Aufgabe 2: Geburtstag

Richard Wohlbold
Team-ID: 00487

14. Februar 2020

Inhaltsverzeichnis

1 Lösungsidee

1.1 Generierung der Tabelle

Meine Lösungsidee für das Problem besteht darin, alle möglichen Terme, die durch die gegebenen Rechenoperationen aus der gegebenen Ziffer erhalten werden können, in einer Tabelle zu speichern. Dabei wird nach der Anzahl der vorkommenden Ziffern verfahren: Angefangen wird bei $n = 1$ Ziffern. Für $n = 1$ Ziffern lässt sich ohne Berücksichtigung der Fakultätsfunktion nur ein Term finden, nämlich die Ziffer selbst. Für $n = [2, \infty)$ Ziffern werden Terme mit einer geringeren Anzahlen an Ziffern i,j über die gegebenen Grundrechenarten kombiniert, sodass $n = i + j$. Dabei sollen die Terme bei den kommutativen Grundrechenarten ($+$, $*$) nicht vertauscht werden, da dort dasselbe Ergebnis entsteht, bei den nicht-kommutativen Grundrechenarten jedoch schon, da oft verschiedene Ergebnisse auftreten.

Um beispielsweise alle Terme für $n = 5$ zu finden, werden alle Terme mit $i = 1$ und $j = 4$ und mit $i = 2$ und $j = 3$ über das kartesische Produkt kombiniert und für alle Rechenarten und Reihenfolgen gespeichert.

Es gibt vier Grundrechenarten, von denen zwei nicht-kommutativ sind. Dadurch ergeben sich sechs mögliche neue Terme durch jedes Termpaar, davon ausgehend, dass keine Dopplungen auftreten und alle Terme valide sind (z.B. keine Division durch 0):

$$N(1) = 1$$
$$N(n) = \sum_{i=1}^{n//2} 6i(n-i)$$

Es ergeben sich die N, die in Abbildung 1 zu sehen sind.

1.2 Scannen

Um die Laufzeit des Programms zu verbessern, wartet der Algorithmus nicht darauf, bis die Zahl in der Tabelle auftaucht, sondern überprüft, ob man zwei Terme aus der Tabelle durch eine Grundrechenart

n	$N(n)$
1	1
2	6
3	36
4	432
5	3.888
6	46.656
7	513.216
8	6.718.464
9	78.941.952
10	1.038.002.688
11	12.939.761.664
12	174.505.383.936

Abbildung 1: Anzahl der möglichen Terme N nach der Anzahl der vorkommenden Ziffern n

kombinieren kann, um auf das gewünschte Ergebnis zu kommen. Dieses Verfahren nenne ich *Scan*. Dazu wird für jede Zahl j, für die ein Term in der Tabelle steht, ein Partner p berechnet, mit dem das gewünschte Ergebnis e erhalten werden kann. Dafür gibt es verschiedene Möglichkeiten:

$$\begin{aligned}
e &= j + p & \Leftrightarrow \quad & p = e - j \\
e &= j - p & \Leftrightarrow \quad & p = j - e \\
e &= p - j & \Leftrightarrow \quad & p = j + e \\
e &= p * j & \Leftrightarrow \quad & p = \frac{e}{j} \\
e &= p/j & \Leftrightarrow \quad & p = e * j \\
e &= j/p & \Leftrightarrow \quad & p = \frac{j}{e}
\end{aligned}$$

Für jedes j werden nun alle möglichen Partner p und es wird geschaut, ob p Teil der Tabelle ist. Ist dies der Fall, wird es einer Menge hinzugefügt. Der Term mit der geringsten Anzahl an Ziffern der Menge ist ein mögliches Ergebnis.

1.3 Zusammenfügen

Es wird immer abwechselnd eine Tabelle mit einer Ziffer mehr als zuvor generiert und ein *Scan* ausgeführt. Dies wird solange wiederholt, bis ein Scan ein Ergebnis mit m Ziffern zurückgibt und eine Tabelle mit $m - 2$ Ziffern vorliegt und ein Scan ausgeführt wurde.

An diesem Punkt wird die beste Lösung verwendet, da eine Lösung mit $n = m - 2$ Ziffern in der Tabelle stehen würde, der Scan eine Lösung mit $n = m - 1$ Ziffern erkannt hätte und eine Lösung mit $n = m$ Ziffern bereits vorliegt.

Der Algorithmus kann nicht schon bei der Tabelle mit $m - 3$ Ziffern terminieren, da eine Lösung, die sich aus Termen mit einer Ziffer und $m - 2$ Ziffern zusammensetzt, durch einen *Scan* nicht erkannt werden würde, sodass der Algorithmus nicht die beste Lösung liefern würde. Algorithmus 1 zeigt die Kombination der Tabellengeneration und des Scans.

1.4 Fakultät und Potenzieren

Bei sowohl der Fakultätsfunktion als auch dem Potenzieren treten sehr große Zahlen auf, was das Programm extrem verlangsamen kann. Deshalb wird vor beiden Operationen bestimmt, wie viele Stellen das Ergebnis hat. Ist das Ergebnis zu lang, wird es nicht berechnet und auch nicht der Tabelle hinzugefügt. Für die maximale Länge des Ergebnisses wähle ich 100 Stellen, da auch wissenschaftliche Taschenrechner mit Zahlen jenseits dieser Höhe nicht zurechtkommen.

Algorithm 1 Kombinieren von GENERATE und SCAN, um den Term mit der geringsten Anzahl an Ziffern zu finden

```
i ← 1
n ← ∞
r ← nil
while i ≤ n − 2 do
    GENERATE(i)
    r ← SCAN
    if r ≠ nil then
        n ← DIGITS(r)
    end if
    i ← i + 1
end while
return r
```

1.4.1 Potenzieren

Über den Zehnerlogarithmus und die Logarithmusgesetze kann die Länge des Ergebnisses schnell berechnet werden:

$$
\begin{aligned}
& N = a^b \\
\Rightarrow \quad & \text{len}(N) = \log_{10} a^b \\
\Leftrightarrow \quad & \text{len}(N) = b * \log_{10} a
\end{aligned}
$$

Da das Potenzieren eine nicht-kommutative binäre Operation ist, ähnelt sie den Grundrechenarten und kann in der Generierung der Tabelle hinzugefügt werden, sodass für jede Kombination zweier Terme der Tabelle 8 neue Terme hinzugefügt werden.

Dem *Scan* wird auch zwei Prüfungen hinzugefügt, ob sich das Ergebnis e durch einen Term der Tabelle j und seinen Partner p darstellen lassen:

$$
\begin{aligned}
e = p^j \quad & \Leftrightarrow \quad p = e^{\frac{1}{j}} \\
e = j^p \quad & \Leftrightarrow \quad p = \log_j e
\end{aligned}
$$

1.4.2 Fakultätsfunktion

Aufgrund des schnellen Anstiegs der Fakultätsfunktion, kann die maximale Zahl, für die die Fakultätsfunktion die gegebene maximale Anzahl an Stellen l_{max} nicht überschreitet, einfach berechnet werden.

Da die Fakultätsfunktion eine unäre Funktion ist, verlangt sie nach keinen weiteren Ziffern. Ein Term und seine Fakultät haben deshalb dieselbe Anzahl an Stellen. Beim Erstellen der Tabelle wird dann jeder Term, der der Tabelle hinzugefügt wird und einen Wert ≥ 3 und $\leq l_{\text{max}}$ aufweist, auch als Fakultät der Tabelle hinzugefügt, sofern diese noch nicht enthalten ist. Dies wird rekursiv gelöst, damit beispielsweise auch $3!! = 720$ der Tabelle hinzugefügt wird.

Algorithm 2 Prozedur, um der Tabelle einen Term hinzuzufügen, unter Berücksichtigung der Fakultätsfunktion

```
procedure ADDTOTABLE(T)
    if 3 ≤ VALUE(t) ≤ l_max then
        f ← FACTORIAL(T)
        ADDTOTABLE(f)
    end if
    d ← DIGITS(T)
    if T ∉ t ∧ DIGITS(T) < DIGITS then
        ADD(t, T)
    end if
end procedure
```

2 Umsetzung

Ich habe die in Abschnitt 1 geschilderte Lösungsidee in Python 3 umgesetzt.

2.1 Repräsentation eines Terms

Ich arbeite mit den objektorientierten Features von Python, um Terme als Datenstrukturen zu repräsentieren.

Dazu definiere ich eine Superklasse `Term`, die alle Methoden enthält, die Subklassen auch bereitstellen müssen. Diese Methoden lösen bei `Term` einen `NotImplementedError` aus, sodass sofort auffällt, wenn eine Subklasse eine der Methoden nicht implementiert. Die Methoden von `Term` sind die folgenden:

- `number_of_digits`: Diese Methode gibt die Anzahl der im Term vorkommenden Ziffern zurück.
- `value`: Diese Methode gibt den numerischen Wert des gesamten Terms zurück.
- `__str__`: Diese Methode gibt den Term als Textrepräsentation zurück.
- `__repr__`: Diese Methode gibt im Fall von `Term` den Term auch den Term als Textrepräsentation zurück. Sie ist so defininiert, dass sie dasselbe Ergebnis wie `__str__` zurückgibt.

Nun gibt es verschiedene Varianten eines Terms.

Die einfachste Variante ist eine Zahl, die eine gewisse Ziffer ein- oder mehrmals enthält (un zwar nur diese). Die Klasse heißt `Number` und erhält ihren Wert als Konstruktorargument. Ihr numerischer Wert gleicht logischerweise ihrem inneren Wert. Die Anzahl an Stellen lässt sich durch die Länge der Textrepräsentation berechnen. Ihre Textrepräsentation ist die Zahl als `str`.

Die zweite Möglichkeit eines Terms ist eine unäre Operation, d.h. eine Operation, die auf einem Term operiert. Im Fall der Aufgabenstellung ist dies nur die Fakultätsfunktion. Die Klasse heißt `UnaryOperation` und erhält in ihrem Konstruktor den Term und die entsprechende Operation (nur `O_FAC` erlaubt). Die Berechnung erfolgt, indem die Fakultätsfunktion für den Wert des inneren Terms berechnet wird. Als Text wird die unäre Operation repräsentiert, indem der Term, gefolgt von einem *!* in Klammern gesetzt wird.

Die dritte Möglichkeit eines Terms ist die binäre Operation, d.h. eine Operation die auf zwei Termen operiert. Im Fall der Aufgabenstellung sind dies die vier Grundrechenarten sowie die Potenzfunktion. Die Klasse heißt `BinaryOperation` und erhält in ihrem Konstruktor die beiden Terme und die entsprechende Operation (nur `OP_ADD`, `OP_SUB`, `OP_MULT`, `OP_DIV`, `OP_POW` erlaubt). Die Berechnung erfolgt, in dem die Operation auf die Werte der beiden inneren Terme ausgeführt wird. Als Text wird die binäre Operation repräsentiert, indem die Terme, getrennt durch das Symbol der entsprechenden Operation, in Klammern gesetzt werden.

Durch das Einklammern aller zusammengesetzten Terme sorgt das Programm dafür, dass am Ende keine Probleme mit der Präzedenz der Operatoren auftreten.

2.2 Datenstruktur der Tabelle

Die Tabelle, die einer Zahl den Term mit der geringsten Anzahl an Ziffern zuordnet, stelle ich als `dict` dar. Dabei ist der Schlüssel des `dicts` die Zahl selbst und der Wert ein Tupel aus einem `Term`-Objekt und der Anzahl an Ziffern im Term, die zwischengespeichert wird, um die Geschwindigkeit des Programms zu erhöhen. Beispielhaft sieht die Tabelle für $n = 1$ und $N = 2$ mit Erlauben der Fakultätsfunktion folgendermaßen aus (die Repräsentation als Zeichenkette der Terme ist abgedruckt):

```
{
  1: (1, 1),
  39916800: ((11!), 2),
  11: (11, 2),
  2: ((1+1), 2),
  0: ((1-1), 2)
}
```

2.3 Generierung der Tabelle

Ich definiere eine Funktion `add_to_table`, die einer Tabelle einen Term hinzufügt. Dabei überprüft die Funktion, ob ein Term mit gleich vielen oder weniger Ziffern bereits in der Tabelle existiert und fügt ihn ihr sonst hinzu. Zustätzlich kann ein Parameter übergeben werden (`extended`), der festlegt, ob die Funktion auch die Fakultät des Terms hinzufügen soll. Um Berechnungen zu vermeiden, bei denen sehr lange Zahlen entstehen, wird die Fakultät eines Terms nur aufgenommen, falls sie 100 oder weniger Stellen besitzt. Der höchste erlaubte Wert eines Terms, dessen Fakultät in die Tabelle aufgenommen wird, wird am Anfang des Programms berechnet und in `MAX_FACTORIAL` gespeichert. In diesem Prozess wird zusätzlich ein `dict` erstellt (`FACTORIALS`), das einem Ergebnis der Fakultätsfunktion seinem Parameter zuordnet. Dieses `dict` wird später in Abschnitt 2.4 verwendet.

Die Funktion `generate` generiert alle Terme der Tabelle, die `num_digits` Ziffern enthalten. Dabei differenziert die Funktion zwischen zwei Tabellen: Der `aggegated_table` und der `split_table`.

Die `aggregated_table` enthält alle, bis zu diesem Zeitpunkt durch die Ziffer dargestellten Terme. Sie folgt dem oben geschilderten Aufbau. Sie wird in der Funktion `scan` eingesetzt, um den Partner p zu ermitteln.

Zusätzlich gibt es die `split_table`, die verschiedene Tabellen enthält. Dabei ordnet sie einer Anzahl an Ziffern N eine Tabelle zu, die nur Terme mit dieser Anzahl an Ziffern enthält. Beispielhaft sieht die `split_table` für $n = 1$ und $N = 2$ mit Erlauben der Fakultätsfunktion folgendermaßen aus (die Repräsentation als Zeichenkette der Terme ist abgedruckt):

```
{
  1: {
    1: (1, 1)
  },
  2: {
    11: (11, 2),
    2: ((1+1), 2),
    0: ((1-1), 2),
    1: ((1/1), 2)
  }
```

Die Funktion geht nun nach dem in Abschnitt 1 geschilderten Verfahren vor. Sie variiert insgesamt die Anzahl an Ziffern des 1. Terms, aus der sich die Anzahl an Ziffern des 2. Terms ergibt. Im Folgenden iteriert sie durch alle Terme mit den gegebenen Anzahlen an Ziffern (aus `split_table`), sodass alle Kombinationen gefunden werden. Die Terme werden nun nach ihrer Größe geordnet (für die Division) und mit allen möglichen Rechenarten (falls `extended` auch mit der Potenzfunktion) verbunden und jeweils mit `add_to_table` der Tabelle hinzugefügt.

Am Ende der Funktion werden alle neuen Terme der neu generierten `split_table` der `aggregated_table` hinzugefügt.

2.4 Scan

Die Funktion `scan` führt das in Abschnitt 1.2 geschilderte Verfahren aus. Falls `extended True` ist, wird auch die Potenzfunktion und die Fakultätsfunktion umgekehrt verwendet. Um die Fakultätsfunktion umzukehren, wird das eingangs generierte `dict FACTORIALS` verwendet, das als Wert die Zahl ausgibt, deren Fakultät den Schlüssel ergibt. Alle gefundenen Möglichkeiten werden einem `set` hinzugefügt. Am Ende der Funktion wird die Möglichkeit mit der geringsten Anzahl an Ziffern ausgewählt. Falls zwei Möglichkeiten dieselbe Anzahl an Ziffern haben, wird die mit weniger Zeichen gewählt. Falls eine Möglichkeit gefunden wurde, wird diese zurückgegeben. Ansonsten wird `None` zurückgegeben.

2.5 Zusammenfügen

Die Funktion `find_shortest` fügt nun `generate_table` und `scan` zusammen. Nach dem in Anbschnitt 1 geschilderten Verfahren wird abwechselnd die Tabelle generiert und ein `Scan` durchgeführt, eine Tabelle mit $m - 2$ Ziffern vorliegt, wobei m die Anzahl an Ziffern des gefundenen Terms ist. Diese Funktion wird insgesamt zweimal ausgeführt, einmal ohne und einmal mit `extended`. Zusätzlich kann festgelegt werden, ob zusätzliche Informationen ausgegeben werden sollen (`debug`).

Das Programm benutzt einen `ArgumentParser`, um die Zahl und die Ziffer aus den Argumenten zu ermitteln, sowie zu überprüfen, ob eine zusätzliche Ausgabe erfolgen soll.

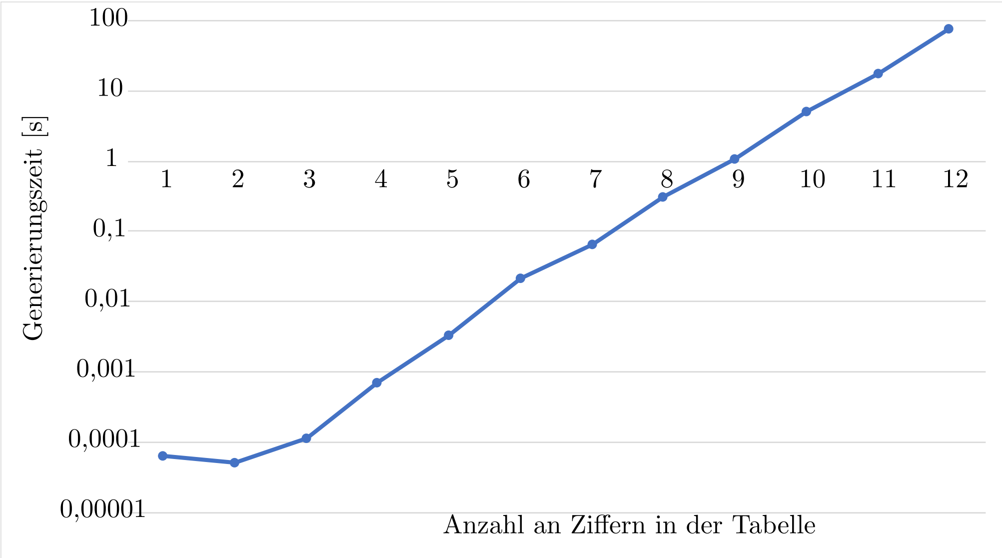

Abbildung 2: Generierungszeit der Tabellen für die Ziffer 4 nach Anzahl der Ziffern pro Term

Sobald endgültige Terme im eingeschränkten bzw. erweiterten Modus gefunden wurden, werden diese mit der Anzahl an Stellen ausgegeben.

2.6 Laufzeitanalyse

Das Element des Programms, welches am längsten benötigt, ist die Generierung der Tabelle. Die Zeit, die für diese genötigt wird, verhält sich exponentiell zu der Anzahl der Ziffern, für die die Funktion `generate` aufgerufen wird. Dies lässt sich auf einem Graphen mit logarithmischer y-Skala, der die Generierungszeit der Tabellen gegen die Anzahl der Ziffern, mit der die Funktion aufgerufen wird, abbildet. Ein solcher Graph ist beispielhaft für die Ziffer 4 in Abbildung 2 zu erkennen.

Um die Geschwindigkeit des Ansteigens zu berechnen, führe ich eine lineare Regression des natürlichen Logarithmus der Generierungszeit gegen die Anzahl der Ziffern pro Term für jede Ziffer durch. Dabei entspricht die erhaltene Steigung m der Wachstumskonstante k beim exponentiellen Wachstum. Die Wachstumskonstante rechne ich nun in den Wachstumsfaktor a um. Die Regression führe ich für sämtliche Ziffern je zweimal durch: Einmal ohne und einmal mit Fakultäts- und Potenzfunktion.

$$\begin{aligned} & \ln t(n) = k * n + b \\ & t(n) = e^b * e^{k*n} = t_0 * e^{k*n} \\ \Leftrightarrow \quad & t(n) = t_0 * (e^k)^n = t_0 * a^n \\ \Rightarrow \quad & a = e^k \end{aligned}$$

Die Wachstumskonstante a stelle ich in einem Säulendiagramm für jede Ziffer dar (siehe Abbildung 3).

Aus beiden Abbildungen zusammen kann erkannt werden, dass die Laufzeit der Tabellengenerierung exponentiell zunimmt, dies aber unterschiedlich schnell geschieht. Es können somit nur eine beschränkte Anzahl an Aussagen zur Laufzeit getroffen werden, da sich die Anzahl der benötigten Tabellen je nach Zahl und Ziffer stark unterscheiden können. Die Generierung der Tabellen mit Fakultäts- und Potenzfunktion braucht aber immer deutlich länger, vor allem für Ziffern ≥ 3. So braucht die Tabelle mit Termen mit 7 Ziffern für die Ziffer 3 ca. $\frac{13^7}{3.8^7} \approx 5484$ Mal länger, wenn Fakultäts- und Potenzfunktionen miteinbezogen werden. Beispielhafte Laufzeiten lassen sich im Abschnitt 3 erkennen.

3 Beispiele

Für jede Ziffer führe ich das Programm für die Zahlen 2019, 2030, 2080 und 2980 aus.

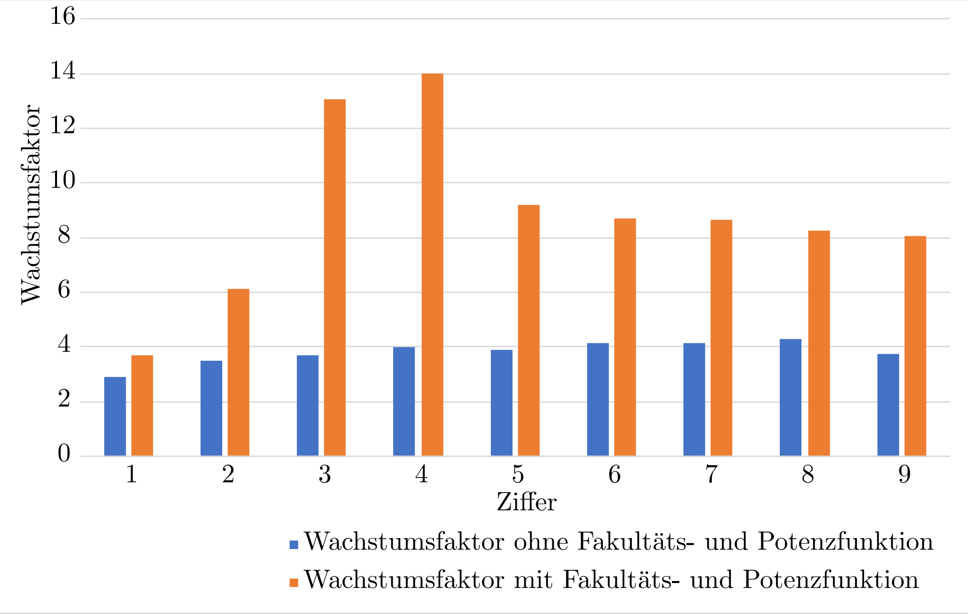

Abbildung 3: Wachstumsfaktoren der Generierungszeiten nach Ziffern, ohne und mit Fakultäts- und Potenzfunktion

3.1 Ziffer 1

3.1.1 2019

Der von meinem Programm gefundene Term hat genauso viele Ziffern wie das Beispiel aus der Aufgabenstellung:

```
$ command time ’time: %E’ python main.py 2019 1
normal result ((((11111-1)/11)*(1+1))-1)
digits: 11

extended result (((1+1)^11)-(((11-1)*((1+1)+1))-1))
digits: 11
0:05.42
```

3.1.2 2030

```
$ command time ’time: %E’ python main.py 2030 1
normal result ((1-11)*(((1+1)*((11-1)-111))-1))
digits: 12

extended result (((1+1)^11)-((((1+1)+1)!)*((1+1)+1)))
digits: 10
0:01.45
```

3.1.3 2080

```
$ command time ’time: %E’ python main.py 2080 1
normal result ((11-1)*(((111-1)*(1+1))-(11+1)))
digits: 12

extended result (((1+1)^11)+((1+1)^((((1+1)+1)!)-1)))
digits: 10
0:01.43
```

3.1.4 2980

```
$ command time 'time: %E' python main.py 2980 1
normal result ((1-11)*((1+1)+(((1+1)+1)*(11-111))))
digits: 13

extended result (((((1+1)+1)!)+1)!)-(((1+1)^11)+(11+1)))
digits: 11
0:05.81
```

3.2 Ziffer 2

3.2.1 2019

Der von meinem Programm gefundene Term hat genauso viele Ziffern wie das Beispiel aus der Aufgabenstellung:

```
$ command time 'time: %E' python main.py 2019 2
normal result (((((22*2)+2)*22)-2)*2)-(2/2))
digits: 10

extended result ((((22*2)+(2/2))^2)-((2+(2/2))!))
digits: 9
0:02.42
```

3.2.2 2030

```
$ command time 'time: %E' python main.py 2030 2
normal result (2+(((((22*2)+2)*22)+2)*2))
digits: 9

extended result (((2^(22/2))-22)+(2^2))
digits: 8
0:00.54
```

3.2.3 2080

```
$ command time 'time: %E' python main.py 2080 2
normal result (((22+(2*2))*(2*2))*(22-2))
digits: 9

extended result (((22-2)*2)*((((2^2)!)+2)*2))
digits: 8
0:00.38
```

3.2.4 2980

```
$ command time 'time: %E' python main.py 2980 2
normal result ((2-((22+2)*((2*(2-22))-22)))*2)
digits: 11

extended result (((2^(2^(2^2)))+((2^2)!))/22)
digits: 8
0:02.97
```

3.3 Ziffer 3

3.3.1 2019

Der von meinem Programm gefundene Term hat genauso viele Ziffern wie das Beispiel aus der Aufgabenstellung:

```
$ command time 'time: %E' python main.py 2019 3
normal result (3+((333+3)*(3+3)))
digits: 7

extended result ((((3!)^3)*(3!))+(((3!)!)+3))
digits: 5
0:00.07
```

3.3.2 2030

```
$ command time 'time: %E' python main.py 2030 3
normal result (33+((333*(3+3))-(3/3)))
digits: 9

extended result (((((3!)!)-3)*3)-((((3!)!)+(3!))/(3!)))
digits: 6
0:00.22
```

3.3.3 2080

```
$ command time 'time: %E' python main.py 2080 3
normal result ((3/3)+((33*3)*(((3+3)*3)+3)))
digits: 9

extended result ((((3!)!)*3)-((((3!)!)/3)/3))
digits: 5
0:00.15
```

3.3.4 2980

```
$ command time 'time: %E' python main.py 2980 3
normal result ((((333*3)-(3+3))*3)+(3/3))
digits: 9

extended result ((((3!)!)+(3/3))+((((3!)!)+33)*3))
digits: 7
0:01.49
```

3.4 Ziffer 4

3.4.1 2019

Der von meinem Programm gefundene Term hat genauso viele Ziffern wie das Beispiel aus der Aufgabenstellung:

```
$ command time 'time: %E' python main.py 2019 4
normal result ((4-(4/4))+((((4*4)*(4*4))-4)*(4+4)))
digits: 10

extended result ((((4+4)!)/((4!)-4))+(4-(4/4)))
digits: 7
0:02.06
```

3.4.2 2030

```
$ command time 'time: %E' python main.py 2030 4
normal result (((444+((4*4)*4))*4)-((4+4)/4))
digits: 10

extended result (((((4+4)!)+(4!))+(4^4))/((4!)-4))
digits: 7
```

```
0:02.09
```

3.4.3 2080

```
$ command time 'time: %E' python main.py 2080 4
normal result ((((4*4)*(4*4))+4)*(4+4))
digits: 7

extended result (((4^4)+4)*(4+4))
digits: 5
0:00.05
```

3.4.4 2980

```
$ command time 'time: %E' python main.py 2980 4
normal result (((((4*4)*4)+4)*44)-((4*4)-4))
digits: 9

extended result ((((((4!)/4)!)+(4!))*4)+4)
digits: 5
0:00.26
```

3.5 Ziffer 5

3.5.1 2019

Der von meinem Programm gefundene Term hat genauso viele Ziffern wie das Beispiel aus der Aufgabenstellung:

```
$ command time 'time: %E' python main.py 2019 5
normal result ((((((55+(5*5))*5)+5)*5)-5)-(5/5))
digits: 10

extended result (((((5!)+5)*((5+5)+5))+(5!))+((5!)/5))
digits: 8
0:04.84
```

3.5.2 2030

```
$ command time 'time: %E' python main.py 2030 5
normal result (5+((((55+(5*5))*5)+5)*5))
digits: 8

extended result (((5!)+(5*5))*(((5!)/5)-(5+5)))
digits: 7
0:00.45
```

3.5.3 2080

```
$ command time 'time: %E' python main.py 2080 5
normal result ((((5*5)*5)+5)*((55/5)+5))
digits: 8

extended result ((5^5)-(55*(((5!)/5)-5)))
digits: 7
0:00.37
```

3.5.4 2980

```
$ command time 'time: %E' python main.py 2980 5
normal result ((55*(55-(5/5)))+(5+5))
digits: 8

extended result (((5^5)-(5!))-(5*5))
digits: 5
0:00.07
```

3.6 Ziffer 6

3.6.1 2019

Der von meinem Programm gefundene Term hat genauso viele Ziffern wie das Beispiel aus der Aufgabenstellung:

```
$ command time 'time: %E' python main.py 2019 6
normal result (((((666+6)*6)+6)*6)/(6+6))
digits: 9

extended result (((6!)+(6!))-(((((6!)+6)+((6!)/6))/6)-(6!)))
digits: 8
0:03.06
```

3.6.2 2030

```
$ command time 'time: %E' python main.py 2030 6
normal result (((6/6)-(6*6))*(((6+6)/6)+(6-66)))
digits: 10

extended result ((((6^6)+(6!))/(6*6))-(6-(6!)))
digits: 7
0:00.78
```

3.6.3 2080

```
$ command time 'time: %E' python main.py 2080 6
normal result (((6-(6*6))-((6+6)/6))*((6/6)-66))
digits: 10

extended result ((((6!)+(6!))+(6!))-(((6!)+(6!))/((6+6)+6)))
digits: 8
0:03.39
```

3.6.4 2980

```
$ command time 'time: %E' python main.py 2980 6
normal result (((((6*6)*(6+6))-6)*(6+(6/6)))-((6+6)/6))
digits: 11

extended result (((((6!)-((6!)/6))/6)-((6!)+(6!)))+((6!)*6))
digits: 8
0:05.09
```

3.7 Ziffer 7

3.7.1 2019

Der von meinem Programm gefundene Term hat genauso viele Ziffern wie das Beispiel aus der Aufgabenstellung:

```
$ command time 'time: %E' python main.py 2019 7
normal result (((77-7)/7)-(7*(7+(7*(7-(7*7))))))
digits: 10

extended result (((((7!)+(7!))*((7!)+7))+(7!))/(((7!)*7)-((7!)+(7!))))
digits: 9
0:36.72
```

3.7.2 2030

```
$ command time 'time: %E' python main.py 2030 7
normal result (((7-(7*(7-(7*7))))*7)-77)
digits: 8

extended result (7*(((7!)/(7+7))+(7-77)))
digits: 7
0:04.35
```

3.7.3 2080

```
$ command time 'time: %E' python main.py 2080 7
normal result ((7-(77*(7-((7+7)*(7+7)))))/7)
digits: 9

extended result ((((((7*7)+7)*7)+7)*7)-(((7!)/7)-7))
digits: 9
0:31.23
```

3.7.4 2980

```
$ command time 'time: %E' python main.py 2980 7
normal result ((((77+7)*7)+(7+(7/7)))*(7-((7+7)/7)))
digits: 11

extended result (((7!)-((7+7)/7))-((7*7)*((7*7)-7)))
digits: 9
0:33.10
```

3.8 Ziffer 8

3.8.1 2019

Der von meinem Programm gefundene Term hat genauso viele Ziffern wie das Beispiel aus der Aufgabenstellung:

```
$ command time 'time: %E' python main.py 2019 8
normal result ((88+(88/8))-(((8+8)+8)*(8-88)))
digits: 11

extended result (((((((8!)+(8!))+(8!))/8)+8)/8)+((8+8)*8))
digits: 9
0:31.39
```

3.8.2 2030

```
$ command time 'time: %E' python main.py 2030 8
normal result (((8+8)/8)*(((8*8)*(8+8))-(8+(8/8))))
digits: 10

extended result ((((8!)/8)/8)-(8-(88*(8+8))))
digits: 8
```

```
0:03.18
```

3.8.3 2080

```
$ command time 'time: %E' python main.py 2080 8
normal result (((((8*8)*8)+8)*(8*8))/(8+8))
digits: 8

extended result (((888*8)+8)+(8-((8!)/8)))
digits: 8
0:02.60
```

3.8.4 2980

```
$ command time 'time: %E' python main.py 2980 8
normal result (((((8*8)*8)*((8*8)*8))+(88+8))/88)
digits: 11

extended result ((8/8)+((((8^8)/(8*8))+8)/88))
digits: 9
0:30.48
```

3.9 Ziffer 9

3.9.1 2019

Der von meinem Programm gefundene Term hat genauso viele Ziffern wie das Beispiel aus der Aufgabenstellung:

```
$ command time 'time: %E' python main.py 2019 9
normal result (9+(((999*(9+9))+(99+9))/9))
digits: 10

extended result (99-((9!)/((9-99)-99)))
digits: 8
0:04.34
```

3.9.2 2030

```
$ command time 'time: %E' python main.py 2030 9
normal result (((9+9)+(99/9))*((9*9)-(99/9)))
digits: 10

extended result ((((9+9)/9)^(99/9))-(9+9))
digits: 8
0:02.57
```

3.9.3 2080

```
$ command time 'time: %E' python main.py 2080 9
normal result ((((9+9)+9)-(9/9))*((9*9)-(9/9)))
digits: 9

extended result (((9*9)*(9*9))-((((9!)/9)+9)/9))
digits: 8
0:02.11
```

3.9.4 2980

```
$ command time 'time: %E' python main.py 2980 9
normal result ((((((9+9)*(9+9))+9)*9)+(9/9))-(9+9))
digits: 10

extended result (((((9!)/(9+9))+99)/9)+((9*9)*9))
digits: 9
0:21.87
```

3.10 Nicht genügend Parameter

Falls dem Programm nicht genügend Parameter übergeben werden, gibt es einen Fehler aus:

```
$ python main.py
usage: main.py [-h] [--verbose] number digit
main.py: error: the following arguments are required: number, digit
```

3.11 Falsche Ziffer

Falls der Parameter der Ziffer keine Ziffer ist, gibt das Programm einen Fehler aus:

```
$ python main.py 1 -1
Error: -1 is not a digit, exiting...
```

3.12 Zusätzliche Ausgabe

```
$ python main.py 2019 1 -v
looking for normal shortest
generated split table with digits: 1
generated split table with digits: 2
generated split table with digits: 3
generated split table with digits: 4
generated split table with digits: 5
generated split table with digits: 6
generated split table with digits: 7
generated split table with digits: 8
found (((1111*(1+1))-1)-((111+(1-11))*(1+1))) with 15 digits, looking if shorter is
    ↪ possible
generated split table with digits: 9
found (((111*(1+1))*(11-(1+1)))-((1-11)-11)) with 14 digits, looking if shorter is
    ↪ possible
generated split table with digits: 10
found ((((11111-1)/11)*(1+1))-1) with 11 digits, looking if shorter is possible

looking for extended shortest
generated split table with digits: 1
generated split table with digits: 2
generated split table with digits: 3
generated split table with digits: 4
generated split table with digits: 5
generated split table with digits: 6
found ((((1+1)^11)+1)+(((1+1)+1)*(1-11))) with 11 digits, looking if shorter is possible
generated split table with digits: 7
found ((((1+1)^11)+1)-((11-1)*((1+1)+1))) with 11 digits, looking if shorter is possible
generated split table with digits: 8
found (((1+1)+1)*(((111+1)*(((1+1)+1)!))+1)) with 11 digits, looking if shorter is
    ↪ possible
generated split table with digits: 9
found (((1+1)^11)-(((11-1)*((1+1)+1))-1)) with 11 digits, looking if shorter is possible

normal result ((((11111-1)/11)*(1+1))-1)
digits: 11

extended result (((1+1)^11)-(((11-1)*((1+1)+1))-1))
digits: 11
```

4 Quellcode (ausschnittsweise)

Hier drucke ich die Funktionen `add_to_table`, `generate`, `scan` und `find_shortest` ab. Die Klassendefinitionen für `Term`, `Number`, `UnaryOperation` sowie `BinaryOperation` werden ausgelassen. Ausgelassen wird auch das Einlesen der Parameter und die Ausgabe des durch `find_shortest` gefundenen Ergebnisses.

```python
def add_to_table(term, table, extended=False):
    val = term.value()
    if extended and val >= 3 and val <= MAX_FACTORIAL:
        add_to_table(UnaryOperation(term, UnaryOperation.OP_FAC), table, extended=extended)
    digits = term.number_of_digits()
    if val in table and table[val][1] < digits:
        return
    table[val] = (term, digits)


def generate(digit, num_digits, aggregated_table, split_table, extended, debug=False):

    current_split_table = split_table[num_digits]

    # Add 3, 33, 333 etc
    num = int(str(digit)*num_digits)
    add_to_table(Number(num), current_split_table, extended)
    swap = False

    for op1_num_digits in range(1, num_digits // 2 + 1):
        op2_num_digits = num_digits - op1_num_digits
        for op1_k in split_table[op1_num_digits]:
            op1_v = split_table[op1_num_digits][op1_k][0]
            for op2_k in split_table[op2_num_digits]:
                op2_v = split_table[op2_num_digits][op2_k][0]
                # Make sure that op1_k > op2_k
                if op1_k < op2_k:
                    op1_k, op2_k = op2_k, op1_k
                    op1_v, op2_v = op2_v, op1_v
                    swap = True


                add_to_table(BinaryOperation(op1_v, op2_v, BinaryOperation.OP_ADD), current_split_table, extended)
                add_to_table(BinaryOperation(op1_v, op2_v, BinaryOperation.OP_SUB), current_split_table, extended)
                add_to_table(BinaryOperation(op2_v, op1_v, BinaryOperation.OP_SUB), current_split_table, extended)
                add_to_table(BinaryOperation(op1_v, op2_v, BinaryOperation.OP_MULT), current_split_table, extended)
                if extended and op1_k >= 2 and op2_k >= 2:
                    if op2_k * math.log(op1_k, 10) <= MAX_DIGITS:
                        add_to_table(BinaryOperation(op1_v, op2_v, BinaryOperation.OP_POW), current_split_table, extended)
                    if op1_k * math.log(op2_k, 10) <= MAX_DIGITS:
                        add_to_table(BinaryOperation(op2_v, op1_v, BinaryOperation.OP_POW), current_split_table, extended)

                if op2_k != 0:
                    res = op1_k / op2_k
                    resint = int(res)
                    if res == resint:
                        add_to_table(BinaryOperation(op1_v, op2_v, BinaryOperation.OP_DIV), current_split_table, extended)

                if swap:
                    op1_k, op2_k = op2_k, op1_k
                    op1_v, op2_v = op2_v, op1_v
                    swap = False
    if debug:
        print("generated split table with digits:", num_digits, file=sys.stderr)

    for k in split_table[num_digits]:
        if k not in aggregated_table:
            aggregated_table[k] = split_table[num_digits][k]
    return aggregated_table, split_table
```

```python

def scan(number, digit, aggregated_table, extended):
    results = set()
    if number in aggregated_table:
        results.add(aggregated_table[number][0])

    for j in aggregated_table:
        if number - j in aggregated_table:
            results.add(BinaryOperation(aggregated_table[j][0], aggregated_table[number-j][0], BinaryOperation.OP_ADD))
        if number + j in aggregated_table:
            results.add(BinaryOperation(aggregated_table[number+j][0], aggregated_table[j][0], BinaryOperation.OP_SUB))
        if j - number in aggregated_table:
            results.add(BinaryOperation(aggregated_table[j][0], aggregated_table[j-number][0], BinaryOperation.OP_SUB))

        if j != 0:
            if (number*j) in aggregated_table:
                results.add(BinaryOperation(aggregated_table[number*j][0], aggregated_table[j][0], BinaryOperation.OP_DIV))

            res = j / number
            resint = int(res)
            if res == resint and resint in aggregated_table:
                results.add(BinaryOperation(aggregated_table[j][0], aggregated_table[resint][0], BinaryOperation.OP_DIV))

            res = number / j
            resint = int(res)
            if res == resint and resint in aggregated_table:
                results.add(BinaryOperation(aggregated_table[number/j][0], aggregated_table[j][0], BinaryOperation.OP_MULT))

        if extended and number > 1 and j > 1:
            res = number ** (1/j)
            if res in aggregated_table and res != 1.0:
                results.add(BinaryOperation(aggregated_table[res][0], aggregated_table[j][0], BinaryOperation.OP_POW))
            res = math.log(number, j)
            if res in aggregated_table:
                results.add(BinaryOperation(aggregated_table[j][0], aggregated_table[res][0], BinaryOperation.OP_POW))


    if extended and j >= 3 and j <= 60 and j in FACTORIALS and FACTORIALS[j] in aggregated_table:
        results.add(UnaryOperation(aggregated_table[FACTORIALS[j]][0], UnaryOperation.OP_FAC))


    if len(results) == 0:
        return None

    # Use term with fewest digits. If there are multiple terms with the same number of digits, use the one that has fewer characters
    res = 0
    n = math.inf
    c = math.inf
    for r in results:
        nr = r.number_of_digits()
        nc = len(str(res))
        if nr < n or (nr == n and nc < c):
            res = r
            n = nr
            c = nc
    return res

def find_shortest(number, digit, extended, debug=False):
    aggregated_table = {}
    split_table = defaultdict(dict)

    i = 1
```

```python
    res_n = math.inf

    # Generate tables until shortest result will be available with scan()
    while i <= res_n - 2:
        aggregated_table, split_table = generate(args.digit, i, aggregated_table, split_table, extended, debug=debug)
        res = scan(number, digit, aggregated_table, extended)
        if res is not None:
            res_n = res.number_of_digits()
            if debug:
                print("found", res, "with", res.number_of_digits(), "digits, looking if shorter is possible")
        i += 1
    return res
```

code.py