszahlen überschreitet, kann nicht vorkommen, weil der Abstand zwischen zwei aufeinanderfolgenden Schnapszahlen immer größer ist als die kleinere der beiden Schnapszahlen.[12]

Wenn eine Schnapszahl s bereits berechnet wurde, lässt sich die nächsthöhere Schnapszahl bestehend aus derselben Ziffer mit $10s + d$ berechnen.

1.3.7 Die vollständigen Algorithmen

Hier werden die Ausführungen aus den vorhergehenden Abschnitten zusammengetragen und die Algorithmen zum Analysieren einer Zahl und zum Durchsuchen der Zerlegungen in vollständiger Form dargestellt.

Zunächst der Algorithmus zum Durchsuchen der Zerlegungen, die vom weiter unten dargestellten Algorithmus generiert werden:

1. Eingabe: Zahl a, für die ein Term gesucht wird; Ziffer d; maximale Ziffernanzahl; Mindestzifferzahl m von a (es ist effizienter, die Mindestzifferzahl dem Algorithmus zu übergeben, da sie bei einem rekursiven Aufruf schon durch die Generierung der Zerlegungen bekannt ist)
2. Wenn maximale Ziffernanzahl kleiner als m ist: Gib zurück, dass kein Term gefunden wurde
3. Wenn die Zahl nur aus Ziffer d besteht: Gib die Zahl als gefundenen Term und ihre Ziffernanzahl zurück
4. Durchsuche den Cache nach a; und wenn sich die Zahl im Cache befindet:
    a. Setze den bisher besten Term und die bisher beste Anzahl an Ziffern auf das Ergebnis aus dem Cache. Speichere auch die alte maximale Anzahl und die alten Tabuzahlen t_a aus dem Cache.

[12] Beispiel mit Ziffer 4: Der Divisor ist kleiner oder gleich 44, der Abstand zu 444 beträgt 400, also müsste der Divisor mehr als 400 überspringen. Das geht aber nicht, weil er selbst kleiner oder gleich 44 ist.

   b. Wenn die Menge der alten Tabuzahlen (t_a) eine Teilmenge der aktuellen Tabuzahlen ist:[13]
      i. Falls der bisher beste Term nicht undefiniert ist:
         Gib den Term zurück, wenn die bisher beste Anzahl kleiner oder gleich der maximalen Ziffernanzahl ist, ansonsten gib zurück, dass kein Term gefunden wurde
      ii. Falls maximale Ziffernanzahl <= maximale Ziffernanzahl aus dem Cache:
         Gib zurück, dass kein Term gefunden wurde
   c. Wenn der bisher beste Term undefiniert ist und die alte maximale Ziffernanzahl kleiner ist als die aktuelle maximale Ziffernanzahl:
      Die bisher beste Anzahl ist im Moment folglich zu niedrig, weil sie den Wert *alte maximale Ziffernanzahl* + 1 hat (siehe Memoisation, 2. Element im Cache). Setze sie stattdessen auf *aktuelle maximale Ziffernanzahl* + 1. Außerdem kann die Menge der alten Tabuzahlen später (Schritt 12) nicht mehr genutzt werden (siehe Abschnitt 1.3.5). Setze t_a daher auf undefiniert.
   d. Wenn die bisher beste Anzahl größer ist als die aktuelle maximale Ziffernanzahl:
      Die Anzahl im Cache ist für die aktuelle maximale Ziffernanzahl zu groß. Der Term muss daher neu, ohne Hilfe des Caches, gesucht werden.
5. Wenn a nicht im Cache ist: Setze die bisher beste Anzahl auf *maximale Ziffernanzahl* + 1 und den bisher besten Term auf undefiniert
6. Füge a zur Menge der Tabuzahlen hinzu
7. Beginne mit Zerlegungen, die mindestens 2 Ziffern benötigen (aktuelle Mindestzifferzahl = 2)
8. Falls die aktuelle Mindestzifferzahl der Zerlegungen größer oder gleich der bisher besten gefundenen Anzahl ist, gehe zu Schritt 11 (es ist nicht mehr möglich, ein besseres Ergebnis zu finden)
9. Für jede der Zerlegungen mit der aktuellen Mindestzifferzahl:
   a. Falls beide Operanden der Zerlegung sich nicht in der Menge der Tabuzahlen befinden:
      i. Prüfe die Zerlegung (siehe Algorithmus weiter unten)
         1. Falls das Ergebnis durch die Zerlegung verbessert wurde und die neue beste Anzahl gleich der aktuellen Mindestzifferzahl ist, gehe zu Schritt 11 (das Ergebnis lässt sich nicht weiter optimieren)
10. Gehe zurück zu Schritt 8, aber betrachte nun Zerlegungen mit einer um 1 höheren Mindestzifferzahl
11. Entferne a aus der Menge der Tabuzahlen
12. Speichere das Ergebnis im Cache: gefundener Term, Anzahl der Ziffern, genutzte maximale Ziffernanzahl und Tabuzahlen. Wenn t_a undefiniert sind, werden als Tabuzahlen im Cache die aktuellen Tabuzahlen gespeichert. Ansonsten wird $t_a \cap$ *aktuelle Tabuzahlen* im Cache gespeichert.
13. Gib das Ergebnis (Term, Anzahl der Ziffern) zurück

Der Algorithmus zum Prüfen einer Zerlegung gehört zum obigen Algorithmus und verwendet die Variablen für die beste bisher gefundene Anzahl und den besten bisher gefundenen Term aus dem obigen Algorithmus:

1. Eingabe: Zerlegung
2. Ermittle den kleineren und den größeren der beiden Operanden der Zerlegung

[13] In der Menge der alten Tabuzahlen und in der der aktuellen Tabuzahlen befindet sich a – entgegen der vorhergehenden Beschreibung – nicht. Das spielt jedoch keine Rolle, weil a in beiden Mengen fehlt.

3. Beginne mit dem kleineren Operanden[14] und benutze obigen Algorithmus, um einem Term für diesen Operanden zu finden. Die maximale Ziffernanzahl ist *beste bisher gefundene Ziffernanzahl - Mindestzifferzahl des größeren Operanden* - 1.
4. Falls in Schritt 3 kein Term gefunden wurde, ist der Algorithmus beendet
5. Wurde ein Term gefunden, suche nun einen Term für den größeren Operanden mit maximaler Ziffernanzahl *beste bisher gefundene Ziffernanzahl - Ziffernanzahl des Terms für den kleineren Operanden* - 1
6. Falls in Schritt 5 ein Term gefunden wurde, aktualisiere die bisher beste Ziffernanzahl und den Term entsprechend

Im Folgenden ist der Algorithmus dargestellt, der eine Zahl analysiert. Es handelt sich um einen Generator, eine Anweisung „Erzeuge x“ bedeutet also, dass x „zurückgegeben“ wird, der Generator aber weitere Elemente erzeugt. Der Generator läuft theoretisch unendlich lange. Die Zerlegungen für Subtraktion und Division werden von weiteren Generatoren erzeugt, sie werden aber nicht weiter besprochen, da sie sich wie beschrieben verhalten: Sie generieren nacheinander Zerlegungen mit der jeweiligen Mindestzifferzahl, beginnend bei $(a + 1) - 1$ beziehungsweise $(2a)/2$.[15]

1. Eingabe: Zahl a
2. Erstelle leere Liste l, deren i-tes Element später eine Liste aller Zerlegungen mit Addition und Multiplikation mit Mindestzifferzahl i enthält
3. MULTIPLIKATION: Für jeden Teiler $1 < x < \sqrt{a}$:
    a. Berechne die Mindestzifferzahl des Teilers
    b. Berechne aus dem Teiler den zweiten Faktor
    c. Berechne die Mindestzifferzahl des zweiten Faktors
    d. Setze s auf die Summe der Mindestzifferzahlen der zwei Faktoren
    e. Füge die Zerlegung zum s-ten Element von l hinzu; wenn l nicht lang genug ist, füge vorher entsprechend viele leere Listen an l an
4. SUMME: Für alle Zerlegungen (Erzeugung durch zwei Generatoren für die Operanden (3. Generator in Fußnote 15)):
    a. Füge Zerlegung zur Liste hinzu, wie 3e (mit s = Summe der Mindestzifferzahlen der Summanden)
5. Initialisiere Generatoren für Zerlegungen mit Subtraktion und Division, wie beschrieben (2. Generator in Fußnote 15)
6. Hole nächstes Element aus Subtraktions-Zerlegungs-Generator: Speichere die nächste Zerlegung für Subtraktion, nebst Mindestzifferzahl
7. Hole nächstes Element aus Divisions-Zerlegungs-Generator: Speichere die nächste Zerlegung für Division, nebst Mindestzifferzahl
8. ZERLEGUNGEN GENERIEREN:
    a. Setze aktuelle Mindestzifferzahl m auf 2
    b. Wenn es in l ein m-tes Element gibt:

[14] Es ist sinnvoll, den kleineren Operanden zuerst zu betrachten, da dieser die größer maximale Ziffernanzahl hat und kleinere Zahlen grundsätzlich leichter zu finden sind als große – ein Term für den größeren Operanden muss auf diese Weise vielleicht gar nicht betrachtet werden, wenn für den kleineren Operanden kein Term gefunden wurde.

[15] Achtung, es gibt insgesamt drei Arten von Generatoren:
1. Der hier vorgestellte zum Analysieren einer Zahl, dieser nutzt den 2. Generator;
2. Die Generatoren zum Erzeugen der Zerlegungen für Subtraktion und Division, diese nutzen den 3. Generator je zweimal (für beide Operanden);
3. Ein Generator, der Werte für Operanden mit ihrer Mindestzifferzahl effizient generiert. Dieser wird zum Erzeugen der Zerlegungen von Subtraktion, Division und Addition benutzt.

   i. Erzeuge die Elemente des m-ten Elements von l
c. Wenn die Mindestzifferzahl des nächsten Elements im Subtraktions-Zerlegungs-Generator gleich m ist:
   i. Erzeuge solange Elemente aus dem Subtraktions-Zerlegungs-Generator, bis die Mindestzifferzahl eines Elements ungleich m ist. Generiere dieses letzte Element nicht, sondern speichere es als nächstes Element.
d. Tue das gleiche wie in 8c, nur mit Divisions-Zerlegungs-Generator
e. Erhöhe m um 1
f. Gehe zu Schritt 8b

1.4 Einfache und erweiterte Termbildung: Alle Terme bis zu einer bestimmten Ziffernanzahl finden

Ein Term mit Ziffernanzahl z kann aus zwei Termen mit niedrigeren Ziffernanzahlen, die in der Summe z ergeben, gebildet werden, indem die beiden Terme durch einen Operator (+-*/) verknüpft werden. Die in diesem Abschnitt vorgestellte Termbildung nutzt dies, um alle Terme bis zu einer vorgegebenen Ziffernanzahl zu finden. Insbesondere ist es so möglich, die Tiefensuche zu beschleunigen, auch nichtganzzahlige Teilterme zu berücksichtigen und Fakultäten und Potenzen einzubeziehen.

Terme werden mit aufsteigender Ziffernanzahl gebildet. Mit Ziffernanzahl 1 gibt es mit Grundrechenarten nur einen Term, nämlich d. Terme mit zwei Ziffern werden danach gebildet, indem zwei Terme mit je einer Ziffer durch einen Operator verknüpft werden. Zusätzlich gibt es den Term dd mit zwei Ziffern, der nicht durch Kombination zweier Terme entstehen kann. Alle Terme, die mit drei Ziffern gebildet werden können, erhält man durch Kombination der Terme mit einer Ziffer und der Terme mit zwei Ziffern. Die Schnapszahl ddd wird wieder gesondert gebildet. Weil Grundrechenarten und Potenzen zweistellige Verknüpfungen darstellen, werden immer zwei Terme zu einem dritten verknüpft. Ab Termen mit vier Ziffern gibt es mehrere Möglichkeiten, einen neuen Term aus bereits gefundenen zu bilden: Entweder aus einem Term mit einer und einem mit drei Ziffern oder aus zwei Termen mit je zwei Ziffern. Um bereits erzeugte Terme verknüpfen zu können, werden alle gebildeten Terme in einem Array a gespeichert, dessen i-tes Element eine Menge ist, die alle Terme bestehend aus i Ziffern enthält. Das Array hat kein nulltes Element, und das erste Element enthält – zumindest, wenn die Operationen auf Grundrechenarten beschränkt sind – nur den Term d.

Dieses Verfahren heißt „einfache Termbildung“, weil nur einfache Termen erzeugt werden. Einfache Terme sind Terme, die nur Grundrechenarten verwenden. In Abschnitt 1.4.2 wird die erweiterte Termbildung beschrieben. Diese erzeugt Terme, die auch Potenzen und Fakultäten verwenden.

Um neue Terme mit Ziffernanzahl c bilden zu können, muss zunächst berechnet werden, Terme welcher Anzahlen kombiniert werden müssen, um als Summe der beiden Anzahlen die neue Anzahl zu erhalten. c ist größer als 1, weil Terme mit nur einer Ziffer nicht aus Kombinationen anderer Terme gebildet werden können. Zur Bildung neuer Terme sind immer Terme mit Ziffernanzahl 1 relevant. Daher werden zunächst neue Terme aus den Termen mit Anzahl 1 und Anzahl $c-1$ gebildet. Darauf folgen Kombinationen aus Termen mit Anzahl 2 und Anzahl $c-2$ und so weiter, bis die erste der beiden Anzahlen größer als $\left\lfloor \frac{c}{2} \right\rfloor$ ist, weil dann alle relevanten Kombinationen berücksichtigt wurden. Wurden beispielsweise Kombinationen mit Anzahlen 4 und 5 betrachtet, müssen keine Kombinationen mit 5 und 4 mehr betrachtet werden. Das liegt daran, dass bei der Termverknüpfung die Reihenfolge der beiden Terme nicht relevant ist, siehe unten.

Für die beiden Anzahlen erhält man die entsprechenden Mengen mit Termen der jeweiligen Anzahl aus a. Daraufhin werden alle Kombinationen aus den beiden Mengen mithilfe des kartesischen

Produkts betrachtet. Es müssen nicht alle Permutationen betrachtet werden, weil es bei der nun folgenden Verknüpfung keine Rolle spielt, in welcher Reihenfolge zwei Terme auftreten. Jede Kombination besteht aus zwei Termen t_1 und t_2. Wie zwei Terme mit Grundrechenarten verknüpft werden können, ist in folgender Aufzählung dargestellt:

1. Addition und Multiplikation sind kommutativ. Daher werden beide Terme addiert bzw. multipliziert: $t_1 + t_2$ bzw. $t_1 \cdot t_2$.
2. Für die Division gibt es zwei Möglichkeiten: t_1/t_2 oder t_2/t_1. Wenn keine nicht-ganzzahligen Teilterme zugelassen sein sollen, muss zusätzlich für jede der beiden Möglichkeiten überprüft werden, ob die Zahlen, zu denen die Terme auswerten, teilbar sind. Es kann sein, dass beide Möglichkeiten valide sind. In diesem Fall würde $t_1 = t_2$ gelten und daher reicht es aus, nur eine der beiden Verknüpfungen zu betrachten (t_1/t_2 wäre 1, und für 1 ist die beste Darstellung immer d/d mit zwei Ziffern,[16] also müsste dieser Fall theoretisch gar nicht betrachtet werden, außer es handelt sich um die Darstellung d/d).
   Sind nicht-ganzzahlige Teilterme erlaubt, ist es irrelevant, ob beide Terme teilbar sind oder nicht. Der Quotient darf allerdings nicht als Gleitkommazahl abgespeichert werden, weil ansonsten später nicht mehr sicher festgestellt werden kann, ob ein Term eine Ganzzahl darstellt oder nicht (siehe unten). Daher ist es sinnvoll, stattdessen mit Brüchen weiterzurechnen, und zwar auch bei allen Termen, die aus dieser Division gebildet werden. Sollte ein Term zu einem Bruch auswerten, wird überprüft, ob der (gekürzte) Bruch den Nenner 1 hat. Wenn ja, handelt es sich um eine Ganzzahl.
3. Für die Subtraktion gibt es wieder zwei Möglichkeiten: $t_1 - t_2$ oder $t_2 - t_1$. Erstere Möglichkeit wird nur betrachtet, wenn $t_1 > t_2$, weil die Differenz ansonsten negativ wäre.[17] Trifft das nicht zu, wird mit der zweiten Möglichkeit ein neuer Term gebildet, allerdings nur, wenn $t_1 \neq t_2$, weil die Differenz ansonsten 0 wäre.[18]

Eine weitere Optimierung ist vorstellbar, indem beispielsweis vor der Verknüpfung eines Terms durch Division überprüft wird, dass die Terme der Operanden nicht durch eine Division gebildet wurden. Dann nämlich existiert eine andere Schreibweise mit Multiplikation: `a/(b/c)=(ab)/c` und `(a/b)/c=a/(bc)`. Dasselbe Prinzip lässt sich auf Subtraktion übertragen. Beides sind eher theoretische Überlegungen, praktisch beansprucht die Überprüfung mehr Zeit und wurde daher nicht implementiert.

Die Terme, die durch diese Verknüpfungen entstanden sind, werden in der entsprechenden Menge in a gespeichert, sodass sie für nachfolgende Termverknüpfungen verfügbar sind. In der Menge befindet sich zu jedem Term auch die Zahl, zu der der Term auswertet, sodass diese Zahl nicht für jede Verknüpfung neu berechnet werden muss.

Ein neuer Term wird nicht nur in der entsprechenden Menge, sondern auch in einem Dictionary gespeichert, sofern der Term zu einer Ganzzahl auswertet. Bei diesem Dictionary handelt es sich um den Cache, der auch für die Tiefensuche verwendet wird, allerdings werden die maximale Ziffernanzahl und die Tabuzahlen nicht gesetzt, weil der gefundene Term mit Sicherheit optimal ist. Im Dictionary kann der Term zu einer Zahl dann einfach gefunden werden. Die Größe der Zahlen, die in den Cache geschrieben werden, kann begrenzt werden, um den Zugriff auf Elemente darin zu beschleunigen.

[16] Außer für Ziffer 1.

[17] Es ist nicht nötig, dass ein Term zu einer negativen Zahl auswertet. Negative Zahlen lassen sich immer mit genauso vielen Ziffern darstellen wie ihr absoluter Wert: Es reicht, ein Minus vor den Term für den absoluten Wert zu schreiben.

[18] Ein Term, der zu 0 auswertet, ist nur relevant, um die Zahl 0 selbst darzustellen (der beste Term wäre $d - d$). Ein solcher Term würde bei Addition und Subtraktion nur bewirken, dass mehr Ziffern benötigt werden, man kann nicht durch diesen Term teilen und als Dividend oder Faktor eignet er sich auch nicht.

Diese Grenze sollte allerdings beispielsweise nicht auf 2980 gesetzt werden, wenn 2980 die größte der Zahlen ist, für die Terme gesucht werden, denn Teilterme können zu größeren Zahlen als der gesuchten Zahl auswerten.

Zudem werden in einer Menge alle Zahlen gespeichert, zu denen bereits gebildete Terme auswerten. Diese Menge ist geeignet, um sicherzustellen, dass kein zweiter Term für eine Zahl gespeichert wird. Wurde bereits ein Term mit Ziffernanzahl 2 gefunden, kann es sein, dass auch mit Ziffernanzahl 5 ein Term gefunden wird, der zur gleichen Zahl auswertet. Da sich die Zahl aber bereits im Dictionary befindet, handelt es sich bei dem Term um eine gleich gute oder – in diesem Fall – schlechtere Darstellung und der Term wird nicht gespeichert.

1.4.1 Beschleunigung der Tiefensuche durch einfache und erweiterte Termbildung

Das im vorhergehenden Abschnitt vorgestellte Verfahren eignet sich nur bedingt für eine effiziente Suche nach Termen. Für Terme mit höherer Ziffernanzahl ist eine deutlich schlechtere Laufzeit als mit Tiefensuche zu erwarten, da die Anzahl der Terme mit einer bestimmten Ziffernanzahl stark zunimmt, wie sich folgendem Diagramm entnehmen lässt.

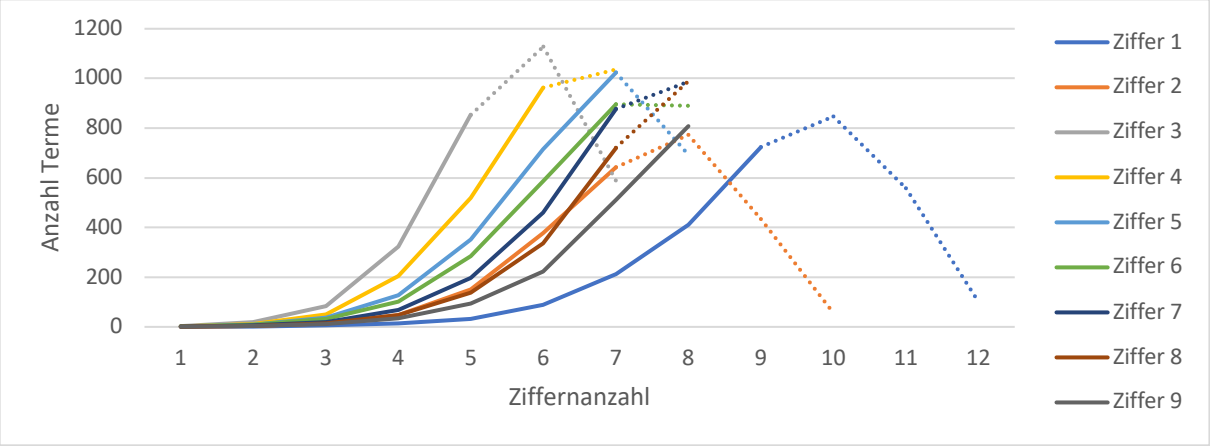

Abb. 2. Dargestellt ist die Anzahl erweiterter Terme (also Terme mit Grundrechenarten, Potenzen und Fakultäten), die durch Termbildung erzeugt wurden. Die Ziffernanzahl nimmt nur aufgrund der in Abschnitt 1.4.4 beschriebenen Optimierung wieder ab. Der Anstieg zu Beginn wird von der Optimierung jedoch nicht beeinflusst. Gut zu erkennen ist auch, dass es vor allem mit Ziffer 1 viele Terme mit größerer Anzahl gibt. Das ist auch der Grund dafür, dass für jede Ziffer Terme bis zu einer anderen Ziffernanzahl berechnet wurden, je nach Anzahl der Ziffern: Für Ziffer 3 werden schon zu Beginn sehr viele Terme gefunden, während für Ziffer 1 bis zu einer größeren Ziffernanzahl Terme gebildet werden müssen, um eine ähnliche Gesamtanzahl zu erhalten.

Eine Beschleunigung der Tiefensuche wird erreicht, indem vor der Durchführung der Tiefensuchen die einfache oder erweiterte Termbildung Terme bis zu einer vorgegebenen Anzahl c erzeugt. Für die dann folgenden Tiefensuchen stehen im Cache bereits Terme bis zu dieser Anzahl. Zudem ist bekannt, dass mit der Tiefensuche keine Terme mit einer Ziffernanzahl kleiner oder gleich c gefunden werden können, weil Terme für diese Anzahlen bereits im Cache stehen. Daher wird im Algorithmus auf Seite 11 für den Fall, dass im Cache kein Term gefunden wurde, überprüft, ob die maximale Ziffernanzahl $\leq c$ ist. Ist das der Fall, kann direkt zurückgegeben werden, dass kein Term gefunden wurde.

1.4.2 Potenzen und Fakultäten

Wie bereits beschrieben ist die Tiefensuche für Potenzen und Fakultäten ungeeignet. Das liegt daran, dass die Berechnung der Mindestzifferzahl sich durch Potenzen stark verändert und für Fakultäten die Mindestzifferzahl nicht mehr berechenbar ist, sondern viele Zahlen potenziell mit nur einer Ziffer oder zumindest sehr wenigen Ziffern dargestellt werden könnten. Stattdessen wird erweiterte Termbildung eingesetzt, um alle erweiterten Terme bis zu einer vorgegebenen Ziffernanzahl zu berechnen. Erweiterte Termbildung ist eine für Potenzen und Fakultäten erweiterte einfache Termbildung. Dabei wird

ein weiterer Cache verwendet, der nur erweiterte Terme speichert. Der Cache hat im Gegensatz zum Cache für einfache Terme keine maximalen Ziffernanzahlen und Tabuzahlen, weil ein mit diesem Verfahren gefundener Term immer optimal ist. Es werden außerdem drei Arrays gespeichert (dies entspricht dem Array a für einfache Termbildung), in denen sich Mengen mit den Termen mit einer bestimmten Ziffernanzahl befinden:

1. Array a_1 enthält alle einfachen Terme, für die auch ein erweiterter Term nicht weniger Ziffern bringt.
2. Array a_2 enthält alle einfachen Terme, für die bessere erweiterte Terme im dritten Array existieren („schlechte einfache Terme“).
3. Array a_3 enthält alle erweiterten Terme.

Einfache und erweiterte Terme werden mit erweiterter Termbildung gleichzeitig erzeugt. Die Kombination von Termen aus zwei Mengen funktioniert etwas anders als nur mit Grundrechenarten. Für zwei Anzahlen c_1 und c_2 (es sollen also neue Terme mit Anzahl $c_1 + c_2$ gefunden werden) kann nicht mehr mit zwei Mengen das kartesische Produkt gebildet werden, sondern es gibt verschiedene Möglichkeiten, die Terme aus nun insgesamt sechs Mengen (pro Anzahl 3 Mengen, aus jedem Array eine) zu neuen Termen zu kombinieren, um entweder neue einfache oder neue erweiterte Terme zu erhalten. Zunächst sollen für die beiden Anzahlen nur neue einfache Terme erzeugt werden. Die zu c_1 gehörende Menge, die nur einfache Terme mit Länge c_1 enthält, ist $a_1[c_1] \cup a_2[c_1]$ (der [x]-Operator liefert das x-te Element aus dem Array). Es werden also die beiden Mengen vereinigt, die einfache Terme enthalten. Die Menge für c_2 wird nach demselben Prinzip gebildet, nur mit c_2 statt c_1. Es ergeben sich neue einfache Terme durch $(a_1[c_1] \cup a_2[c_1]) \times (a_1[c_2] \cup a_2[c_2])$. Danach sollen erweiterte Terme aus Termen der gegebenen Anzahlen entstehen. Dafür gibt es drei Kombinationsmöglichkeiten:

- $a_3[c_1] \times a_3[c_2]$
- $a_3[c_1] \times a_1[c_2]$
- $a_1[c_1] \times a_3[c_2]$

Die Mengen werden also so kombiniert, dass es mindestens eine Menge aus a_3 gibt, da dieses Array erweiterte Terme enthält und dadurch auch die Kombinationen erweiterte Terme werden. Eine Menge aus a_2 kommt nicht vor, weil sie Terme enthält, mit denen keine erweiterten Terme gebildet werden sollten, da es für die durch die Terme dargestellten Zahlen bessere erweiterte Terme gibt. Es wird nicht einfach $a_3[c_1] \cup a_1[c_1]$ mit $a_3[c_2] \cup a_1[c_2]$ kombiniert, weil auf diese Weise auch neue einfache Terme entstehen würden.

Für einen neuen Term wird wie bei Grundrechenarten überprüft, ob die Zahl, zu der der Term auswertet, bereits gefunden wurde. Ist der Term ein erweiterter, müssen beide Caches nach der gefundenen Zahl durchsucht werden. Taucht die Zahl nicht auf, ist der Term eine neue Darstellung für die Zahl und wird im Cache für erweiterte Terme sowie in der Menge mit erweiterten Termen gespeichert. Handelt es sich um einen einfachen Term, gibt es mehrere Möglichkeiten. Befindet sich die Zahl nicht im Cache für einfache Terme, wird der neue Term auf jeden Fall im Cache für einfache Terme gespeichert. Wenn sich die Zahl im Cache für erweiterte Terme befindet, bedeutet dies, dass für diese Zahl ein besserer erweiterter Term existiert. Dann gehört der Term ins Array a_2, ansonsten in a_1.

Für Potenzen gibt es zwei Bildungsmöglichkeiten aus zwei Termen, t_1^t_2 und t_2^t_1. Im Gegensatz dazu handelt es sich bei der Fakultät um einen einstelligen Operator, weshalb Terme mit Fakultäten nicht aus der Kombination zweier Terme gebildet werden können. Stattdessen wird immer, wenn ein Term für eine Zahl, die sich noch nicht im Cache befindet, gefunden wurde, versucht, die Fakultät dieser Zahl auch hinzuzufügen. Allerdings muss auch für die gefundene Fakultät versucht werden, wiederum die Fakultät zu bilden (beispielsweise `((3!)!)=720`). Das würde zu einer theoretisch unendlich langen Rekursion führen, die verhindert werden muss. Am einfachsten ist es festzulegen, dass für eine Zahl keine

Fakultät mehr gebildet wird, sofern sie über einer gewissen Grenze liegt. Dazu muss aber eine passende Grenze ermittelt werden, die möglichst keine Fakultäten ausschließt, die eigentlich für einen Term benötigt würden. Dazu wird zuerst betrachtet, warum es überhaupt sinnvoll sein könnte, sehr große Fakultäten zuzulassen: Zahlen werden mit den gegebenen Operationen nur durch Subtraktion oder Division kleiner. Eine sehr große Zahl, die durch wenige Ziffern mit einer Fakultät dargestellt wird, wird also nur zu einer vernünftig kleinen Zahl, wenn eine ähnlich große Zahl subtrahiert wird oder die Fakultät durch eine andere Zahl dividiert wird. Bei Subtraktion ist es vorstellbar, als Subtrahenden eine weitere Fakultät oder eine Potenz zu benutzen. Da der Subtrahend jedoch ähnlich groß wie der Minuend sein sollte, benötigt man mit einer Potenz als Subtrahenden unnötig viele Ziffern. Eine Subtraktion mit einer anderen Fakultät ist jedoch vorstellbar, wobei berücksichtigt werden muss, dass diese Differenz mit steigenden Fakultäten merklich zunimmt (beispielsweise 15!-14! ≈ 1.2204961e+12). Die Division scheint daher grundsätzlich besser geeignet zu sein, doch auch mit Potenzen ist zu erwarten, dass für die Darstellung der Potenz einige Ziffern benötigt werden, zumal auch darauf geachtet werden muss, dass eine Ganzzahl entsteht, die Fakultät also durch den Divisor teilbar ist. Die Division zweier Fakultäten, wobei der Dividend größer als der Divisor ist, ist jedoch immer möglich:

$$\frac{n!}{m!} = n \cdot (n-1) \cdot \ldots \cdot (m+1), n > m$$

Es ist dabei auch möglich, die Operanden leicht zu verändern. 121 lässt sich beispielsweise so darstellen: $\frac{(3!)!+3!}{3!}$. Sehr groß werden die Fakultäten in diesem Beispiel nicht, aber auch andere Beispiele dieser Art mit größeren Fakultäten sind denkbar.

Um herauszufinden, auf welche Höhe Fakultäten beschränkt werden sollten, werden zunächst erweiterte Terme mit Grenze 100! mit erweiterter Termbildung erzeugt. Diese Grenze wurde gewählt, da mit 100! oder mehr vermutlich nicht mehr viele Terme sinnvoll gebildet werden können. 100!/98!=9900, und dieses Ergebnis ist immer noch weit entfernt von den Zielzahlen unter 2981. Allein für die Darstellung von 100 und 98 werden einige Ziffern benötigt. Zur Durchführung des Tests siehe Abschnitt 2.7.3. Betrachtet wurde, welche Fakultäten von Zahlen kleiner als 10.000 verwendet werden. Der Test ergab, dass, ein paar Ausreißer nicht berücksichtigt, maximal 68! verwendet wird, und zwar zur Darstellung von 4556 mit Ziffer 6 (wie erwartet, werden zwei Fakultäten dividiert; der Term lautet `(((66+((6+6)/6))!)/(66!))`). Aus diesem Grund wurde 70! als Maximum für Fakultäten festgelegt. Die Ausreißer treten nur mit den Ziffern 7, 8 und 9 auf, wobei immer zwei große Fakultäten dividiert werden:

1. Ziffer 7: 77! und 79!; 6162=`(((7+((7!)/(77-7)))!)/(77!))`
2. Ziffer 8: 88! und 90!; 8010=`((((((88-8)/8)!)/(8!))!)/(88!))`
3. Ziffer 9: 81! und 83!; 6806=`((((9*9)+((9+9)/9))!)/((9*9)!))`

Die Zahlen liegen alle über 5000 und werden aufgrund ihrer Größe und hohen Ziffernanzahl eher nicht für die Zahlen 2019, …, 2980 benötigt. Sollte eine höhere Fakultätsgrenze für bestimmte Zahlen nötig sein, lässt sich diese im Programm einfach ändern. Um sicherzugehen, wurde der Test wiederholt, aber mit maximaler Fakultät 200!. Es zeigt sich fast die gleiche Verwendung von Fakultäten, lediglich für die Ziffer 5 werden Terme gefunden, die 120! Und 122! verwenden. In allen Fällen außer einem handelt es sich dabei um Terme, die wie die Terme mit maximaler Fakultät 100! auch 6 Ziffern besitzen, es entsteht durch die höhere Fakultät also keine Verbesserung. In einem Term wird 122! verwendet, und dieser Term ist als einziges besser. Es handelt sich um 7381, dargestellt durch `((((5!)+((5+5)/5))!)/(((5!)!)+((5!)!)))`. Man kann vermuten, dass mit steigender Fakultätsgrenze mehr Terme der Form `((d!)!)` verwendet werden.

Eine Grenze für den Exponenten von Potenzen festzusetzen, ist ebenfalls sinnvoll. Zum einen ist eine Grenze sinnvoll, damit das Programm schneller läuft, zum anderen ist es sinnvoll, um nicht mit unnötig vielen großen Zahlen rechnen zu müssen, die letztendlich sowieso nicht in den Termen auftauchen – was das Programm letztendlich auch beschleunigt. Die Grenze für den Exponenten wurde auf 100 festgelegt. 2^100 ergibt bereits 1267650600228229401496703205376, und mit größeren Zahlen lässt sich wohl kaum etwas anfangen, zumal andere Zahlen nötig sind, um aus solch großen Zahlen wieder kleinere zu erhalten.

1.4.3 Potenzen und Fakultäten – Algorithmus zum Finden des Terms und Optimierung I

Wurden mit dem beschriebenen Verfahren erweiterte Terme bis zu einer bestimmten Ziffernanzahl gesucht, ist es möglich, dass sich eine gesuchte Zahl schon im Cache für erweiterte Terme befindet. In diesem Fall muss nicht weitergesucht werden und der beste Term wurde bereits gefunden. Ansonsten wurde die erweiterte Termbildung offenbar nicht bis zur erforderlichen Ziffernanzahl durchgeführt. Betrachtet man beispielsweise den Term `2030=((6!)+(((((6!)+(6^6))/6)/6)-6))`, bemerkt man aber, dass nur Teilterme mit Potenzen und Fakultäten mit einer maximalen Anzahl von 2 Ziffern vorkommen: `((6!)+(((((6!)+(6^6))/6)/6)-6))`. Die drei blau markierten Teilterme sind erweiterte Terme, da sie Potenzen und Fakultäten enthalten. Der dritte mit den meisten Ziffern besteht aus 2 Ziffern. Das bedeutet, dass zum Finden dieses Terms nur erweiterte Terme bis zur Ziffernanzahl 2 erzeugt werden müssten. Natürlich lässt sich diese Grenze nicht ermitteln, sondern dies stellt eine Möglichkeit dar, schneller einen Term zu finden, auch wenn dieser vielleicht nicht optimal ist. Nachdem erweiterte Terme bis zu einer gewissen Anzahl gesucht wurden, ist es möglich, weiterzusuchen, allerdings indem nur noch Grundrechenarten verwendet werden. Auf die erweiterte Termbildung folgt also eine einfache Termbildung. Auch während der einfachen Termbildung dürfen erweiterte Terme entstehen, wenn zuvor eine erweiterte Termbildung durchgeführt wurde; die einfache Termbildung bildet lediglich keine neuen Potenzen oder Fakultäten. Wird nach der erweiterten Termbildung bis zur Ziffernanzahl 2 eine einfache Termbildung ab Anzahl 3 bis Anzahl 5 durchgeführt, lässt sich obiger Term aus folgenden Teiltermen bilden, wobei alle Teilterme durch die Suche gefunden wurden (ein Teilterm in rot, der andere in grün): `6!-6+(((6!+6^6)/6)/6)`. Es ist zu beachten, dass Aufteilungen wie `6!+(((6!+6^6)/6)/6)-6` oder `6!+(((6!+6^6)/6)/6)-6` nicht möglich sind, weil dann einfache Terme bis zur Ziffernanzahl 6 gesucht worden sein müssten.

Diese häufige Eigenschaft erweiterter Terme macht sich das Programm zunutze, um erweiterte Terme auch zu finden, wenn sie nicht im Cache gespeichert sind. Nach der Termbildung kann ein erweiterter Term für eine Zahl gesucht werden. Dabei werden alle erzeugten erweiterten Terme mit aufsteigender Anzahl durchsucht. Dazu wird das Array a_3 verwendet, das alle erweiterten Terme enthält. Es werden zuerst alle erweiterten Terme mit Ziffernanzahl 1, dann alle mit 2 und so weiter durchgegangen. Im Beispiel reicht es theoretisch schon, erweiterte Terme bis zur Anzahl 2 zu durchsuchen, denn unter den Termen mit Ziffernanzahl 2 befindet sich `714=6!-6`. Für diesen Teilterm existieren verschiedene Möglichkeiten zum Bilden der gesuchten Zahl. Um die Zahl zu bilden, werden nur Grundrechenarten betrachtet (wenn die Zahl als Potenz oder Fakultät darstellbar ist, wird das separat betrachtet, siehe nächsten Absatz). Durch Multiplikation ist es nicht möglich, 2030 zu erhalten, da 2030 mod 714 = 602. Subtraktion wäre allerdings möglich: 2030+714 = Minuend = 2744. Befindet sich 2774 in einem der zwei Caches (wenn sie sich in beiden Caches befindet, wird der Cache für erweiterte Terme vorgezogen), wäre damit ein neuer möglicher Term gefunden. Ob der Term auch besser ist, entscheidet, ob die Summe der Ziffernanzahl beider Terme geringer ist als die bisher gefundene Anzahl. Ist er besser, wird der Term mit Anzahl als bisher bestes Ergebnis gespeichert. Es müssen keine erweiterten Terme aus a_3 mehr betrachtet werden, sobald deren Ziffernanzahl um 1 kleiner als die bisher beste gefundene Anzahl ist. Dann kann nur noch eine gleich gute (wenn der zweite Term nur 1 Ziffer hat), nicht aber eine bessere Lösung gefunden werden. Zurück zu möglichen Kombinationen: Weil 714 kleiner als 2030 ist, kommen auch Addition und Division infrage: 2030-714 = Summand = 1316 und 2030*714 =

Dividend = 1 449 420. Wäre 714 größer als die Zahl, für die ein Term gesucht wird, wäre ebenfalls Division möglich, aber mit Dividend und Divisor getauscht, wobei *Operand* (hier 714) mod *gesuchte Zahl* (kleiner als 714) = 0 gelten müsste. Außerdem gäbe es eine zweite Möglichkeit für Subtraktion, nämlich *Operand* − *Subtrahend* = *gesuchte Zahl*.

Es gibt noch eine weitere Möglichkeit, die Ziffernanzahl, bis zu der Terme gebildet werden, zu begrenzen. Lässt sich die gesuchte Zahl x weder als Potenz noch als Fakultät darstellen (es gilt also nicht $n^m = x$ oder $n! = x$, wobei $n, m \in \mathbb{N}$), dann müssen nur Terme bis *Ziffernanzahl des Terms* − 1 gebildet werden. Zu Beginn ist natürlich nicht bekannt, welche Ziffernanzahl ein Term für die gesuchte Zahl haben wird. Dennoch ist folgende Optimierung möglich: Zunächst wird ein erweiterter Term gefunden, indem zuerst erweiterte Terme bis zu einer niedrigen Anzahl (beispielsweise 3) gebildet werden und dann alle diese erweiterten Terme wie bereits beschrieben durchgegangen werden, um durch Kombination mit einem anderen Term die gesuchte Zahl zu erhalten. Der dadurch gefundene Term ist vielleicht nicht optimal, aber es ist zumindest eine maximale Ziffernanzahl bekannt. Dadurch können nun nochmals erweiterte Terme gebildet werden, und zwar bis zur Anzahl *eben gefundene Ziffernanzahl* - 1. Mit dieser Anzahl wird erneut ein Term gesucht, wieder indem die erweiterten Terme durchgegangen werden. Der daraus resultierende Term ist sicher optimal. Dasselbe funktioniert übrigens auch mit einem eventuell schon per Tiefensuche gefundenen einfachen Term, der nun verbessert wird.

1.4.4 Optimierung II: Einschränkung der Größe der Zahlen für Ziffernanzahlen

Wie groß die maximale Zahl ist, für die ein Term gefunden werden soll, ist bekannt – in den Beispielen ist es 2980. Wenn außerdem bekannt ist, welche Ziffernanzahl maximal zu erwarten ist, lässt sich die Größe der Zahlen, die während der einfachen oder erweiterten Termbildung für eine bestimmte Anzahl gefunden werden darf, einschränken. Als einzige Operationen, um aus einer großen Zahl eine kleinere zu erhalten, kommen Division und Subtraktion infrage. Division ist dabei besser geeignet, weil durch Division eine große Zahl „schneller“ kleiner wird, d.h. mithilfe eines zweiten Operanden, der kleiner ist als mit Subtraktion. Werden beispielsweise Terme bis zur Ziffernanzahl 6 gebildet (es spielt keine Rolle, ob einfache oder erweiterte Termbildung verwendet wird) und es wird erwartet, dass die Terme maximal 7 Ziffern benötigen, kann die Größe der Zahlen ab Termen mit Zifferanzahl 4 (also die aufgerundete Hälfte) eingeschränkt werden. Die Terme mit Zifferanzahl 4 dürfen nur noch zu Zahlen auswerten, die kleiner als oder gleich groß wie 2980**größte gefundene Zahl mit Ziffernanzahl 3* sind. Jede größere Zahl ist nämlich zu groß, um mit nur 7 Ziffern 2980 zu erhalten. Eine Zahl mit Ziffernanzahl 4 hat maximal drei Ziffern „zur Verfügung“, um 2980 zu erreichen, weil die maximale Anzahl 7 beträgt. Das wird am schnellsten durch Division erreicht, aber für alle Zahlen, die größer als die genannte Grenze sind, gibt es keinen Divisor, der groß genug ist, um 2980 zu erreichen. Diese Optimierung ist der Grund dafür, dass die Anzahl der Terme in obigem Diagramm wieder abnimmt. Die Terme für das Diagramm wurden übrigens mit unterschiedlicher maximaler Ziffernanzahl gesucht, was zur Folge hat, dass die Anzahl der Terme an unterschiedlichen Stellen wieder abnimmt.

2 Umsetzung

Das Programm ist in Python geschrieben. Aufgrund der Verwendung neuer Sprachelemente sind die Skripte nur mit Python in Version 3.8 kompatibel.

Um Wiederholungen zu vermeiden, werden in diesem Abschnitt Struktur und Teile der Funktionsweise der Skripte beschrieben. Detailreiche Beschreibungen finden sich zusammen mit dem Quellcode im letzten Abschnitt.

Die Klasse `BirthdayNumberFinder` aus aufgabe2.py stellt alle Methoden bereit, um einfache und erweiterte Terme zu finden. Pro Ziffer wird eine Instanz des `BirthdayNumberFinders` benötigt. Dem Konstruktor wird daher die Ziffer übergeben, für die Terme gefunden werden sollen; die Ziffer wird im

Attribut `digit` gespeichert.[19] Außerdem wird dem Konstruktor die größte Zahl, die später berechnet werden wird, übergeben. Für die Beispiele ist dies 2980. Diese Zahl im Attribut `max_number` wird von einfacher und erweiterter Termbildung für die zweite Optimierung (Abschnitt 1.4.4) genutzt. Der Konstruktor erhält auch die Ziffernanzahl, die gefundene einfache Terme maximal haben werden. Die Anzahl darf höher als nötig sein, wenn sie nicht genau bekannt ist – dies ist der Fall, wenn noch keine Terme berechnet wurden. Wenn festgestellt wird, dass die Ziffernanzahl der gefundenen Terme höher liegt als die übergebene maximale Ziffernanzahl, sollte die Suche mit einer besseren Anzahl wiederholt werden, weil es dann möglich ist, dass die Ergebnisse fehlerhaft sind. Die maximale Ziffernanzahl für einfache Terme wird dazu verwendet, den Cache für einfache Terme (`BirthdayNumberFinder.found_simple_terms`, ein `dict`) mit Schnapszahlen bestehend aus `digit` bis zur vorgegebenen maximalen Anzahl zu befüllen. Dies ist ein Unterschied zum in der Lösungsidee vorgestellten Algorithmus: Die Schnapszahlen werden schon vorher in den Cache geschrieben, und bei der Tiefensuche wird nicht jedes Mal überprüft, ob die Zahl eine Schnapszahl bestehend aus `digit` ist, weil es effizienter ist, die Schnapszahlen vorher zu generieren als für jede Zahl zu prüfen, ob es sich um eine Schnapszahl bestehend aus `digit` handelt. Die maximale Ziffernanzahl für einfache Terme wird später für die zweite Optimierung der Termbildung (Abschnitt 1.4.4) benötigt und daher im Attribut `max_simple_count` gespeichert.

2.1 Caches

Ein `BirthdayNumberFinder` besitzt zwei Attribute zur Implementierung der Caches. Den folgenden beiden `dicts` ist gemein, dass sie als Schlüssel die Zahlen besitzen, für die ein Term gefunden wurde:

- `found_simple_terms`, um alle gefundenen einfachen Terme zu speichern. Die Werte sind Tupel, die die auf Seite 8 dargestellten Elemente enthalten: Der gefundene Term (oder `None`), die Ziffernanzahl (`int`), die maximale Ziffernanzahl (`int`; nur relevant, wenn gefundener Term `None` ist) und Tabuzahlen (`set`).
- `found_extended_terms` speichert alle erweiterten Terme. Die Werte sind wieder Tupel, aber nur mit gefundenem Term und zugehöriger Anzahl.

Zusätzlich gibt es ein Attribut `all_found_terms`. Dieses nutzt `ChainMap`[20], damit beide Caches bequem durchsucht werden können (`found_extended_terms` wird bevorzugt):

```
self.all_found_terms = ChainMap(self.found_extended_terms, self.found_simple_terms)
```

2.2 Klassen für Terme bzw. Operationen

Für jede Operation gibt es eine Klasse: `Summation`, `Subtraction`, `Multiplication`, `Division`, `Factorial`, `Power`. Jede dieser Klasse hat zwei (`term1` und `term2`) oder ein (`term1`) Attribut, je nachdem, ob es sich um eine einstellige oder zweistellige Verknüpfung handelt. Darin werden die Terme der beiden Operanden (wiederum eine der Klassen) oder ein `int` gespeichert. Diese Art der Darstellung ist effizienter als beispielsweise die Terme als String abzuspeichern, weil auf diese Weise nicht für jeden Term ein neuer String gebildet werden muss. Durch die Referenzen auf Objekte werden keine gleichen Teilterme doppelt gespeichert.

Jede der Klassen besitzt ein Klassenattribut `OPERATOR`. Dieses Attribut ist eine Funktion, die die Operation anwendet, die durch die Klasse dargestellt wird. Werte für `OPERATOR` sind beispielsweise `operator.add`[21], `math.factorial`, oder auch `fractions.Fraction`[22] für die Division (letzteres ist

[19] Es gibt ein zweites Attribut, `str_digit`, das `str(self.digit)` speichert. Dieses wird als Parameter vor allem für die kleinen Funktionen verwendet, die häufig die Ziffer als String und nicht als Integer benötigen.
[20] docs.python.org/3/library/collections.html#chainmap-objects
[21] docs.python.org/3/library/operator.html
[22] docs.python.org/3/library/fractions.html#fractions.Fraction

eigentlich eine Klasse und keine Funktion, das spielt in Python allerdings keine Rolle; zu `Fractions` später mehr). Alle Klassen, die zweiseitige Verknüpfungen darstellen, erben von `BinaryOperation`, alle anderen von `UnaryOperation`. `Binary-` und `UnaryOperation` erben von `Operation`, dies ist für die Dokumentation aber nicht weiter wichtig.

2.3 Einfache und erweiterte Termbildung

Für einfache und erweiterte Termbildung ist die Methode `search_terms` zuständig. Diese Methode wird aufgerufen, bevor mit den Methoden `find_simple_term` oder `find_extended_term` nach einfachen oder erweiterten Termen gesucht wird. Als Argumente erhält sie zum einen die Ziffernanzahl `extended_count_limit`, bis zu der nach erweiterten Termen gesucht werden soll. Nachdem Terme bis zu dieser Anzahl gesucht wurden, werden die gefunden Terme weiter kombiniert, allerdings nur mit Grundrechenarten und solange, bis sie die Ziffernanzahl `simple_count_limit` (zweites Argument, dieses muss größer gleich `extended_count_limit` sein) erreicht haben. Das dritte Argument ist `max_found_term`, ein `int`. Dieses Argument ermöglicht eine Begrenzung der gebildeten Terme, die in die Caches geschrieben werden, um so schnellere Zugriffszeiten zu erreichen. Nur, wenn ein Term zu einer Zahl `<= max_found_term` auswertet, wird er in den entsprechenden Cache geschrieben.

Die Arrays a_1, a_2 und a_3 werden durch die Variablen `simple_terms`, `bad_simple_terms` (beide lokal) und `self.extended_terms` (Attribut von `BirthdayNumberFinder`) dargestellt. Die gefundenen erweiterten Terme werden für die Suche nach einem erweiterten Term später benötigt und daher im Gegensatz zu den anderen in einem Attribut gespeichert. Die Listen haben jeweils die Länge `simple_count_limit` – bis zu dieser Anzahl werden Terme gesucht. Elemente der Listen sind wiederum Listen. Diese Listen enthalten alle Terme mit der entsprechenden Ziffernanzahl. Ein Term wird durch ein Tupel dargestellt, das aus der Zahl, zu der der Term auswertet, und aus dem zugehörigen Term besteht.

Eine erste for-Schleife iteriert über die aktuellen Zifferanzahlen `count` von 1 bis `extended_count_limit`, eine zweite danach von `extended_count_limit+1` bis `simple_count_limit`. In jedem Schleifenkörper wird zunächst die Schnapszahl für die aktuelle Anzahl gebildet und an die der aktuellen Anzahl entsprechende Liste aus `simple_terms` angefügt. Jede der zwei Schleifen enthält eine weitere Schleife, die über die Anzahlen `c1` und `c2` iteriert, deren Summe `count` ist. Der Schleifenkörper dieser Schleifen ist der einzige Unterschied zwischen den beiden äußeren Schleifen: In der ersten Schleife werden Terme auch mit Potenzen und Fakultäten kombiniert, in der zweiten nicht.

Um diese Erklärungen nicht im Abschnitt „Quellcode“ zu wiederholen, ist der Quellcode dieser zwei inneren Schleifen bereits im Folgenden dargestellt:

Innere Schleife der ersten Schleife:

```
for c1, c2 in zip(
        range(1, math.floor(count / 2) + 1),
        range(count - 1, math.ceil(count / 2) - 1, -1)
):
    combine_terms_to_simple(itertools.chain(simple_terms[c1], bad_simple_terms[c1]),
                            lambda: itertools.chain(simple_terms[c2], bad_simple_terms[c2]))
    combine_terms_to_extended(self.extended_terms[c1], self.extended_terms[c2])
    combine_terms_to_extended(self.extended_terms[c1], simple_terms[c2])
    combine_terms_to_extended(simple_terms[c1], self.extended_terms[c2])
```

Innere Schleife der zweiten Schleife:

```
for c1, c2 in zip(
        range(1, math.floor(count / 2) + 1),
        range(count - 1, math.ceil(count / 2) - 1, -1)
):
    combine_terms_to_simple_only_basic_operations(
        itertools.chain(simple_terms[c1], bad_simple_terms[c1]),
        lambda: itertools.chain(simple_terms[c2], bad_simple_terms[c2]))
```

```
    combine_terms_to_extended_only_basic_operations(self.extended_terms[c1],
                                                    self.extended_terms[c2])
    combine_terms_to_extended_only_basic_operations(self.extended_terms[c1], simple_terms[c2])
    combine_terms_to_extended_only_basic_operations(simple_terms[c1], self.extended_terms[c2])
```

Besonders die Berechnung der Kombinationen für die beiden Anzahlen in den Schleifenköpfen ist hervorzuheben. Auch gut zu erkennen sind die Kombinationsmöglichkeiten für erweiterte Terme. Der jeweils erste Funktionsaufruf kombiniert einfache Terme zu einfachen Termen, und die folgenden drei kombinieren Terme so, dass erweiterte Terme entstehen, wie in der Lösungsidee auf Seite 16 dargestellt.

In obigem Quellcode werden verschiedene lokale Funktionen[23] verwendet, um Listen mit Termen zu kombinieren. Die Funktionen sind sich sehr ähnlich, um jedoch unnötige if-Bedingungen zu vermeiden und damit die Laufzeit des Programms zu verbessern, existiert für jeden Zweck eine Funktion. Alle Funktionen erhalten zwei Argumente, nämlich die beiden Listen, die Terme enthalten. Eine Ausnahme stellen die Funktionen dar, in denen neue einfache Terme gebildet werden: Es müssen zwei Listen verkettet werden. Damit nicht eine neue Liste erstellt werden muss, wird ein Iterator übergeben, der über die Elemente beider Listen iteriert. In der Funktion werden zwei verschachtelte Schleifen verwendet, um das kartesische Produkt zu bilden. Die äußere Schleife iteriert über die Elemente des ersten Parameters, die innere über die Elemente des zweiten. Da jedoch beim zweiten Durchlaufen der äußeren Schleife der Iterator für die innere Schleife keine Elemente mehr hätte, wird der zweite Parameter als Funktion übergeben, sodass der Iterator bei jedem Durchlauf neu erzeugt werden kann:

```
for t1_num, t1 in terms1:  # terms1 ist erster Parameter
    for t2_num, t2 in terms2():  # terms2 ist zweiter Parameter, eine Funktion
```

Im Folgenden alle Funktionen, die aus zwei Listen mit Termen neue Terme bilden:

Funktion	Kombiniert Terme auch durch Potenzen und Fakultäten?	Bildet nur erweiterte Terme (d.h. sollte mindestens ein Argument nur erweiterte Terme enthalten)?
`combine_simple_terms`	ja	nein
`combine_extended_terms`	ja	ja
`combine_simple_terms_only_basic_operations`	nein	nein
`combine_extended_terms_only_basic_operations`	nein	ja

Diese Funktionen iterieren über die beiden Listen mit Termen und für jede so entstandene Kombination werden weitere lokale Funktionen aufgerufen, die den neu gefundenen Term überprüfen und gegebenenfalls speichern. `combine_simple_terms` und `combine_extended_terms` bzw. `combine_simple_terms_only_basic_operations` und `combine_extended_terms_only_basic_operations` unterscheiden sich nur durch die Namen der Funktionen, die sie aufrufen, um einen neuen Term zu speichern. Den Funktionen, die einen neuen Term speichern, werden immer die beiden Operanden als Zahlen und als Term sowie die Operation (eine der in 2.2 vorgestellten Klassen), die auf die Terme angewendet werden soll, übergeben. Jede der Funktionen berechnet zunächst die Zahl, zu der der neue Term auswertet, mit dem `OPERATOR`-Attribut und erstellt den zugehörigen Term als Objekt der übergebenen Klasse mit den/mit dem übergebenen Operanden. Außerdem wird geprüft, ob die Zahl schon gespeichert wurde (zu diesem Zweck existieren zwei Mengen, `found_simple_numbers` und `found_extended_numbers`). Nur, wenn die Zahl sich noch nicht in der Menge bzw. den Mengen

[23] Funktionen, die sich in der Methode `search_terms` befinden und daher nur innerhalb `search_terms` genutzt werden können. Lokale Funktionen eignen sich in Python besonders zur Strukturierung des Quellcodes.

befindet, wird fortgefahren. Ist die berechnete Zahl außerdem kleiner als `max_found_term` (die übergebene Grenze für die Caches), wird der Term im entsprechenden `dict` gespeichert.

Die Funktionen sind in folgender Aufzählung dargestellt:

- `add_factorial`: Diese Funktion wird von keiner `combine_*`-Funktion direkt aufgerufen, sondern von `add_simple_number` und `add_extended_number`, wenn ein neuer Term gebildet wurde. Dieser Funktion wird die Zahl übergeben, von der die Fakultät gebildet werden soll. Wenn diese Zahl kleiner als 3 oder größer als die Grenze für Fakultäten (70) ist, tut die Funktion nichts.[24] Ansonsten wird ein neuer Term (`Factorial`) gebildet und dieser Term `self.found_extended_terms` (Cache) und der entsprechenden Liste aus `self.extended_terms` hinzugefügt. Die Funktion ruft sich selbst auf, sofern ein neuer Term gefunden wurde. Eine unendliche Rekursion wird durch die Fakultätsgrenze verhindert.
- `add_simple_number`: Dieser Funktion müssen zwei einfache Terme übergeben werden. Sie wird nur von `combine_simple_terms` verwendet. Wenn die Zahl nicht in der Menge `found_simple_numbers` ist, ist der Term auf jeden Fall neu. In welcher Liste (`simple_terms` oder `bad_simple_terms`) er jedoch gespeichert wird, entscheidet, ob er sich auch in `found_extended_numbers` befindet. Wenn ja, handelt es sich um einen schlechten einfachen Term.
- `add_extended_number`: Diese Funktion wird von `combine_simple_terms` (für Potenzen) und von `combine_extended_terms` (für alle neuen Terme) aufgerufen. Sie ist genauso aufgebaut wie `add_simple_number`, mit den Unterschieden, dass `add_extended_number` neue Terme zum Cache für erweiterte Terme `self.found_extended_terms` hinzufügt und dass Terme immer zur entsprechenden Liste aus `self.extended_terms` hinzugefügt werden.
- `add_simple_number_without_factorial`: Wie `add_simple_number`, nur ohne Aufruf von `add_factorial`, also ohne Bildung einer neuen Fakultät. Diese Funktion wird daher von `combine_simple_terms_only_basic_operations` genutzt.
- `add_extended_number_without_factorial`: Wie `add_extended_number`, nur ohne Aufruf von `add_factorial`, also ohne Bildung einer neuen Fakultät. Diese Funktion wird daher von `combine_extended_terms_only_basic_operations` genutzt.

2.3.1 Nicht-Ganzzahlige Terme

Die vorgestellte Termbildung soll auch nicht-ganzzahlige Teilterme bilden können. Um die Bildung nicht-ganzzahliger Terme zu verhindern, kann die Konstante[25] `ENABLE_NON_INTEGERS` (globale Variable im Skript aufgabe2.py) auf `False` gesetzt werden. Ist sie auf `False` gesetzt, wird in den `combine_*`-Funktionen vor dem Aufruf einer `add_*`-Funktion für Division geprüft, ob die beiden Operanden teilbar[26] sind. Ist das nicht der Fall, wird die `add_*`-Funktion nicht aufgerufen. Dadurch läuft das Programm deutlich schneller, es gibt aber eventuell falsche Terme.

Für die Bildung nicht-ganzzahliger Terme ist `Division.OPERATOR` auf `Fraction` gesetzt. Wird also `Division` als Operation an eine `add_*`-Funktion übergeben, erstellt diese mit den Operanden einen neuen Bruch, zu der der neue Term auswertet. Wird mit einem `Fraction`-Objekt mit beliebigen Operationen weitergerechnet, ist das Ergebnis selbst auch ein `Fraction`.[27] Die `add_*`-Funktionen

[24] Die Fakultätsgrenze ist in der lokalen Variablen `MAX_FACTORIAL_N` von `BirthdayNumberFinder.search_terms` gespeichert.
[25] In Python werden Konstanten üblicherweise großgeschrieben, genaugenommen sind es aber keine Konstanten, da sich ihr Wert trotzdem verändern lässt.
[26] „Teilbar“ -> Bei der Division ist der Rest = 0.
[27] Wird der Bruch als Exponent genutzt, ist das Ergebnis stattdessen ein `float`, mit dem nicht weitergerechnet werden sollte. Da Brüche als Exponenten aber ohnehin nicht relevant sind, werden Potenzen mit einem `Fraction` als Exponent erst gar nicht gebildet.

überprüfen daher, ob die neu gebildete Zahl ein Bruch ist (`type(num) == Fraction`) und falls ja, überprüfen sie auch, ob der Nenner (`num.denominator`) `== 1` ist. Ist das nicht der Fall, handelt es sich nicht um eine Ganzzahl und der neue Term wird nicht dem Cache hinzugefügt und `add_factorial` wird für diese Zahl nicht aufgerufen. Ist das aber der Fall, ist trotz der Verwendung von Brüchen nun eine Ganzzahl entstanden, für die `add_factorial` aufgerufen wird und die wie gewöhnlich in den entsprechenden Cache geschrieben wird.

2.4 Erweiterten Term finden

Wurde die Termbildung durchgeführt, kann die Methode `find_extended_term` aufgerufen werden, um einen erweiterten Term zu suchen. Der Methode wird die gesuchte Zahl `number` und der bisher beste gefundene Term (beispielsweise der einfache Term für `number`) als `best_term` mit Anzahl `best_count` übergeben, wodurch ausgeschlossen wird, dass ein Ergebnis schlechter als `best_count` gesucht oder gefunden wird. Im einfachsten Fall befindet sich `number` in `self.found_extended_terms`, dann wird der gecachte Term direkt zurückgegeben. Ansonsten wird versucht, `best_term` und `best_count` zu verbessern. In diesem Fall ist es möglich, dass nicht der optimale Term gefunden wird, weil `self.extended_terms` keine Terme bis zur eigentlich nötigen Ziffernanzahl enthält. Wie in der Lösungsidee beschrieben, werden alle Terme in `self.extended_terms` durchgegangen, beginnend mit der kleinsten Ziffernanzahl `extended_term_count=1`, und `extended_term_count` wird solange erhöht, bis `extended_term_count < best_count-1`. Für einen Term aus `self.extended_terms` wird für jede passende Grundrechenart ein zweiter Operand gesucht. Befindet sich der Operand in `self.all_found_terms` und verbessert die Ziffernanzahl des gespeicherten Terms die bisher beste gefundene Anzahl `best_count` (d.h. *`Ziffernanzahl zweiter Operand`* `<= best_count - extended_term_count - 1`), werden `best_count` und `best_term` entsprechend aktualisiert. Das Ergebnis – `best_term` und `best_count` – wird zurückgegeben.

2.5 Einfachen Term finden – Tiefensuche

Das Tiefensuche-Verfahren, um einen einfachen Term zu finden, ist in zwei Methoden implementiert:

- `BirthdayNumberFinder.analyse_number`, um eine Zahl zu zerlegen, und
- `BirthdayNumberFinder.find_simple_term`, die die eigentliche Tiefensuche implementiert und von `analyse_number` generierte Zerlegungen für eine Zahl durchgeht.

`analyse_number` wird die Zahl übergeben, für die Zerlegungen generiert werden sollen. `analyse_number` ist ein Generator, es werden per `yield` neue Zerlegungen zurückgegeben. Die Zerlegungen sind als ein Objekt einer `Operation`-Klasse (Abschnitt 2.2) dargestellt. Die Operanden (Attribute `term1` und `term2`) sind jedoch Tupel, die den eigentlichen Wert des Operanden sowie die zugehörige Mindestzifferzahl enthalten. Zerlegungen für Addition und Multiplikation werden in einer Liste nach ihrer Mindestzifferanzahl zwischengespeichert (entspricht Liste l im Algorithmus auf Seite 12). Die Generierung von Zerlegungen mit Subtraktion und Division übernehmen zwei weitere Generatoren, wie in der Lösungsidee beschrieben. Um Zahlen zusammen mit ihrer Mindestzifferzahl zu generieren (Optimierung aus Abschnitt 1.3.6), gibt es die Generatoren `range_with_required_digits`, `range_with_required_digits_from` und `range_with_required_digits_unlimited`. Eine genaue Beschreibung dieser Funktionen ist am besten mit dem Quellcode zusammen möglich, siehe dort. Wie schon in der Lösungsidee beschrieben, erzeugt der Generator nun nacheinander Zerlegungen mit aufsteigender Mindestzifferzahl. Dabei macht er deutlich, dass eine neue Anzahl beginnt, indem er zuerst `None` erzeugt (`yield None`). Damit bricht `find_simple_term` die Iteration über die Zerlegungen, die `analyse_number` liefert, zunächst ab, und erhält von `analyse_number` die neue Mindestzifferzahl, indem `analyse_number` `yield count` ausführt.

`find_simple_term` wird die Zahl `number` übergeben, zu der ein Term gesucht wird, sowie die maximale Ziffernanzahl `max_count` und die bestmögliche Anzahl (siehe Eingabe des Algorithmus in Abschnitt 1.3.7). Zurückgeben werden ein Term oder `None`, wenn kein Term gefunden wurde, und die Ziffernanzahl des Terms. Die Methode prüft zunächst, ob sich die Zahl bereits in `self.found_simple_terms` befindet. Die aktuellen Tabuzahlen werden im Attribut `taboo_numbers` (ein `set`) von `BirthdayNumberFinder` gespeichert. Falls `number` im Cache steht, kann so per `old_taboo_numbers.issubset(self.taboo_numbers)` (`old_taboo_numbers` ist die Menge aus dem Cache) überprüft werden, ob die alten Tabuzahlen eine Teilmenge der aktuellen sind.

`number` wird zu `self.taboo_numbers` hinzugefügt, bevor die von `analyse_number` generierten Zerlegungen durchgegangen werden, und bevor die Funktion ihr Ergebnis zurückgibt wieder aus der Menge entfernt. Eine Zerlegung wird nur betrachtet, wenn sich ihre Operanden nicht in `self.taboo_numbers` befinden, wenn also

```
term.term1[0] not in self.taboo_numbers and term.term2[0] not in self.taboo_numbers
```

zu `True` auswertet (`term` ist die aktuell betrachtete Zerlegung). Eine Zerlegung wird betrachtet, indem `find_simple_term` rekursiv zunächst für den kleineren Operanden[28] aufgerufen wird. Ist der resultierende Term nicht `None`, wird auch ein Term für den größeren Operanden gesucht (jeweils mit einer entsprechend berechneten maximalen Ziffernanzahl, die als Parameter `max_count` übergeben wird). Ist auch der Term für den größeren Operanden nicht `None`, verbessert die Zerlegung den aktuellen Term.

`BirthdayNumberFinder.search_terms` setzt das Attribut `simple_search_count_limit` auf den Wert des Parameters `simple_count_limit`, also auf die Ziffernanzahl, bis zu der per Termbildung einfache Terme gesucht wurden. `BirthdayNumberFinder.search_terms` nutzt dieses Attribut für den Fall, dass eine Zahl nicht im Cache gefunden wurde. Ist die maximale Ziffernanzahl `max_count <= self.search_count_limit`, wird direkt `None` als gefundener Term zurückgegeben.

2.5.1 Kleine Funktionen

Eine Reihe kleinerer Funktionen wird benötigt. Für diese wird der Quellcode bereits hier aufgeführt, um Wiederholungen zu vermeiden.

```
@lru_cache(maxsize=None)
def get_repdigit(str_digit: str, length: int):
    return int(str_digit * length)
```

Diese Funktion erstellt eine Schnapszahl mit vorgegebener Länge und Ziffer. Genutzt wird sie beispielsweise vom `BirthdayNumberFinder`-Konstruktor. Die Erstellung eines `int`s aus einem String ist schneller als die Berechnung der Schnapszahl, wie folgender Versuch zeigt (funktioniert auch mit anderen Ziffernanzahlen als 10):

```
>>> timeit.timeit("f(10, '2')", "f=lambda l,d:int(d*l)")
0.5736585570002717
>>> timeit.timeit("f(10, 2)", "f=lambda l,d:d*((10**l-1)//9)")
1.0174948559997574
```

```
@lru_cache(maxsize=None)
def get_digit_count(number: int):
    return math.floor(math.log10(number)) + 1
```

Diese Funktion berechnet, wie viele Ziffern eine Zahl im Dezimalsystem benötigt.

---

[28] Zur Begründung, weshalb der kleinere Operand zuerst betrachtet wird, siehe Fußnote 14.

```python
@lru_cache(maxsize=None)
def get_min_required_digits(number: int, str_digit: str):
    length = get_digit_count(number)
    repdigit = get_repdigit(str_digit, length)
    if number <= repdigit:
        return length
    return length + 1
```

Diese Funktion berechnet die Mindestzifferzahl für eine übergebene Zahl (Abschnitt 1.1.1).

```python
def factors(number):
    for i in range(2, int(math.sqrt(number)) + 1):
        if number % i == 0:
            yield i
```

Dieser Generator erzeugt die Teiler einer Zahl, die kleiner oder gleich der abgerundeten Quadratwurzel der Zahl sind. Er wird in `analyse_number` für die Generierung der Zerlegungen mit Multiplikation genutzt.

2.6 Das Hauptprogramm

aufgabe2.py enthält eine Funktion `main`, die Terme für 2019, 2020, 2030, 2080 und 2980 für jede Ziffer von 1 bis 9 mit und ohne Potenzen und Fakultäten berechnet und im Verzeichnis `Aufgabe2/results` speichert. Beim Ausführen von aufgabe2.py wird standardmäßig diese Funktion aufgerufen. Pro Ziffer spezifiziert ein Objekt des `NamedTuples` `DigitForSearching`, welche Werte für `extended_count_limit` und `simple_count_limit` an `BirthdayNumberFinder.search_terms` für die Termbildung übergeben werden. Diese Werte sind optimal gewählt, sodass das Programm am wenigsten Zeit benötigt und trotzdem die richtigen Ergebnisse liefert. Nach der Termbildung werden `BirthdayNumberFinder.find_simple_term` und `find_extended_term` für jede der Zahlen aufgerufen. Auch für `find_simple_term` sind im `DigitForSearching`-Objekt optimale `max_count`-Werte gespeichert, also die erwartete Ziffernanzahl des Terms. Außerdem sind für `find_extended_term` die erwarteten Ziffernanzahlen gespeichert und das Programm gibt eine Warnung aus, wenn die resultierende Ziffernanzahl nicht mit der erwarteten übereinstimmt.

2.7 Empirische Betrachtungen

2.7.1 Schnapszahlen-Optimalität

Die Schnapszahlen-Optimalität wird in jeder `add_*`-Funktion überprüft. Es wird für jeden neu gebildeten Term die Zahl, zu der der Term auswertet, geprüft. Handelt es sich um eine Schnapszahl und benötigt der gefundene Term weniger Ziffern als die Schnapszahl, wird eine Warnung ausgegeben:

```python
if (c := (str_num := str(num)).count(self.str_digit)) == len(str_num) and c > count:
    print(f"WARNING: Found better term for repdigit {self.str_digit*c}: {operation(term1,
                                                                                   term2)}")
```

Ob die Zahl eine Schnapszahl ist, wird geprüft, indem die Zahl in einen String umgewandelt wird und dann überprüft wird, ob der String nur aus der Ziffer `BirthdayNumberFinder.str_digit` besteht. Ist das der Fall, sollte die Länge der Schnapszahl kleiner sein als die aktuelle Ziffernanzahl (`count`). Ist das der Fall, wurde ein besserer Term für eine Schnapszahl als die Schnapszahl selbst gefunden. Beim Ausführen des Hauptprogramms wird nur ein besserer Term für `999999999` gefunden. Bei Versuchen mit größeren Werten für `extended_count_limit` und `simple_count_limit` wurden jedoch folgende Terme gefunden:

`11111111111`: `((((11-1)^11)-1)/((11-1)-1))`
`111111111111`: `((((11-1)^(1+11))-1)/((11-1)-1))`
Für `999999999` wurden gleich mehrere Terme gefunden.
Am kürzesten ist jedoch `(((9+(9/9))^9)-(9/9))`.

`9999999999`: `(((9+(9/9))^(9+(9/9)))-(9/9))`
`99999999999`: `(((9+(9/9))^(99/9))-(9/9))`
`999999999999999999`: `(((99+(9/9))^9)-(9/9))`
`9999999999999999999999999999`: `((((9/9)+999)^9)-(9/9))`
`99999999999999999999999999999999999999`: `(((99+(9/9))^(9+9))-(9/9))`
72mal `9`: `(((9+(9/9))^((9*9)-9))-(9/9))`
81mal `9`: `(((9+(9/9))^(9*9))-(9/9))`
90mal `9`: `(((9+(9/9))^(99-9))-(9/9))`
99mal `9`: `(((9+(9/9))^99)-(9/9))`

Die Größe dieser Zahlen und die hohe Ziffernanzahl sprechen jedoch dafür, dass die Terme für Zahlen bis 2980 keine Verwendung finden.

2.7.2 Nicht-ganzzahlige Exponenten

Um herauszufinden, ob nicht-ganzzahlige Exponenten verwendet werden können, wurden – wie bereits in der Lösungsidee beschrieben – möglichst viele Zahlen bis 3000 mit jeder Ziffer gesucht. Der Quellcode dazu findet sich in aufgabe2.py unten und ist auskommentiert. Für jede Ziffer wird die Funktion `save_extended_terms_up_to_number` aufgerufen. Diese erstellt einen `BirthdayNumberFinder` und ruft die Methode `search_terms` auf. Die Werte für `extended_count_limit` und `simple_count_limit` unterscheiden sich; beispielsweise für Ziffer 1 liegen beide bei 11, für Ziffer 4 beide nur bei 6. Das führt dazu, dass für jede Ziffer ungefähr gleich lang gerechnet wird und etwa gleich viele Zahlen bis 3000 gefunden werden (die Zahl der Terme pro Ziffernanzahl, die für eine Ziffer gefunden werden, variiert mitunter stark; siehe auch das Diagramm in der Lösungsidee). `save_extended_terms_up_to_number` speichert alle gefundenen Zahlen inklusive des zugehörigen Terms und Ziffernanzahl unter `Aufgabe2/extended_terms/digit`*`Ziffer`*`.txt` ab (Die Dateien sind in der Einsendung enthalten. Nicht alle Terme in den Dateien müssen richtig sein, weil zum Teil ab einer bestimmten Ziffernanzahl nur einfache, keine erweiterte Termbildung durchgeführt wurde). Diese Dateien können anschließend von `Aufgabe2/analyse_non_integer_exponents.py` analysiert werden. Das Skript probiert für jede Zahl, die es in einer Datei mit erweiterten Termen findet, Exponenten aus, und zwar mit allen möglichen Kombinationen für einen Nenner von 2 bis 10 und einen Zähler von 1 bis 10 (Brüche mit gleichem Zähler und Nenner ausgenommen). Für jeden Exponentenbruch wird eine optimale Darstellung mit einem naiven Verfahren gesucht, das alle Möglichkeiten unter den Zahlen in der Datei durchgeht. Danach geht folgender Code alle Zahlen durch (`numbers` ist ein `dict`, das allen Zahlen aus einer der `digit`*`Ziffer`*`.txt`-Dateien ein Tupel zuordnet, bestehend aus Ziffernanzahl und Term):

```
    for number, (count, term) in numbers.items():
        # Versuche Werte für den Exponenten der Form a/b (wobei x^(a/b) = number, also
        # berechnet sich x aus (a-te Wurzel aus (number^b)). x sollte ganzzahlig sein)
        for a in range(1, 11):
            for b in range(2, 11):
                if a == b:
                    continue
                exponent_count, exponent_term = exponent_counts[(a, b)]
                power = number ** b
                x = int(round(power ** (1 / a)))
                if x ** a == power:  # testen, ob der nicht-gerundete Wert für x
                                     # ganzzahlig ist
                    if x in numbers:
                        if numbers[x][0] + exponent_count < count:
                            print("GEFUNDEN: Zahl", number, "wird dargestellt durch", term,
                                  f"(Ziffernanzahl {count}), könnte aber besser durch",
                                  f"{numbers[x][1]}^({exponent_term}) dargestellt werden")
```

Das Ergebnis: Es wird kein „GEFUNDEN“ ausgegeben. Die getesteten Zahlen können also nicht mit den getesteten Exponenten verbessert werden.

2.7.3 Fakultätsgrenze

In diesem Abschnitt wird beschrieben, wie ermittelt wird, welche Fakultäten von erweiterten Termen genutzt werden. Die Fakultäten werden durch Veränderungen an der Termbildung ermittelt. Alle Zeilen, die zu diesem Zweck hinzukommen, enden mit `# stats` und sind standardmäßig auskommentiert. Alle Zeilen, die mit `# non-stats` enden, sollten auskommentiert werden, sobald die Auskommentierung von Zeilen mit `# stats` aufgehoben wird.[29] Die wichtigste Veränderung ist, dass nun jedes Objekt einer Operationen-Klasse (`Summation`, `Subtraction`, `Factorial`, …) ein zusätzliches Attribut `factorials` (ein `set`) erhält. Dieses speichert, welche Fakultäten in dem Term verwendet werden. Im Konstruktor wird das `factorials`-Attribut aus der Vereinigung der `factorials`-Attribute der beiden bzw. des einen Operanden gebildet. Das führt jedoch zu einem Problem, wenn ein Operand eine Zahl ist: Dann gibt es kein `factorials`-Attribut. Daher werden Zahlen, bevor sie als Operand verwendet werden, in die Klasse `Number` „eingewickelt“. `Number` besitzt das `factorials`-Attribut; es ist eine leere Menge. Mit diesen Veränderungen wird für jede Ziffer wie gewöhnlich `BirthdayNumberFinder.search_terms` aufgerufen, entsprechender Quellcode findet sich auskommentiert unten in aufgabe2.py. Danach werden die Ergebnisse mit einem Aufruf von `BirthdayNumberFinder.print_factorial_terms` in eine Datei im Verzeichnis `Aufgabe2/factorials_in_terms_max100` geschrieben. In die Datei werden alle gefundenen Zahlen kleiner als 10000 geschrieben und, da nun der zur Zahl gehörende Term ein `factorials`-Attribut besitzt, werden die verwendeten Fakultäten mit in die Datei geschrieben. Die letzte Zeile listet alle verwendeten Fakultäten auf. Im Verzeichnis `Aufgabe2/factorials_in_terms_max200` findet sich übrigens auch das Ergebnis, aber mit Fakultätsgrenze 200. Das Skript wurde mit `ENABLE_NON_INTEGERS = False` ausgeführt.

3 Beispiele

3.1 Übersichten

Die folgende Tabelle gibt eine Übersicht darüber, welche Mindestwerte als Parameter an `search_terms` zur Durchführung der Termbildung übergeben werden müssen, um korrekte Ergebnisse für erweiterte Terme zu erhalten. Dargestellt ist auch der Term/die Terme, für den/die ein schlechteres Ergebnis gefunden würde, wenn einer der Parameter niedriger wäre. Die Werte in Klammern in der `simple_count_limit`-Spalte geben an, mit welchem Wert die Beispiele tatsächlich berechnet werden. In zwei Fällen werden Ergebnisse nämlich schneller gefunden, wenn zuvor mehr einfache Terme als nötig per Termbildung gesucht werden.

Ziffer	`extended_count_limit`	`simple_count_limit`	Term(e), durch den/die sich die Werte ergeben (einfacher Teilterm grün, erweiterter Teilterm rot) (mit vereinfachter Klammersetzung)
1	4	6 (8)	2980=((1+((1+(1+1))!))!-11)-1-(1+1)^11
2	4	6	2019=(((2*(2+2))!)/(22-2))+(2+(2/2))[30] 2980=((2*((2+2)!))^2)+((2+((2+2)!))^2)
3	2	4	2980=(((3!)!)+(3/3))+(3*(((3!)!)+33)) 2019=((3!)!)+(3+((3!)*((3!)^3)))

[29] Reguläre Ausdrücke zum „Einschalten“ der Veränderungen: Aus `#(.*)# stats` wird `$1# stats` und aus `(.*)# non-stats` wird `#$1# non-stats` (Multiline-Flag darf nicht gesetzt sein). Es ist mit den Veränderungen nur möglich, `search_terms` von `BirthdayNumberFinder` aufzurufen; die `find_*_term`-Methoden funktionieren nicht mehr. Mit den Veränderungen wird außerdem `MAX_FACTORIAL_N` (siehe Fußnote 24) in `search_terms` auf 100, und nicht 70, gesetzt.

[30] Es ist zu beachten, dass der Term `2020=(2*((22*(2+(2*22)))-2))` (es wäre ein `simple_count_limit` von 7 nötig) nicht zählt. Es handelt sich zwar um den Term, der für Ziffer 2 als erweiterter Term gefunden wird; der Term kann aber, da es sich um einen einfachen Term handelt, genauso gut durch Tiefensuche gefunden werden.

4	2	5	2980=((4!)*((4!)+(4+((4!)*4))))+4 2020=4+(((4+4)!)/((4!)-4))
5	2	5	2020=((5!)+(5^5))-(5*((5!)+((5!)+5))) 2019=((5+(5^5))-(5555/5))[31]
6	2	5 (6)	2019=((6!)+((6!)+(6!)))-(((6!)+(6+((6!)/6)))/6) 2030=((6!)-6)+((((6!)+(6^6))/6)/6)
7	1	7	2020=((7+(7+7))*(((7!)/(7*7))-7))+7
8	1	7	2980=(((8!)+(8*888))/(8+8))+(8+8)
9	6	6	2030=(((9+9)/9)^(99/9))-(9+9)

Die folgende Tabelle führt die gefundene Ziffernanzahlen für jede Ziffer und Zahl auf. In der Zeile mit d=„1“ befinden sich Werte für einfache Terme, in den Zeilen der Form „1e“ für erweiterte. Bei den zwei Termen, für die mit nicht-ganzzahligen Teiltermen bessere Ergebnisse gefunden werden, ist die Ziffernanzahl mit nur ganzzahligen Teilterme in Klammern notiert.

Die Spalte „Zeit Termbildung“ gibt an, wie lang der Aufruf von `BirthdayNumberFinder.search_terms` mit den Werten in obiger Tabelle benötigte (alle Zeitangaben in Sekunden). Nach der Termbildung wurden einfache Terme für die Zahlen mit Aufrufen von `BirthdayNumberFinder.find_simple_term` (Tiefensuche) gesucht und erweiterte Terme mit Aufrufen von `.find_extended_term`. Die für die Aufrufe benötigten Zeiten sind in den Spalten „Zeit Tiefensuche/erweiterte Terme durchsuchen“ dargestellt. Die letzte Spalte „Gesamtzeit“ ist die Summe aller Aufrufe der `find_simple_term`- bzw. `find_extended_term`-Methode. Alle Zeiten entstanden mit `ENABLE_NON_INTEGERS = False`. Zur Spalte „Termbildungszeit“ ist zu beachten, dass `search_terms` mit unterschiedlichen Parametern (siehe obige Tabelle) aufgerufen wurde und sich daher auch die benötigte Zeit verändert.

	Ziffernanzahl					**Zeit Termbildung** (`search_terms`)	**Zeit Tiefensuche/ erweiterte Terme durchsuchen** (`find_simple_term` bzw. `find_extended_term`)					**Gesamt-zeit**
d	**2019**	**2020**	**2030**	**2080**	**2980**		**2019**	**2020**	**2030**	**2080**	**2980**	
1	11	10	12	12	13	0.104	0.069	0.009	0.222	0.265	1.997	2.578
1e	11	10	10	10	11		0.011	0.011	0.011	0.011	0.011	0.076
2	10	8	9	9	11	0.079	0.249	0.028	0.025	0.017	2.952	3.288
2e	9	8	8	8	8		0.012	0.011	0.012	0.013	0.011	0.074
3	7	9	9	9	9	0.036	0.012	0.578	0.969	0.961	1.373	3.912
3e	5	6	6	5	7		0.001	0.004	0.004	0.001	0.004	0.029
4	10	8	10	7	9	0.347	4.141	0.126	7.976	0.004	0.703	12.967
4e	7	5	7	5	5		0.066	**0.000**	0.066	**0.000**	**0.000**	0.148
5	10	8	8	8	8	0.061	9.524	0.112	0.054	0.031	0.034	9.77
5e	8	7	7	7	5		0.012	0.009	0.009	0.010	**0.000**	0.064
6	9	10	10	10	11	0.358	0.163	0.688	0.699	0.673	34.802	37.046
6e	8	7	7	8	8		0.070	0.009	0.009	0.066	0.067	0.238
7	10	9	8	9	11	0.317	0.127	0.127	0.010	0.069	1.656	2.194
7e	9	8 (9)	7	9	8 (9)		0.010	0.097	**0.000**	0.102	0.098	0.417
8	11	9	10	8	11	0.606	3.471	0.156	0.168	0.006	4.925	8.742
8e	9	9	8	8	9		0.111	0.111	0.019	0.019	0.106	0.379

[31] 2020 kann auch mit `extended_count_limit=1` mit 7 Ziffern dargestellt werden (`(((5!)-(((5!)/5)-5))*((5*5)-5)`), 2019 aber nicht.

9	10	9	10	9	10	2.014	1.331	0.155	1.193	0.078	1.542	4.322
9e	8	9	8	8	9		0.111	0.113	0.116	0.117	0.113	0.593

Besonders hervorzuheben sind die fettgedruckten Zeiten für erweiterte Terme. Ein erweiterter Term wurde in diesen Fällen bereits durch die Termbildung (`search_terms`) gefunden und wird direkt zurückgegeben.

Zum Vergleich im Folgenden die Zeiten mit `ENABLE_NON_INTEGERS = True`.

d	Zeit Termbildung	Gesamtzeit Tiefensuche/ erweiterte Terme durchsuchen
1	2.481	2.811
1e		8.938
2	0.622	3.437
2e		3.024
3	0.235	4.014
3e		0.669
4	5.246	14.561
4e		10.750
5	0.690	10.477
5e		2.815
6	8.699	39.162
6e		25.034
7	30.449	2.148
7e		76.891
8	30.109	8.072
8e		78.368
9	12.607	4.541
9e		48.776

3.2 Alle Terme

Im Folgenden alle Terme, die das Skript findet:

Ziffer 1

```
simple  : 2019 = (((1+1)*((11111-1)/11))-1) (digit count: 11)
extended: 2019 = (((1+1)*((11111-1)/11))-1) (digit count: 11)

simple  : 2020 = ((1+1)*((11111-1)/11)) (digit count: 10)
extended: 2020 = ((1+1)*((11111-1)/11)) (digit count: 10)

simple  : 2030 = ((11-1)*(1+(((1+1)*1111)/11))) (digit count: 12)
extended: 2030 = (((1+1)^11)-(1+(11+((1+(1+1))!)))) (digit count: 10)

simple  : 2080 = ((11-1)*((11*(((1+1)*(11-1))-1))-1)) (digit count: 12)
extended: 2080 = (((1+1)^11)+((11*(1+(1+1)))-1)) (digit count: 10)

simple  : 2980 = ((11-1)*(1+((1+(1+1))*((111-11)-1)))) (digit count: 13)
extended: 2980 = ((((((1+((1+(1+1))!))!)-11)-1)-((1+1)^11)) (digit count: 11)
```

Ziffer 2

```
simple  : 2019 = ((2*((22*(2+(2*22)))-2))-(2/2)) (digit count: 10)
extended: 2019 = ((((2*(2+2))!)/(22-2))+(2+(2/2))) (digit count: 9)

simple  : 2020 = (2*((22*(2+(2*22)))-2)) (digit count: 8)
extended: 2020 = (2*((22*(2+(2*22)))-2)) (digit count: 8)

simple  : 2030 = (2+(2*(2+(22*(2+(2*22)))))) (digit count: 9)
extended: 2030 = (((2^(22/2))-2)-(2^(2+2))) (digit count: 8)

simple  : 2080 = ((2+2)*((22-2)*(2+(2+22)))) (digit count: 9)
extended: 2080 = ((2+((2+2)!))*(2*(2*(22-2)))) (digit count: 8)
```

```
simple  : 2980 = (2*(2+((2+22)*(22+(2*(22-2)))))) (digit count: 11)
extended: 2980 = (((2*((2+2)!))^2)+((2+((2+2)!))^2)) (digit count: 8)
```

Ziffer 3

```
simple  : 2019 = (3+((3+3)*(3+333))) (digit count: 7)
extended: 2019 = (((3!)!)+(3+((3!)*((3!)^3)))) (digit count: 5)

simple  : 2020 = ((3+(3/3))+((3+3)*(3+333))) (digit count: 9)
extended: 2020 = ((((3!)!)*3)-((((3!)!)+(((3!)!)/(3!)))/(3!))) (digit count: 6)

simple  : 2030 = (33+(((3+3)*333)-(3/3))) (digit count: 9)
extended: 2030 = ((3*(((3!)!)-3))-(((3!)+((3!)!))/(3!))) (digit count: 6)

simple  : 2080 = ((3/3)+(3*((3+(3*(3+3)))*33))) (digit count: 9)
extended: 2080 = ((((3!)!)*3)-(((3!)!)/(3*3))) (digit count: 5)

simple  : 2980 = ((3/3)+(3*(3+(3*(333-3))))) (digit count: 9)
extended: 2980 = ((((3!)!)+(3/3))+(3*(((3!)!)+33))) (digit count: 7)
```

Ziffer 4

```
simple  : 2019 = ((4-(4/4))+((4+4)*((4*(4*(4*4)))-4))) (digit count: 10)
extended: 2019 = ((((4+4)!)/((4!)-4))+(4-(4/4))) (digit count: 7)

simple  : 2020 = (4+((4+4)*((4*(4*(4*4)))-4))) (digit count: 8)
extended: 2020 = (4+(((4+4)!)/((4!)-4))) (digit count: 5)

simple  : 2030 = ((4*((4*(4*(4*(4+4))))-4))-((4+4)/4)) (digit count: 10)
extended: 2030 = (((4!)+(((4+4)!)+(4^4)))/((4!)-4)) (digit count: 7)

simple  : 2080 = ((4+4)*(4+(4*(4*(4*4))))) (digit count: 7)
extended: 2080 = ((4+4)*(4+(4^4))) (digit count: 5)

simple  : 2980 = (4+(4*((((4*4)+(4/4))*44)-4))) (digit count: 9)
extended: 2980 = (4+(4*((4!)+(((4!)/4)!)))) (digit count: 5)
```

Ziffer 5

```
simple  : 2019 = (((5*(5+(5*(55+(5*5)))))-(5/5))-5) (digit count: 10)
extended: 2019 = ((5+(5^5))-(5555/5)) (digit count: 8)

simple  : 2020 = ((5*(5+(5*(55+(5*5)))))-5) (digit count: 8)
extended: 2020 = ((5^5)-((5*((5!)+((5!)+5)))-(5!))) (digit count: 7)

simple  : 2030 = (5+(5*(5+(5*(55+(5*5)))))) (digit count: 8)
extended: 2030 = ((5+((5!)/5))*((5!)-(55-5))) (digit count: 7)

simple  : 2080 = ((5+(55/5))*(5+(5*(5*5)))) (digit count: 8)
extended: 2080 = ((5^5)-(55*(((5!)/5)-5))) (digit count: 7)

simple  : 2980 = (5+(5*((5*((5*(5*5))-5))-5))) (digit count: 8)
extended: 2980 = (((5^5)-(5*5))-(5!)) (digit count: 5)
```

Ziffer 6

```
simple  : 2019 = ((6+(6*(6+666)))/((6+6)/6)) (digit count: 9)
extended: 2019 = (((6!)+(6!))+((6!)-(((6!)+(6+((6!)/6)))/6))) (digit count: 8)

simple  : 2020 = ((6-(6/6))*(6+((6*66)+((6+6)/6)))) (digit count: 10)
extended: 2020 = ((6!)+((6!)+((6!)-(((6!)+((6!)/6))/6)))) (digit count: 7)

simple  : 2030 = ((6-(6/6))*(((66-6)/6)+(6*66))) (digit count: 10)
extended: 2030 = ((6!)+(((((6!)+(6^6))/6)/6)-6)) (digit count: 7)

simple  : 2080 = ((((6*6)+((6+6)/6))-6)*(66-(6/6))) (digit count: 10)
extended: 2080 = (((6!)+(6!))+((6!)-(((6!)-(((6!)+(6!))/6))/6))) (digit count: 8)

simple  : 2980 = (((6+(6/6))*((6*(6+66))-6))-((6+6)/6)) (digit count: 11)
extended: 2980 = (((6!)*6)-((6!)+((6!)-(((6!)-((6!)/6))/6)))) (digit count: 8)
```

Ziffer 7

```
simple  : 2019 = (((77-7)/7)+(7*((7*((7*7)-7))-7))) (digit count: 10)
extended: 2019 = ((7777-(7!))-(((7!)-(7+7))/7)) (digit count: 9)

simple  : 2020 = ((77/7)+(7*((7*((7*7)-7))-7))) (digit count: 9)
extended: 2020 = (((7+(7+7))*(((7!)/(7*7))-7))+7) (digit count: 8)
```

```
simple  : 2030 = ((7*(7+(7*((7*7)-7))))-77) (digit count: 8)
extended: 2030 = (7*(7+(((7!)/(7+7))-77))) (digit count: 7)

simple  : 2080 = ((7+(77*(((7+7)*(7+7))-7)))/7) (digit count: 9)
extended: 2080 = ((7+(77*(((7+7)*(7+7))-7)))/7) (digit count: 9)

simple  : 2980 = ((7-((7+7)/7))*(7+((7/7)+(7*(7+77))))) (digit count: 11)
extended: 2980 = ((7+((((7!)+((7!)+7))/7)/7))*(7+7)) (digit count: 8)
```

Ziffer 8

```
simple  : 2019 = (((8+(8*((8*(8*8))-8)))/((8+8)/8))-(8/8)) (digit count: 11)
extended: 2019 = ((((8!)/(8+8))-(8*(8*8)))+(88/8)) (digit count: 9)

simple  : 2020 = ((8+(8*((8*(8*8))-8)))/((8+8)/8)) (digit count: 9)
extended: 2020 = ((8+(8*((8*(8*8))-8)))/((8+8)/8)) (digit count: 9)

simple  : 2030 = (((8+8)/8)*(((8*(8*(8+8)))-(8/8))-8)) (digit count: 10)
extended: 2030 = (((8!)/(8*8))+((88*(8+8))-8)) (digit count: 8)

simple  : 2080 = ((8/((8+8)/8))*(8+(8*(8*8)))) (digit count: 8)
extended: 2080 = ((8/((8+8)/8))*(8+(8*(8*8)))) (digit count: 8)

simple  : 2980 = ((8/8)+((8+(8*(8*(8*(8*(8*8)))))/88)) (digit count: 11)
extended: 2980 = ((((8!)+(8*888))/(8+8))+(8+8)) (digit count: 9)
```

Ziffer 9

```
simple  : 2019 = (9+((9+(99+((9+9)*999)))/9)) (digit count: 10)
extended: 2019 = (((9!)/((99+99)-9))+99) (digit count: 8)

simple  : 2020 = (((9+9)/9)*(999+(99/9))) (digit count: 9)
extended: 2020 = (((9+9)/9)*(999+(99/9))) (digit count: 9)

simple  : 2030 = ((9+(9+(99/9)))*((9*9)-(99/9))) (digit count: 10)
extended: 2030 = ((((9+9)/9)^(99/9))-(9+9)) (digit count: 8)

simple  : 2080 = ((9+(9+(9-(9/9))))*((9*9)-(9/9))) (digit count: 9)
extended: 2080 = ((9*(9*(9*9)))-((9+((9!)/9))/9)) (digit count: 8)

simple  : 2980 = ((9+(9/9))*((9+(99*(9+(9+9))))/9)) (digit count: 10)
extended: 2980 = (((99+((9!)/(9+9)))/9)+(9*(9*9))) (digit count: 9)
```

4 Quellcode

4.1 Termbildung

Die Methode `BirthdayNumberFinder.search_terms` ist für die Termbildung zuständig.

Zunächst der Quellcode der lokalen Funktion `add_simple_number` als Beispiel für alle `add_*`-Funktionen. Der Quellcode der anderen `add_*`-Funktionen ist ähnlich aufgebaut; auf Unterschiede wird in den Kommentaren eingegangen:

```
def add_simple_number(operand1, operand2, term1, term2, operation: Type[Operation]):
    nonlocal biggest_num  # erlaube Verändern der lokalen Variablen biggest_num
    # Bilde neue Zahl. Das Attribut OPERATOR ist bei Division 'Fraction' (siehe Abschnitt 2.3.1)
    num = operation.OPERATOR(operand1, operand2)
    if num > max_num:
        # Die neue Zahl ist zu groß (Optimierung 2)
        return
    # Überprüfung der Schnapszahlen-Optimalität
    if (c := (str_num := str(num)).count(self.str_digit)) == len(str_num) and c > count:
        print(f"WARNING: Found better term for repdigit {self.str_digit*c}: {operation(term1,
                                                                                term2)}")
    # in add_extended_number(_without_factorial) werden in der folgenden Zeile beide Mengen,
    # found_simple_numbers und found_extended_numbers, auf Nichtenthaltensein überprüft
    if num not in found_simple_numbers:
        # die Zahl wurde noch nicht gefunden. Wäre die Zahl bereits in found_simple_numbers,
        # wurde ein besserer oder gleich guter Term bereits gefunden
        found_simple_numbers.add(num)
        # 'operation' ist eine Klasse: Summation, Subtraction, Power, ...
        term = operation(term1, term2)
```

```
        if num > biggest_num:
            # Aktualisiere biggest_num
            biggest_num = num
        if type(num) == Fraction:
            # Die neue Zahl ist ein Bruch, ...
            if num.denominator == 1:
                # ... und sie ist eine Ganzzahl. Das Fraction-Objekt wird also gar nicht
                # gebraucht.
                num = num.numerator
                is_int = True  # speichert, ob die Zahl eine Ganzzahl ist oder nicht
            else:
                is_int = False
        else:
            is_int = True
        if is_int and num <= max_found_term:
            # Die Zahl ist eine Ganzzahl, dürfte also in den Cache geschrieben werden. Da sie
            # auch kleiner als die Cache-Grenze max_found_term (Argument von search_terms)
            # ist, füge sie dem Cache hinzu:
            self.found_simple_terms[num] = (term, count, None, set())
        # Der folgende Test entfällt bei add_extended_number(_without_factorial), die Zahl und
        # der Term werden einfach new_extended_terms hinzugefügt.
        if num not in found_extended_numbers:
            new_simple_terms.append((num, term))
            # in add_simple_number_without_factorial und add_extended_number_without_factorial
            # fehlen die folgenden beiden Zeilen
            if is_int:
                add_factorial(num, term)
        else:
            # Es gibt bereits einen erweiterten Term, der zu num auswertet, also ist term ein
            # schlechter einfacher Term
            new_bad_simple_terms.append((num, term))
```

Der Vollständigkeit halber hier noch `add_factorial`.

```
def add_factorial(num, term):
    nonlocal biggest_num
    if 2 < num <= MAX_FACTORIAL_N:
        factorial = math.factorial(num)
        # Überprüfung der Schnapszahlen-Optimalität
        if (c := (str_num := str(num)).count(self.str_digit)) == len(str_num) and c > count:
            print(f"WARNING: Found better term for repdigit {self.str_digit*c}:
                                                                    {Factorial(term)}")
        if factorial > max_num:
            return
        if (factorial not in found_simple_numbers     # gleicher Test wie in
         and factorial not in found_extended_numbers):# add_extended_number(_without_factorial)
            found_extended_numbers.add(factorial)
            factorial_term = Factorial(term)
            if factorial > biggest_num:
                biggest_num = factorial
            new_extended_terms.append((factorial, factorial_term))
            if factorial <= max_found_term:
                self.found_extended_terms[factorial] = (factorial_term, count)
            add_factorial(factorial, factorial_term)  # unendliche Rekursion wird durch
                                                      # MAX_FACTORIAL_N unterbunden
```

Auch die `combine_*`-Funktionen sind sich sehr ähnlich. Hier `combine_simple_terms`:

```
def combine_simple_terms(terms1: Iterable, terms2: Callable[[], Iterable]):
    # bei combine_*_only_basic_operations würden add_*_without_factorial-Funktionen aufgerufen
    # werden
    # statt add_simple_number heißt es in combine_extended_terms(_only_basic_operations)
    # add_extended_number
    for t1_num, t1 in terms1:
        for t2_num, t2 in terms2():
            # ADDITION
            add_simple_number(t1_num, t2_num, t1, t2, Summation)
            # MULTIPLIKATION
            add_simple_number(t1_num, t2_num, t1, t2, Multiplication)
            # DIVISION
```

```
            # Wenn ENABLE_NON_INTEGERS True ist, sind alle Divisionen erlaubt, ansonsten nur
            # solche, die eine Ganzzahl ergeben
            if ENABLE_NON_INTEGERS or t1_num % t2_num == 0:
                add_simple_number(t1_num, t2_num, t1, t2, Division)
            if ENABLE_NON_INTEGERS or t2_num % t1_num == 0:
                add_simple_number(t2_num, t1_num, t2, t1, Division)
            # SUBTRAKTION
            if t1_num > t2_num:
                add_simple_number(t1_num, t2_num, t1, t2, Subtraction)
            elif t1_num != t2_num:
                add_simple_number(t2_num, t1_num, t2, t1, Subtraction)
            # POTENZ (wird bei combine_*_only_basic_operations-Funktionen nicht berücksichtigt)
            if t2_num < 100 and type(t1) != Power and type(t2_num) != Fraction:
                add_extended_number(t1_num, t2_num, t1, t2, Power)
            if t1_num < 100 and type(t2) != Power and type(t1_num) != Fraction:
                add_extended_number(t2_num, t1_num, t2, t1, Power)
```

Und schließlich die eigentliche Funktion:

```
def search_terms(self, extended_count_limit, simple_count_limit, max_found_term):

# Einrückung fehlt, um Platz zu sparen

assert extended_count_limit <= simple_count_limit, \
    "extended_count_limit must be smaller than or equal to simple_count_limit"

MAX_FACTORIAL_N = 70  # größte Zahl, die n in n! haben darf

# self.simple_search_count_limit wird später von self.find_simple_term verwendet, um zu prüfen,
# ob eine Zahl, die sich nicht im Cache befindet, mit dem übergebenen max_count-Attribut
# überhaupt gefunden werden kann (siehe dort)
self.simple_search_count_limit = simple_count_limit

# alle Zahlen, die durch gefundene einfache Terme dargestellt werden. __init__ hat
# self.found_simple_terms bereits mit Schnapszahlen befüllt, also stehen in der Menge zu Beginn
# diese Schnapszahlen
found_simple_numbers = set(self.found_simple_terms.keys())
# alle Zahlen, die durch gefundene erweiterte Terme dargestellt werden
found_extended_numbers = set()

# einfache Terme, nach Ziffernanzahl
# die größtmögliche Ziffernanzahl ist für alle Sorten von Termen simple_count_limit
simple_terms = [[] for _ in range(simple_count_limit + 1)]
# schlechte einfache Terme, nach Ziffernanzahl
bad_simple_terms = [[] for _ in range(simple_count_limit + 1)]
# erweiterte Terme, nach Ziffernanzahl
self.extended_terms = [[] for _ in range(simple_count_limit + 1)]

# OPTIMIERUNG 2 (Abschnitt 1.4.4)
# folgende Liste speichert jeweils die größte Zahl, die mit einer Ziffernanzahl gebildet wurde
# Dadurch, dass math.inf standardmäßig für jede Ziffernanzahl eingetragen ist, ergibt sich
biggest_nums = [math.inf for _ in range(self.max_simple_count + 1)]
# das 0. Element von biggest_num wird benutzt, wenn count == self.max_simple_count (s.u.). Dann
# sollte die maximale Zahl max_num == self.max_number sein, weshalb biggest_nums[0] auf 1
# gesetzt wird
biggest_nums[0] = 1

for count in range(1, extended_count_limit + 1):
    # Berechne, zu welcher Zahl ein Term mit count Ziffern maximal auswerten darf, sodass
    # self.max_number noch erreicht werden kann
    max_num = biggest_nums[self.max_simple_count - count] * self.max_number
    # damit die combine_*-Funktionen einfacher auf die Listen mit den neuen einfachen,
    # schlechten einfachen und erweiterten Termen zugreifen können, zeigen folgende Variablen
    # auf diese Listen
    new_simple_terms = simple_terms[count]
    new_bad_simple_terms = bad_simple_terms[count]
    new_extended_terms = self.extended_terms[count]
    # Berechne die Schnapszahl für diese Ziffernanzahl
    repdigit_num = get_repdigit(self.str_digit, count)
    if repdigit_num <= max_num:
        # Die Schnapszahl darf gefunden werden und ist nicht zu groß.
```

```
        # biggest_num speichert die größte Zahl, die mit dieser Ziffer entstehen kann. Zu Beginn
        # ist es die Schnapszahl. biggest_num wird von den add_*-Funktionen immer dann
        # verändert, wenn ein Term gefunden wurde, der zu einer noch größeren Zahl auswertet.
        # biggest_num wird am Ende in biggest_nums[count] geschrieben
        biggest_num = repdigit_num
        # Die Schnapszahl wird zur Liste mit gefundenen Termen hinzugefügt. Zweites Element im
        # Tupel ist normalerweise der Term (Summation, Subtraktion, Factorial, ...).
        new_simple_terms.append((repdigit_num, repdigit_num))
        # Erzeuge die Fakultät der Schnapszahl als neuen Term
        add_factorial(repdigit_num, repdigit_num)
    else:
        biggest_num = 0
    for c1, c2 in zip(
            range(1, math.floor(count / 2) + 1),
            range(count - 1, math.ceil(count / 2) - 1, -1)
    ):
        # zu itertools.chain und lambda: siehe Umsetzung
        # Erzeuge neue einfache Terme
        combine_simple_terms(itertools.chain(simple_terms[c1], bad_simple_terms[c1]),
                             lambda: itertools.chain(simple_terms[c2], bad_simple_terms[c2]))
        # Erzeuge neue erweiterte Terme
        combine_extended_terms(self.extended_terms[c1], self.extended_terms[c2])
        combine_extended_terms(self.extended_terms[c1], simple_terms[c2])
        combine_extended_terms(simple_terms[c1], self.extended_terms[c2])
    biggest_nums[count] = biggest_num  # siehe oben

for count in range(extended_count_limit + 1, simple_count_limit + 1):
    # […]
    # Quellcode ist identisch mit dem in der ersten Schleife, ...
    for c1, c2 in zip(
            range(1, math.floor(count / 2) + 1),
            range(count - 1, math.ceil(count / 2) - 1, -1)
    ):
        # ... nur hier werden andere Funktionen aufgerufen (siehe Umsetzung)
        combine_simple_terms_only_basic_operations(
            itertools.chain(simple_terms[c1], bad_simple_terms[c1]),
            lambda: itertools.chain(simple_terms[c2], bad_simple_terms[c2]))
        combine_extended_terms_only_basic_operations(self.extended_terms[c1],
                                                     self.extended_terms[c2])
        combine_extended_terms_only_basic_operations(self.extended_terms[c1],simple_terms[c2])
        combine_extended_terms_only_basic_operations(simple_terms[c1],self.extended_terms[c2])
    biggest_nums[count] = biggest_num
```

4.2 Zerlegungen generieren

`BirthdayNumberFinder.analyse_number` generiert mögliche Zerlegungen, die von `BirthdayNumberFinder.find_simple_term` genutzt werden.

```
def analyse_number(self, number):

# Einrückung fehlt, um Platz zu sparen

# Folgende Liste speichert alle mit Addition und Multiplikation gefundenen Terme
results = []

# MULTIPLIKATION
for factor1 in factors(number):
    factor2 = number // factor1
    # Berechne Mindestzifferzahl für beide Faktoren
    min_digit_count1 = get_min_required_digits(factor1, self.str_digit)
    min_digit_count2 = get_min_required_digits(factor2, self.str_digit)
    count = min_digit_count1 + min_digit_count2
    term = Multiplication((factor1, min_digit_count1), (factor2, min_digit_count2))
    try:
        results[count].append(term)
    except IndexError:
        # results ist nicht lang genug, füge genug neue leere Listen an
        results.extend([] for _ in range(count - len(results) + 1))
        results[count].append(term)
```

```
# ADDITION
# Mindestzifferzahlen werden durch Generatoren gebildet, ansonsten wie Multiplikation
half_number = number / 2
for (addend1, min_digit_count1), (addend2, min_digit_count2) in zip(
     range_with_required_digits_from(math.floor(half_number), self.digit, self.str_digit),
     range_with_required_digits(math.ceil(half_number), number-1, self.digit, self.str_digit),
):
    count = min_digit_count1 + min_digit_count2
    term = Summation((addend1, min_digit_count1), (addend2, min_digit_count2))
    try:
        results[count].append(term)
    except IndexError:
        results.extend([] for _ in range(count - len(results) + 1))
        results[count].append(term)

def sub():
    # Generator für SUBTRAKTION
    # Zähle Minuend beginnend bei number+1 hoch,
    # Subtrahend beginnend bei 1
    for (minuend, minuend_min_count), (subtrahend, subtrahend_min_count) in zip(
            range_with_required_digits_unlimited(number + 1, self.digit, self.str_digit),
            range_with_required_digits_unlimited(1, self.digit, self.str_digit)
    ):
        yield (minuend_min_count + subtrahend_min_count,
               Subtraction((minuend, minuend_min_count), (subtrahend, subtrahend_min_count)))

def div():
    # Generator für DIVISION
    # Berechne die Mindestzifferzahl für den ersten Dividenden (number*2) sowie die Schnapszahl,
    # bei deren Überschreitung sich die Mindestzifferzahl des Dividenden das nächste Mal ändert
    dividend_min_count = get_min_required_digits(number*2, self.str_digit)
    # next_dividend_repdigit ist größer oder gleich number*2 (gleich, wenn
    # number*2==Schnapszahl aus self.digit)
    next_dividend_repdigit = get_repdigit(self.str_digit, dividend_min_count)
    # Iteriere unendlich lang durch Werte für Divisor, beginnend bei 2
    for divisor, divisor_min_count in range_with_required_digits_unlimited(2, self.digit,
                                                                           self.str_digit):
        dividend = number * divisor
        if dividend > next_dividend_repdigit:
            # Dividend hat die nächste Schnapszahl überschritten, erhöhe Dividenden-
            # Mindestzifferzahl um 1 und setze die Schnapszahl auf die nächsthöhere
            dividend_min_count += 1
            next_dividend_repdigit = next_dividend_repdigit * 10 + self.digit
        yield (divisor_min_count + dividend_min_count,
               Division((dividend, dividend_min_count), (divisor, divisor_min_count)))

sub_gen = sub()
div_gen = div()
# sub_count ist die Anzahl der neuen Zerlegung für Subtraktion, next_sub
sub_count, next_sub = next(sub_gen)
div_count, next_div = next(div_gen)

# Beginne mit Mindestzifferzahl 2
for count in itertools.count(2):
    yield None  # yield None bedeutet, dass das nächste yield die neue Mindestzifferzahl liefert
    yield count
    try:
        # Erzeuge alle Zerlegungen aus results, ...
        yield from results[count]
    except IndexError:
        # ... sofern results lang genug ist
        pass
    if sub_count == count:
        # die Mindestzifferzahl von next_sub ist count
        yield next_sub
        # Erzeuge solange Elemente aus Generator sub_gen, bis sich Mindestzifferzahl verändert.
        # sub_count und next_sub werden dann in einer der folgenden Iterationen erzeugt, sobald
        # count == sub_count ist
        for sub_count, next_sub in sub_gen:
            if sub_count != count:
                break
```

```
            yield next_sub
    if div_count == count:
        # wie für Subtraktion
        yield next_div
        for div_count, next_div in div_gen:
            if div_count != count:
                break
            yield next_div
```

4.3 Mindestzifferzahl-Generatoren

`range_with_required_digits` liefert aufsteigend Zahlen mit ihrer Mindestzifferzahl von `begin` bis `end`, in 1er Schritten.

```
def range_with_required_digits(begin: int, end: int, digit: int, str_digit: str):
    # str(digit) == str_digit (digit ist BirthdayNumberFinder.digit, str_digit ist
    # BNF.str_digit)
    current_count = get_digit_count(begin)  # aktuelle Mindestzifferzahl
    next_repdigit = get_repdigit(str_digit, current_count)  # Schnapszahl, bei der sich
                                                            # current_count das nächste Mal ändert
    if begin > next_repdigit:
        # begin ist für next_repdigit schon zu groß (bspw. mit digit=4 und begin=50:
        # current_count wäre 2, aber next_repdigit wäre 44)
        current_count += 1
        next_repdigit = next_repdigit * 10 + digit  # Berechnung der nächsten Schnapszahl
    do_return = False
    if end <= next_repdigit:
        # nur noch Zahlen bis end erzeugen, dann Funktion beenden
        next_repdigit = end
        do_return = True
    # erstmal alle Zahlen bis zur nächsten Schnapszahl, erst dann kann die Schleife unten beginnen
    for x in range(begin, next_repdigit + 1):
        yield x, current_count
    if do_return:
        # 'end' war <= next_repdigit
        return
    current_repdigit = next_repdigit
    current_count += 1
    while True:
        next_repdigit = current_repdigit * 10 + digit
        if end <= next_repdigit:
            # nur noch Zahlen bis 'end' erzeugen
            next_repdigit = end
            do_return = True
        for x in range(current_repdigit + 1, next_repdigit + 1):
            yield x, current_count
        if do_return:
            return
        current_count += 1
        current_repdigit = next_repdigit
```

`range_with_required_digits_unlimited` beginnt bei `start` und liefert aufsteigend Zahlen mit ihrer Mindestzifferzahl in 1er Schritten, und zwar ohne Ende. Aufgebaut ist die Funktion genauso wie `range_with_required_digits`, nur ohne `end`. Der Quellcode ist entsprechend ähnlich.

`range_with_required_digits_from` liefert Zahlen beginnend bei `begin`, allerdings in -1er Schritten, also von `begin` bis 1. Der Aufbau der Funktion ist wieder sehr ähnlich wie `range_with_required_digits`, bei `current_count == 0` wird abgebrochen.

4.4 Tiefensuche: Einfachen Term finden

`BirthdayNumberFinder.find_simple_term` findet einen einfachen Term:

```
def find_simple_term(self, number, max_count, best_possible_count):
    def new_term(term):
        # Funktion, um Zerlegung (term) zu überprüfen und ggf. best_term/_count zu verbessern
        nonlocal best_term, best_count
        # Sortiere Operanden nach Größe
```

```
        # Zur Erinnerung: Erstes Element des Tupels eines Operanden enthält Wert, das zweite
        # Element die Mindestzifferzahl
        if term.term1[0] < term.term2[0]:
            small_term, small_min_count = term.term1
            big_term, big_min_count = term.term2
            swap = False  # gibt an, in welcher Reihenfolge die Operanden in der Zerlegung
                          # stehen (kleiner/größerer zuerst)
        else:
            small_term, small_min_count = term.term2
            big_term, big_min_count = term.term1
            swap = True
        # Versuche, Term für kleineren Operanden zu finden, mit entsprechendem Wert für
        # max_count
        term1, length1 = self.find_simple_term(small_term, best_count-big_min_count-1,
                                                                        small_min_count)
        if term1 is not None:
            # Es wurde ein Term für den kleineren Operanden gefunden, suche auch einen für den
            # größeren
            term2, length2 = self.find_simple_term(big_term, best_count-length1-1,
                                                                            big_min_count)
            if term2 is not None:
                # Terme für beide Operanden wurden gefunden. Da max_count so groß war, dass nur
                # Terme gefunden werden, die best_term verbessern, ist die neue Zerlegung besser
                best_count = length1 + length2
                # die Zerlegung kann einfach gespeichert werden, indem die Tupel der Operanden
                # von term durch die gefunden Terme ersetzt werden. So muss kein neues Objekt
                # erstellt werden. Die Zerlegung wird sowieso nicht mehr gebraucht.
                if swap:
                    term.term1 = term2
                    term.term2 = term1
                else:
                    term.term1 = term1
                    term.term2 = term2
                best_term = term
                # Gib zurück, dass best_term verbessert wurde
                return True
        return False

    if max_count < best_possible_count:
        # die maximale Ziffernanzahl ist größer als die bestmögliche Anzahl (Parameter
        # best_possible_count ist die Mindestzifferzahl für number), also ist es nicht möglich,
        # einen Term zu finden
        return None, math.inf

    if number in self.found_simple_terms:
        # number wurde im Cache gefunden
        best_term,best_count,old_max_count,old_taboo_numbers = self.found_simple_terms[number]
        if old_taboo_numbers.issubset(self.taboo_numbers):
            # die alten Tabuzahlen sind eine Teilmenge der aktuellen Tabuzahlen -> gecachtes
            # Ergebnis darf verwendet werden
            if best_term is not None:
               if best_count <= max_count:
                   # der gespeicherte Term hat eine Ziffernanzahl, die zu max_count passt
                   return best_term, best_count
               # best_count passt nicht zu max_count, es kann kein besserer Term gefunden werden
               return None, math.inf
            if max_count <= old_max_count:
              # Es kann kein besserer Term gefunden werden, weil diese Zahl bereits mit einem
              # höheren Wert für max_count gesucht wurde und früher kein Ergebnis gefunden wurde
              return None, math.inf
        if best_term is None and old_max_count < max_count:
            # "die bisher beste Anzahl ist im Moment folglich zu niedrig", siehe Algorithmus auf
            # Seite 11, Schritt 4.c.
            best_count = max_count + 1
            old_taboo_numbers = None
        if best_count > max_count + 1:
            # siehe wieder die Darstellung des vollständigen Algorithmus zur Erklärung
            best_term = None
            best_count = max_count + 1
    else:
```

```
        # self.simple_search_count_limit speichert, bis zu welcher Ziffernanzahl einfache Terme
        # per Termbildung gesucht wurden.
        if max_count <= self.simple_search_count_limit:
            # Die Zahl steht nicht im Cache und max_count ist kleiner als die Ziffernanzahl, bis
            # zu der alle Terme im Cache stehen. Es kann folglich kein Term gefunden werden
            return None, math.inf
        old_taboo_numbers = None
        best_count = max_count + 1
        best_term = None

    self.taboo_numbers.add(number)

    term_generator = self.analyse_number(number)  # generiert Zerlegungen
    while True:
        do_break = False
        # wenn self.analyse_number None liefert, bedeutet das, dass eine neue Mindestzifferzahl
        # beginnt. Es können am Anfang auch mehrere Mindestzifferzahlen ohne Zerlegungen
        # vorkommen (Mindestzifferzahl beginnt bei 2, es kann aber sein, dass es mit dieser
        # Mindestzifferzahl keine Zerlegungen gibt), daher reicht es nicht aus, einmal per 'if
        # term is None' zu prüfen, sondern folgende Schleife läuft solange, bis der term nicht
        # None ist und somit eine Zerlegung mit Mindestzifferzahl min_count ist.
        term = next(term_generator)
        while term is None:
            min_count = next(term_generator)  # die aktuelle Mindestzifferzahl
            if min_count >= best_count:
                # Das Ergebnis kann nicht weiter verbessert werden, weil die Mindestzifferzahl
                # zu groß ist.
                # Beide while-Schleifen müssen abgebrochen werden. In Python geht das mit
                # unübersichtlichen break-, continue- und while-else-Anweisungen. So wird dies
                # in aufgabe2.py gelöst, für hier wurde die übersichtlichere Variante mit einer
                # Variablen gewählt.
                do_break = True
                break
            term = next(term_generator)
        if do_break:
            break
        if term.term1[0] not in self.taboo_numbers and term.term2[0] not in self.taboo_numbers:
            if new_term(term):
                # new_term hat best_term (und best_count) verbessert
                if min_count == best_count:
                    # der gefundene Term kann nicht weiter verbessert werden
                    break

    self.taboo_numbers.remove(number)

    taboo_numbers = self.taboo_numbers.copy()  # Tabuzahlen für den Cache
    if old_taboo_numbers:
        # Bilde Schnittmenge für die Tabuzahlen, die im Cache gespeichert werden, falls es alte
        # Tabuzahlen gibt
        new_taboo_numbers = taboo_numbers & old_taboo_numbers
        taboo_numbers = new_taboo_numbers
    self.found_simple_terms[number] = (best_term, best_count, max_count, taboo_numbers)
    return best_term, best_count
```

4.5 Erweiterten Term suchen

`BirthdayNumberFinder.find_extended_term` findet einen erweiterten Term auch, wenn mit `search_terms` nicht bis zur benötigten Ziffernanzahl Terme gebildet wurden:

```
def find_extended_term(self, number, best_term, best_count):
    # best_term und best_count sollten der einfache Term für number sein. Dieses Ergebnis wird
    # nun verbessert.
    extended_term_count = 1  # aktuelle Ziffernanzahl des erweiterten Terms
    if number in self.found_extended_terms:
        return self.found_extended_terms[number]
    while extended_term_count < best_count-1:
        max_count2 = best_count - extended_term_count - 1  # Ziffernanzahl, die der zweite
                                # Operand maximal haben darf, damit best_term verbessert wird.
        try:
            extended_terms = self.extended_terms[extended_term_count]
```

```
        except IndexError:
            # Es gibt in self.extended_terms keine Terme mehr mit Ziffernanzahl
            # extended_term_count. Es kann nicht weiter gesucht werden.
            break
        # Gehe alle erweiterten Terme mit Ziffernanzahl extended_term_count durch (einer der
        # Operanden)
        for num, term in extended_terms:
            if num < number:
                # ADDITION
                # Versuche, term als einen Summanden zu verwenden
                addend2 = number - num  # Wert zweiter Operand (2. Summand)
                if addend2 in self.all_found_terms:
                    term_addend2, addend2_count, *_ = self.all_found_terms[addend2]
                    # der zweite Summand ist im Cache ...
                    if term_addend2 is not None and addend2_count <= max_count2:
                        # ... und hat die erforderliche Ziffernanzahl, um best_term zu
                        # verbessern
                        best_count = extended_term_count + addend2_count
                        best_term = Summation(term, term_addend2)
                # DIVISION
                # dividend / num = number <=> dividend = number * num
                dividend = number * num
                if dividend in self.all_found_terms:
                    term_dividend, dividend_count, *_ = self.all_found_terms[dividend]
                    # […] siehe Addition
            else:
                # SUBTRAKTION
                # num - subtrahend = number <=> subtrahend = num - number
                subtrahend = num - number
                if subtrahend in self.all_found_terms:
                    term_subtrahend, subtrahend_count, *_ = self.all_found_terms[subtrahend]
                    # […] siehe Addition
                # DIVISION
                # num / divisor = number <=> divisor = num / number
                if num % number == 0:
                    divisor = num // number
                    if divisor in self.all_found_terms:
                        term_divisor, divisor_count, *_ = self.all_found_terms[divisor]
                        # […] siehe Addition
            # SUBTRAKTION
            # minuend - num = number <=> minuend = num + number
            minuend = num + number
            if minuend in self.all_found_terms:
                term_minuend, minuend_count, *_ = self.all_found_terms[minuend]
                # […] siehe Addition
            # MULTIPLIKATION
            if number % num == 0:
                factor = number // num
                if factor in self.all_found_terms:
                    term_factor, factor_count, *_ = self.all_found_terms[factor]
                    # […] siehe Addition
        extended_term_count += 1

    res = (best_term, best_count)
    self.found_extended_terms[number] = res
    return res
```