Lukas Kesch
lukas.kesch@gmail.com
Teilnahme-ID: 53757

Aufgabe 2: Geburtstag

38. Bundeswettbewerb für Informatik - Runde 2 (20.04.2020)

1. Aufgabenstellung

Informationen in diesem Abschnitt sind aus dem offiziellen Aufgabenblatt entnommen (vgl. Aufgabenblatt o. J.).

1.1 Grundlegende Aufgabe

Zu einer beliebigen Jahreszahl (also einer natürlichen Zahl $n \geq 1$ oder der Zahl 0) soll ein kürzestmöglicher Term aus natürlichen Zahlen und Rechenoperationen erstellt werden. Die Länge eines Terms wird über die Anzahl der vorkommenden Ziffern definiert und nicht über seine Zeichenlänge (Anzahl der benötigten Zeichen, um den Term darzustellen).

1.2 Einschränkungen

Der Term muss folgende Bedingungen erfüllen:

- Er darf nur aus einer einzigen vorgegebenen Ziffer (zwischen 1 bis 9) bestehen. Es ist jedoch legitim, aus dieser Ziffer mehrstellige Zahlen zu bilden. Wenn etwa 1 die gewählte Ziffer ist, so darf auch 11, 111, 1111, usw. verwendet werden.
- In Teil A dürfen nur die 4 Grundrechenarten (+, -, *, /) verwendet werden, um die Ziffern / Terme miteinander zu verbinden.
- In Teil B ist auch die Verwendung von Potenz- und Fakultätsfunktionen erlaubt.

1.3 Lösung

Das Programm soll den kürzestmöglichen Term für eine gegebene Zahl und Ziffer berechnen, der nur die vier Grundrechenarten beinhaltet, und dann überprüfen, ob ein noch kürzerer Term mit dem Einsatz von Potenz- und Fakultätsfunktionen gebildet werden kann.

2. Lösungsidee

2.1 Systematische Erstellung aller n-stelligen Terme

Um den tatsächlich kürzesten Term zu ermitteln, erstellt das Programm systematisch alle möglichen Terme / Zahlen. Hierbei werden zuerst alle einstelligen Terme (also alle Terme der Länge 1) erstellt, dann alle zweistelligen, danach alle dreistelligen und so weiter, bis ein erstellter Term im Wert der eingegebenen Zahl gleicht. Dieses Vorgehen garantiert ein optimales Ergebnis, sofern kein Teilterm die Grenze des verwendeten Datentyps sprengt.

Falls dies jedoch der Fall sein sollte, so kann es eine oder mehrere eventuell bessere Lösungen geben. Diese enthalten dann mindestens einen Teilterm, der in seinem Wert die Grenze des verwendeten Datentyps sprengt. Um dennoch Klarheit über den Gültigkeitsbereich der Lösung zu haben, wird erfasst, ob die Grenze von mindestens einem Teilterm überschritten wurde. Somit ist bekannt, ob die Lösung allgemein oder möglicherweise nur in den Grenzen des Datentyps gültig ist.

Das Programm bricht die Suche nach dem Auffinden der ersten Lösung ab, könnte aber auch alle Terme der aktuellen Länge noch vollständig durchgehen, um alle Lösungen zu ermitteln (was in der Aufgabenstellung aber nicht gefordert ist).

2.2 Speichern aller Ergebnisse

Um die Verwendung von Termen, die den gleichen Wert wie ein bereits kreierter Term gleicher oder kürzerer Länge aufweisen, zu verhindern, werden alle Ergebnisse als Paare aus Wert und zugehörigem Term zwischengespeichert. Ein Term wird nur gespeichert und weiterverwendet, wenn dessen Wert in der Liste noch nicht vorhanden ist.

Die Speicherung der Paare aus Wert und zugehöriger Termlänge soll in einer laufzeitschonenden Datenstruktur realisiert werden. Denn diese wird sehr oft nach Elementen durchsucht und um neue erweitert.

2.3 Getroffene Einschränkungen

Da Jahreszahlen mit der Ausnahme der Zahl null ausschließlich natürliche Zahlen sind, ist es legitim, alle negativen Terme zu verwerfen. Der Subtraktionsoperator erstellt durch die Kombination zweier Terme zu jedem negativen Ergebnis auch ein positives, welches dem Betrag nach dem negativen gleicht (a-b = | b-a | für a>b) und wegen a + (-b) = a - b müssen keine Teilterme mit negativem Wert verwendet werden. Für Teilaufgabe A reduziert dies die Anzahl der Terme in jedem Schritt um ⅛.

Bei der Addition und Multiplikation zweier Terme wird das Kommutativgesetz (a+b = b+a und a*b = b*a) beachtet, um das Erstellen von Dubletten zu vermeiden. Die Anzahl der Terme in Teil A wird dadurch in jedem Schritt um ¼ reduziert.

Bei der Division werden nur Terme gespeichert, die einen natürlichen Wert aufweisen, da die Aufgabenstellung so zu verstehen ist, dass alle Zwischenergebnisse natürlichzahlig sein müssen und Terme wie 6 * (5 / 3) nicht zulässig sind, sondern als (6 / 3) * 5 geschrieben werden müssen. Zudem wird stets nur durch die kleinere Zahl dividiert. Wenn eine kleinere Zahl durch eine Größere geteilt wird, kann das Ergebnis nicht natürlich sein. Mithilfe dieses Vorgehens wird die Anzahl der Terme für Teil A in jedem Schritt um mindestens ⅛ reduziert.

Durch die Kombination der gewählten Einschränkungen wird die Anzahl der zu erstellenden Terme für Teilaufgabe A in jedem Schritt mehr als halbiert. Dies spart exponentiell Rechenzeit ein. Eine genauere Analyse befindet sich in dem Kapitel der *Analyse des Termwachstums* in den Abschnitten *Recheneinsparung durch Termbeschränkung* und *Recheneinsparung durch Termüberprüfung*.

2.4 Änderungen für Teil B

Grundlegend gleicht das Vorgehen im Teil B dem des Teils A. Der Unterschied ist, dass in Teil B auch Potenz- und Fakultätsfunktionen verwendet werden dürfen. Dies lässt die zusätzlich möglichen Terme in ihrem Wert und ihrer Anzahl nach extrem ansteigen. Der Programmablauf ist aber größtenteils gleich. Noch wichtiger als in Teil A ist jetzt, dass vor dem Erstellen jedes Termes überprüft wird, ob dieser eine festgelegte Grenze (z. B. long Integer) überschreitet. Ohne festgelegte Grenze würde sich das Programm in einer Endlosschleife befinden und irgendwann den Arbeitsspeicher aufbrauchen, da die Fakultätsfunktion für jeden Term unendlich viele weitere Terme gleicher Länge produzieren kann: a!, (a!)!, ((a!)!)! usw. Somit müsste selbst bei der Verwendung von BigInteger-Typen (Integer mit beliebig großem Wert) eine Grenze festgelegt werden.

Aufgrund der Grenzsetzung kann es vorkommen, dass kürzere Terme, die aus ein oder mehreren Teiltermen bestehen, die die Obergrenze überschreiten, und Lösung sind, übersehen werden. Deshalb kann der Algorithmus in Teil B nicht das allgemein beste Ergebnis garantieren, lediglich das beste in den Grenzen des ausgewählten Datentyps.

3. Umsetzung

3.1 User Interface

Das User Interface ist so gestaltet, dass der Benutzer eine beliebige Jahreszahl und die zu verwendende Ziffer (1-9) eingeben kann. Er kann zudem die Modulo Erweiterung (Siehe Kapitel: *Erweiterungen* Abschnitt: *Modulo*) aktivieren, als auch das Speichern von statistischen Daten in eine csv-Datei. Daraufhin besteht die Möglichkeit, Teil A beziehungsweise B lösen zu lassen. Nach der Berechnung wird dem Benutzer der entsprechende Term inklusive statistischer Daten ausgegeben. Zu diesen gehört die Anzahl an untersuchten Termen, die Anzahl an ungültigen Termen (deren Ergebnis ist nicht mit dem Datentyp long kompatibel), die Anzahl an Termen, deren Wert bereits existiert, und die Anzahl an schlussendlich verwendeten Termen. Außerdem wird die Anzahl an theoretisch möglichen Termen ohne Optimierung (Teil A) und die Anzahl an Ziffern des Termes sowie die benötigte Rechenzeit (single-core) ausgegeben.

3.2 Allgemeiner Programmablauf

Das Programm nimmt die Eingabedaten entgegen und konvertiert diese in geeignete Datentypen. Anschließend wird die CalculateTerm Methode aufgerufen, die den besten Term ermittelt. Das genaue Vorgehen dieser Methoden wird in den folgenden Abschnitten erläutert: *Termerstellung*, *Termbeschränkung*, *Termüberprüfung*. Anschließend wird das Ergebnis ausgegeben.

Die Berechnungen von Teil A oder Teil B alleine arbeiten unabhängig voneinander. Die Berechnung von Teil A und Teil B zusammen jedoch nicht. Die Berechnung in Teil B wird hier abgebrochen, wenn klar ist, dass kein besseres Ergebnis als in Teil A erreicht werden kann.

3.3 Datenstrukturen

Um den Buchhaltungsaufwand für die Terme gering zu halten, ist einen objektorientierter Ansatz sinnvoll.

3.3.1 Grenzbestimmender Variablentyp

Es stellt sich heraus, dass ein herkömmlicher 32-bit signed integer bereits zu klein für die Speicherung mancher Termwerte ist. Die meisten Beispielaufgaben erstellen, spätestens bei einer Termlänge von elf, Terme, die in ihrem Wert den int32.MaxValue (2.147.483.647) überschreiten. Deshalb wird der größere Datentyp long verwendet. Dieser ist der größte limitierende Datentyp für die Speicherung von Ganzen Zahlen. Er ist ein 64-bit signed integer mit einer Grenze (long.MaxValue) von 9.223.372.036.854.775.807.

3.3.2 Term-Klasse

Diese (abstrakte) Klasse bildet das Grundgerüst. Sie beinhaltet zwei virtuelle Methoden, eine, die den Term als String ausgibt, und eine weitere, die den Wert des Termes als long-Wert zurückgibt. Alle weiteren Klassen führen direkt (1. Generation) auf diese zurück und überschreiben die virtuellen Methoden mit ihrer spezifischen Implementierung.

3.3.3 Literal-Klasse

Diese Klasse speichert ausschließlich Zahlen beziehungsweise die Aneinanderreihungen der vom Benutzer angegebenen Ziffer. Bei dem Konstruieren eines Objektes dieser Klasse muss der Wert der Zahl übergeben werden. Nach der Erstellung kann dieser Wert als long-Wert und String zurückgegeben werden.

3.3.4 Operatorklassen

Es gibt sechs (mit der später noch folgenden Modulo-Erweiterung sieben) verschiedene Operatorklassen. Diese repräsentieren die folgenden Rechenarten: Addition, Subtraktion, Multiplikation, Division, (Modulo), Potenz sowie die Fakultät. Jede dieser Klassen besitzt einen Konstruktor, zwei objektspezifische und eine statische Methode.

Der Konstruktor nimmt zwei (für die Fakultät ein) Termobjekte beziehungsweise deren Zeiger entgegen, aus denen der neue Term gebildet wird. Diese werden in dem neuen Termobjekt wieder nur als Zeiger (die auf den entsprechenden Term in der Term-Liste zeigen) hinterlegt. So muss jeder Term tatsächlich nur einmal gespeichert werden. Dies umgeht die Erzeugung von Kopien, die zusätzlich Arbeitsspeicher, Kopier- und Rechenzeit benötigen würden.

Die zwei objektspezifischen Methoden können den Term als String als auch den Wert des Terms als long-Wert ausgeben.

Die statische Methode nimmt wie der Konstruktor zwei Teilterme entgegen und überprüft daraufhin, ob der verknüpfte Term die long Grenze sprengt oder einen nicht natürlichzahligen Wert aufweist. Genauere Implementierungsdetails zur Überprüfung sind in dem Abschnitt *Termbeschränkung* zu finden.

Sollte der long.MaxValue überschritten werden und das Programm sich in der Berechnung des Teils A befinden, so wird ein Flag geändert, damit nach der Berechnung der Gültigkeitsbereich der ermittelten Lösung eindeutig ist.

3.3.5 Speicherung der Terme

Alle Termobjekte werden (wieder nur über Zeiger) in einer verschachtelten Liste gespeichert. Die äußere Liste speichert in jedem Index eine weitere Liste. Der Index der äußeren gibt die Termlänge n an. An diesem Index befindet sich dann die Liste aller Terme der Länge n. Diese Termlisten werden mit einer Anfangskapazität von 100.000 initialisiert, um unnötiges Zeigerkopieren zu vermeiden, wenn die standardmäßige Anfangskapazität überschritten wird.

3.3.6 Speicherung der Termwerte

Die Werte aller Terme werden in einem SortedDictionary (vgl. SortedDictionary Class (System.Collections.Generic) o. J.) gespeichert. In diesem können Key Value Paare gespeichert werde. Der Key ist jeweils der Wert des Terms und der Value ein Zeiger auf den Term. Diese Datenstruktur ist vorteilhaft, da sie die Paare ihren Keys nach in ein binären Suchbaum einordnet und somit schnelles Einfügen und Suchen ermöglicht. Eine SortedList eignet sich nicht. Sie benötigt zwar weniger Speicher und erlaubt gleich schnelles Suchen, das Einfügen neuer Elemente ist jedoch nicht laufzeitschonend. Eine genauere Betrachtung befindet sich in dem Kapitel Laufzeitanalyse unter dem Abschnitt Nichtlineare Zunahme der Laufzeit pro Termüberprüfung.

3.4 Termerstellung

Wie bereits in der Lösungsidee beschrieben werden systematisch alle n-stelligen Terme erstellt. Implementiert ist dies mit einer hochzählenden for-Schleife über n. Bevor diese jedoch startet, wird die Ziffer als Literal erstellt und überprüft, ob sie bereits Lösung ist. Die Schleife startet dann mit der Erstellung aller zweistelligen Terme. Für jeden weiteren Durchlauf erstellt sie alle um eins längeren Terme und die um eins längere Zahl, bestehend aus der gegebenen Ziffer. Sie läuft solange, bis ein erstellter Term die verlangte Jahreszahl als Wert aufweist.

Zu Beginn jedes Durchlaufes wird zunächst der nun mögliche Literal erstellt und überprüft. Daraufhin wird, wenn sich das Programm in Teil B befindet, auf alle um eins kürzeren Terme so oft wie möglich die Fakultät angewandt und deren Werte werden analysiert. Nun folgen

drei ineinander verschachtelte for-Schleifen. Die erste durchläuft alle Termlängen bis maximal zur Hälfte der zu erstellenden Termlänge. Die zweite durchläuft alle Terme der von der äußeren Schleife festgelegten Termlänge. In ihr wird die benötigte Länge des Termes ermittelt, um mit der von der eins weiter außen befindlichen Schleife festgelegten Termlänge die von der äußersten Schleife geforderte zu erreichen. Die dritte Schleife durchläuft nun alle Terme der vorig berechneten Länge. In dieser werde nun die zwei Terme (aktueller Term der zweiten und der dritten Schleife) der CreateTerms Methode übergeben. Wenn eine Kombination der beiden Terme die Zielzahl ergibt, so wird die Berechnung gestoppt, da nun das erste und beste Ergebnis gefunden wurde.

Die CreateTerms Methode kombiniert die Terme mit Hilfe der Operatoren. Zunächst werden die beiden Terme nach der Höhe ihres Ergebnisses geordnet, um somit die bereits in dem Kapitel *Lösungsidee* im Abschnitt *Getroffenen Einschränkungen* erwähnten Einschränkungen leicht umzusetzen. Deswegen muss die erste Schleife in der Hauptschleife auch nur bis zur Hälfte hochgezählt werden. Der Term mit dem höheren Wert steht stets auf der linken Seite des Operators, während der Term mit dem geringeren Wert auf der rechten Seite steht. Der einzige Operator, der die Terme auf zwei Wegen kombinieren muss, ist der Potenzoperator in Teil B. Wenn a und b die beiden Teilterme sind und a größer b gilt, dann werden nun folgende Terme produziert: a+b, a-b, a*b, a/b, (a%b), a^b und b^a.

3.5 Termbeschränkung

Für die Division und generell für Teil B muss eine wichtige Einschränkung getroffen werden. Vor dem Erstellen jedes Termes muss überprüft werden, dass dieser ein natürlichzahliges Ergebnis innerhalb des long-Wertebereiches aufweist.

Bei der Division wird die Natürlichkeit eines Ergebnisses überprüft, indem die beiden Terme mit dem Modulo Operator verrechnet werden. Ist das daraus resultierende Ergebnis null, so liefert die Division ein ganzzahliges Ergebnis beziehungsweise ein natürlichzahliges Ergebnis, da alle gespeicherten Terme stets positiv sind.

Für die Addition wird kontrolliert, dass die Summe der beiden Terme nicht höher als die long-Grenze ist. Dies wird umgesetzt, indem von long.MaxValue das Ergebnis beider Terme subtrahiert wird. Sollte das Ergebnis negativ sein, so sprengt die Summe diese Grenze und der Term wird verworfen. Wegen long.MinValue = - long.MaxValue - 1 kann es dabei zu keinem Überlauf kommen.

Für die Multiplikation und Potenzierung bedarf es eines Tricks, um laufzeitschonend zu überprüfen, dass das Produkt und der Potenzwert sich innerhalb der long-Grenze befinden. Eine Umwandlung unter Nutzung einer try-catch Verschachtelung mit integriertem Parsen des Ergebnisses zu einem long-Wert besitzt einen zu großen Overhead und ist somit nicht optimal. Anstelle dessen werden Logarithmengesetze genutzt. Diese führen nach ein- beziehungsweise zweifacher Anwendung die beiden Operatoren auf eine simple Addition zurück. Nun muss nur die Summe zweier Gleitkommazahlen bestimmt werden.

Bei der Fakultät stellt sich heraus, dass 21! als erster Term bereits die long-Grenze sprengt. Somit wird das Anwenden der Fakultät auf Zahlen größer 20 untersagt.

3.6 Termüberprüfung

Alle Terme werden in einem binären Suchbaum (vgl. SortedDictionary Class (System.Collections.Generic) o. J.) als Paar aus Term-Wert und Term gespeichert, sodass vor der Speicherung eines jeden neuen Termes überprüft werden kann, ob dessen Wert bereits vorhanden ist. Sollte das der Fall sein, so wird der neue Term nicht gespeichert. Da die Terme systematisch der Länge nach erstellt werden, ist der neuere länger oder gleich lang dem alten. Somit ist es legitim, den neuen zu verwerfen.

Ist das Ergebnis eines Termes null, so wird dieser auch verworfen, da die Zahl 0 keinen Nutzen in der Erstellung von kürzestmöglichen Termen mit natürlichen Ergebnissen hat.

Wenn das Ergebnis der angegebenen Jahreszahl gleicht, so wird die weitere Termerstellung abgebrochen. Der Term wird dann dem Benutzer mit weiteren Daten ausgegeben.

3.7 Sonderfall: Jahreszahl = 0

Sollte es der Fall sein, dass die eingegebene Jahreszahl gleich null ist, so findet eine Fallunterscheidung statt, da der Algorithmus Terme mit Nullen als Ergebnis aufgrund der Laufzeitoptimierung verwirft. Für diesen Fall wird immer der Term (Ziffer-Ziffer) ausgegeben. Dieser ist sowohl in Teil A als auch B für alle Ziffern der kürzeste Term und somit die allgemein beste Lösung für diesen Fall.

4. Analyse des Termwachstums

Das Termwachstum meint die Zunahme der Anzahl der Terme. Diese hängt bis auf die letzte Stufe, in der die Lösung gefunden wird, nur von der Ziffer ab, nicht von der Jahreszahl.

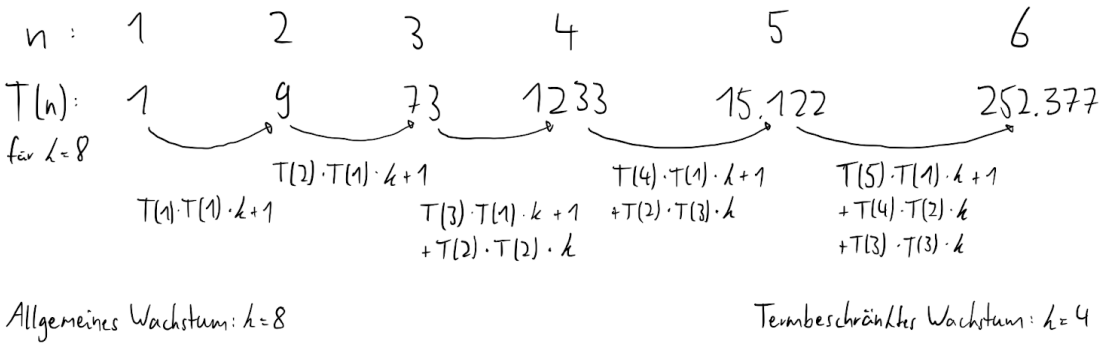
4.1 Teil A

4.1.1 Allgemeines Termwachstum

Das Wachstum kann weder explizit noch rekursiv exakt beschrieben werden, da es sich im Verlauf auf eine unterschiedliche Anzahl an Vorgängern (die verwendeten Terme) bezieht.

Die Anzahl der Terme jeder Stufe kann über die Anzahl der verwendeten Verknüpfungen exakt aus der Anzahl der verwendeten Terme der Vorgängerstufen berechnet werden. Diese wird also rekursiv aus den Anzahlen der Terme der Stufe 1 bis n-1 ermittelt. Somit kann das Wachstum als Reihe aufgefasst werden.

Die Reihe / Anzahl an möglichen Termen nimmt mit der Termlänge n exponentiell zu, da der Bestand für jede Inkrementierung von n um eins mit mindestens dem Faktor acht anwächst. Dies ist aus dem nachstehenden Schema über den Wachstumsverlauf erkennbar. In jedem Schritt wird die Anzahl an möglichen Termen um mindestens das Produkt aus der Anzahl der vorherigen (n-1) Terme und die Anzahl der Kombinationsmöglichkeiten (Rechenoperatoren * 2) erhöht. Die folgende Grafik bildet den Beginn des Wachstumsvorgangs ab.



Ein Graph dieser Reihe befindet sich im Abschnitt *Diagramm 1: Mögliche Termeinsparung*.

4.1.2 Recheneinsparung durch Termbeschränkung

Wie bereits in dem Kapitel *Lösungsidee* im Abschnitt *Getroffene Einschränkungen* und in dem Kapitel *Umsetzung* Abschnitt *Termbeschränkung* beschrieben wird, wird auf das Erstellen irrelevanter Terme verzichtet. Diese weisen eine der folgenden zwei Eigenschaften auf: unzulässiger Wert (negativ oder nicht natürlichzahlig) oder unnötiger Wert (KG).

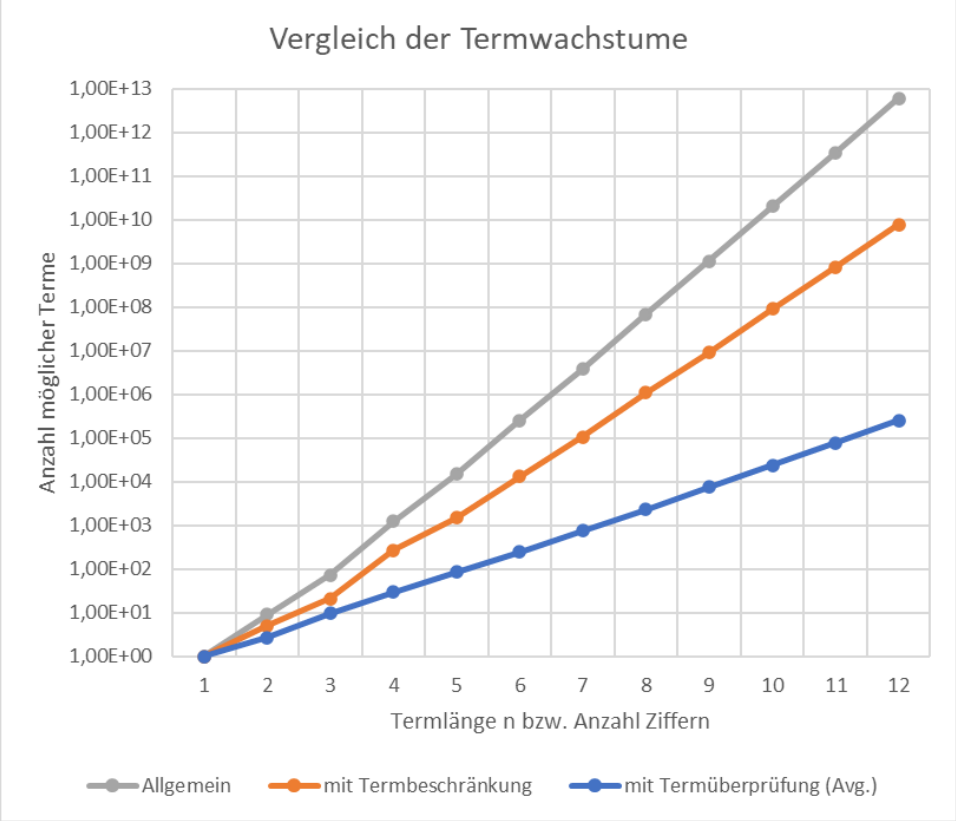
Der Wachstumsvorgang gleicht dem Allgemeinen, bis auf die Konstante k, die nun vier beträgt. Die Anzahl der Terme nimmt mit der Termlänge n nun mit einem geringeren Wachstumsfaktor zu.

Anhand des Diagramms 1 (siehe übernächster Abschnitt), welches die beiden Reihen in Relation abbildet, ist zu erkennen, dass die Termbeschränkung eine sehr sinnvolle Maßnahme ist. Bei einer Termlänge von 8 wird lediglich mit 1 % der theoretisch möglichen Terme gerechnet und bei einer Termlänge von 12 nur noch mit knapp 0,1 %.

4.1.3 Recheneinsparung durch Termüberprüfung

Wie sich die Maßnahmen der Termüberprüfung (Verwerfen von Termen mit einem nicht natürlichen, zu hohen oder bereits vorhandenen Wert. Siehe Kapitel *Lösungsidee*, Abschnitt *Getroffene Einschränkungen* und Kapitel *Umsetzung*, Abschnitt *Termüberprüfung*) auf das Termwachstum auswirken, kann analytisch nicht vorhergesagt werden. Um aber dennoch diese Maßnahme bewerten zu können, wird das durchschnittliche Termwachstum statistisch ermittelt. Der dazu benutzte Datensatz befindet sich im Anhang. Es stellt sich heraus, dass auch diese Maßnahme sehr sinnvoll ist und mit einem massiven Laufzeitvorteil einhergeht. Bereits bei einer Termlänge von 5 (Reduktion um 3) wird mit weniger als 1 % der theoretisch möglichen Terme gerechnet und bei einer Termlänge von 6 (Reduktion um 6) nur noch mit 0,1 %.

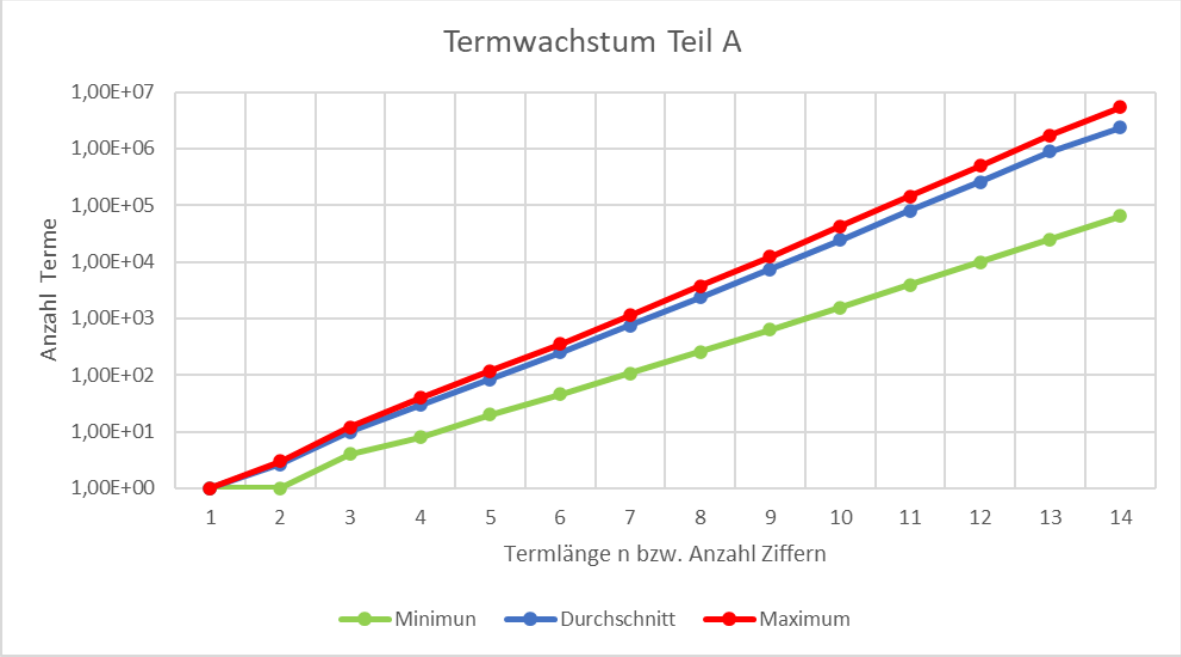
4.1.4 Diagramm 1: Mögliche Termeinsparung



4.1.5 Durchschnittliches Termwachstum

Mithilfe des ermittelten Datensatzes kann das Termwachstum (Wachstum mit Termbeschränkung und Überprüfung) noch genauer betrachtet werden. Das Diagramm im nächsten Abschnitt stellt dies dar. Die Kurve des minimalen Termwachstumes bestimmt die Ziffer 1, die des maximalen legen die Ziffern 7 und 8 fest. Daraus folgt, dass für die Ziffer 1 am wenigsten Terme mit unterschiedlichem Ergebnis erstellt werden können und für die Ziffern 7 und 8 am meisten. Es fällt auf, dass der Durchschnitt sehr nahe an dem maximalen Wert liegt. Somit ist der minimale Wert eher unwahrscheinlich.

4.1.6 Diagramm 2: Termwachstum (durchschnittlich)



4.2 Teil B

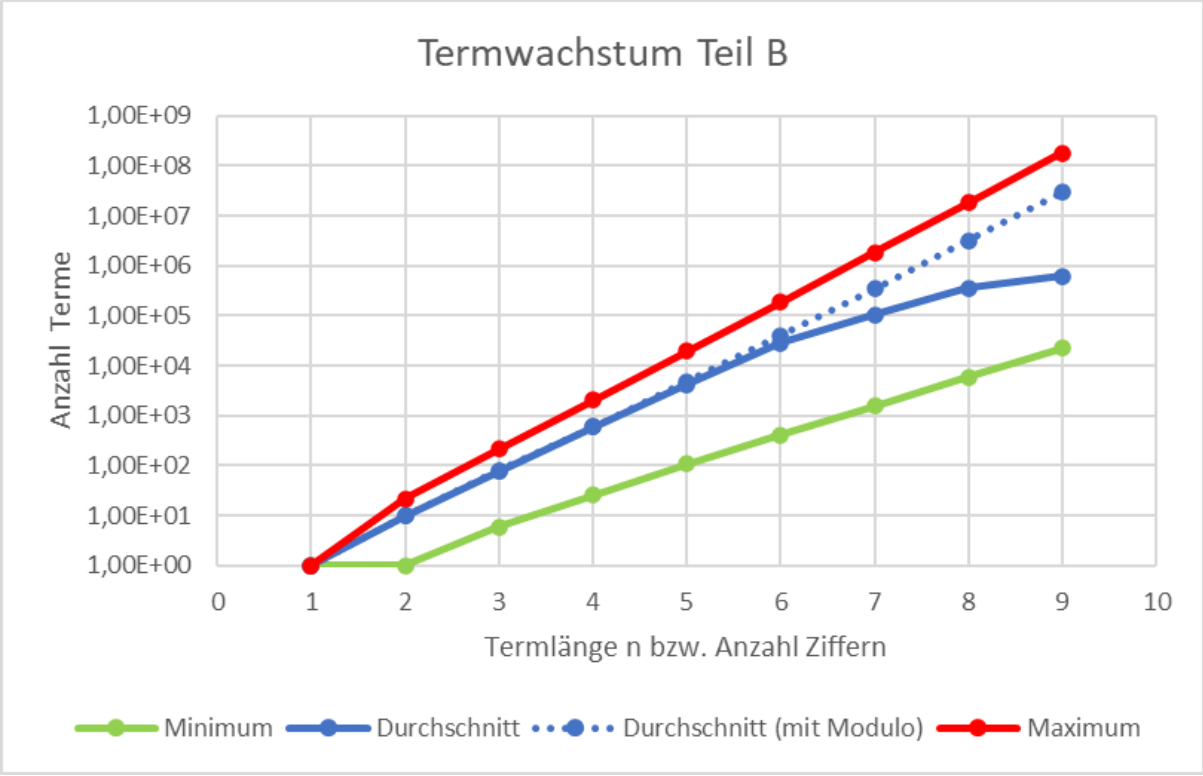
4.2.1 Durchschnittliches Termwachstum

Für den Teil B kann aufgrund der mehrfachen Anwendung der Fakultät kein maximales Termwachstum ohne jegliche Beschränkungen vorhergesagt werden. Somit kann das Wachstum mit Termüberprüfung nicht gegen das maximale Wachstum verglichen werden. Es ist aber davon auszugehen, dass die Einsparungen pro Ziffer höher sind als in Teil A, da Teil B allgemein eine höhere Termzunahme pro Ziffer aufweist.

Das durchschnittliche Termwachstum mit der Termbeschränkung und Termüberprüfung kann aber statistisch erfasst werden. Hierfür befindet sich im folgenden eine Darstellung über das durchschnittliche Termwachstum. In dieser befinden sich drei durchgezogene Kurven. Eine beschreibt die durchschnittliche Termanzahl, eine weitere die maximal mögliche Anzahl an Termen (Ziffer 3) und die letzte die minimalst möglichen Terme (Ziffer 1) für jede Termlänge. Es fällt auf, dass bereits bei einer Termlänge von 9 die Anzahl der Terme um ein 10.000 faches abweichen kann.

Die gepunktete Kurve bezieht sich auf die Modulo-Erweiterung. Mehr dazu in dem Kapitel *Erweiterungen* im Abschnitt *Modulo Termwachstum*

4.2.2 Diagramm 3: Durchschnittliches Termwachstum



4.3 Obergrenze

Unabhängig von Teil A oder B kommt das Programm grob ab sechsstelligen Jahreszahlen in Kombination mit einer ungünstigen Ziffer an seine Grenzen. Die limitierende Ressource ist der Arbeitsspeicher (16GB). Denn jeder Term erfordert Speicherplatz.

5. Laufzeitanalyse

5.1. Problematik

Es ist sehr schwierig, das Laufzeitverhalten des Programms vorauszusehen. Es kann lediglich eine grobe Abschätzung anhand der Höhe der Jahreszahl gemacht werden. Die tatsächliche Laufzeit kann aber ein Vielfaches vom Durchschnitt abweichen, allein für vierstellige Zahlen ist die Standardabweichung in Teil A und B fast genauso hoch wie der eigentliche Durchschnitt. Deshalb ist es sinnlos, diesen Ansatz zu verfolgen. Es bietet sich jedoch an, die Laufzeit pro Term(erstellung) zu analysieren.

5.2 Lineare Zunahme der Laufzeit pro Termerstellung

Das reine Erstellen und Beschränken eines Termes ist ein (nahezu) linearer Prozess. Hierbei wird für jeden Term geprüft, ob er ein natürliches Ergebnis im long-Wertebereich aufweist, und wenn ja, so wird dieser als neues Objekt initialisiert. Hierfür werden k Schritte benötigt. Daraus resultiert eine Laufzeit von O(k). Diese ist unabhängig von der Termlänge n oder von der Gesamtanzahl an Termen.

5.3 Nichtlineare Zunahme der Laufzeit pro Termüberprüfung

Die Überprüfung eines Termes ist kein linearer Prozess. Denn für jeden Term wird sein Ergebnis mit den Ergebnissen aller vorherigen Terme verglichen und dieser anschließend der Liste hinzugefügt.

Wenn dies mit einer einfachen Liste / Array mit n Elementen umgesetzt wird, beträgt die Laufzeit für das Durchsuchen O(n) und die für das Hinzufügen eines Elementes O(1). Die Gesamtkomplexität für eine Termerstellung und Überprüfung beträgt dann O(k + n + 1). Dies entspricht O(n).

Sollten die Ergebnisse hingegen sortiert sein, kann die Laufzeit des Durchsuchens unter Einsatz der binären Suche von O(n) auf O(log(n)) reduziert werden. Somit beträgt die Gesamtkomplexität O(k + log(n) + x), wobei x die Zeit für das Einfügen eines Elementes in die Datenstruktur beträgt. Um diesen Laufzeitvorteil des Suchens auszunutzen, wird eine sich sortierende Datenstruktur benötigt. Hierfür bietet sich eine Art SortedList oder ein binärer Suchbaum an.

Eine SortedList (vgl. SortedList Class (System.Collections.Generic) o. J.) speichert seine Werte in einem Array und ist somit speicherschonend und schnell zu durchlaufen. Das Einfügen eines neuen Elementes in die Datenstruktur hat eine Laufzeit von O(n), da das Array aufgrund seiner Größenänderung kopiert werden muss. Somit beträgt die Gesamtkomplexität O(k + log(n) + n) = O(n).

Ein binärer Suchbaum (In C# umgesetzt mit: SortedDictionary vgl. SortedDictionary Class (System.Collections.Generic) o. J.) benötigt mehr Speicher als die SortedList und hat eine höhere Durchlaufzeit als diese, da seine Elemente als Objekte über den Heap verteilt gespeichert werden. Für das Einfügen eines neuen Elementes beträgt seine Komplexität lediglich O(log(n)). Daraus resultiert eine Gesamtkomplexität von O(k + log(n) + log(n)) = O(log(n)).

Bei dem Vergleich der drei Möglichkeiten überwiegt die Laufzeitklasse des binären Suchbaumes die der normalen und sortierten Liste und ist somit zu bevorzugen.

5.4 Laufzeitanalyse anhand eines Beispiels

Unter Zuhilfenahme eines Performance Profilers kann erfasst werden, welche Vorgänge wie viel (prozentuale) Rechenzeit in Anspruch nehmen. Dieser läuft im Hintergrund mit und sammelt die benötigten Daten zur Auswertung. Die folgenden Daten wurden mit dem Beispiel Jahr: 2019; Ziffer: 7; Teil: B ermittelt.

Das Suchen und Einfügen in den binären Suchbaum braucht jeweils ca 29 % der Rechenzeit, das Verwalten der Termliste ca 20 %. Die Überprüfung der Berechenbarkeit des Multiplikations- und Potenzoperators benötigt rund 2 %. Weitere 20 % werden von dem Framework im Hintergrund verbraucht.

6. Beispiele

6.1 Gegebene Beispieljahre

6.1.1 Jahr 2020

Term A	n	Zeit (s)	Term B	n	Zeit (s)
(((11111-1)*(1+1))/11)	10	0,008			0,029
(((((22*2)+2)*22)-2)*2)	8	0,000			0,009
((((333+3)*(3+3))+(3/3))+3)	9	0,008	((((3)!)!*3)-((((3)!)!+(((3)!)!/(3)!))/(3)!))	6	0,161
(((44+((4+4)/4))*44)-4)	8	0,001	((((4+4))!/((4)!-4))+4)	5	0,001
(((((55+(5*5))*5)+5)*5)-5)	8	0,006	(((((5)!-5)*((5)!-5))-(5)^(5))/5)	7	0,060
((((66*6)+((6+6)/6))+6)*(6-(6/6)))	10	0,049	(((((6)!*66)-(6)!)/6)/6)+(6)!)	7	0,081
((((((7*7)-7)*7)-7)*7)+(77/7))	9	0,017			2,129
(((((((8*8)*8)-8)*8)+8)*8)/(8+8))	9	0,014			1,988
(((999+(99/9))*(9+9))/9)	9	0,007			1,539

6.1.2 Jahr 2030

Term A	n	Zeit (s)	Term B	n	Zeit (s)
((((1111*(1+1))/11)+1)*(11-1))	12	0,013	(((((1+1))^(11)-(((1+1)+1))!)-11)-1)	10	0,064
((((((22*2)+2)*22)+2)*2)+2)	9	0,001	(((((22)^(2)+((2+2))!)*2)*2)-2)	8	0,013
(((333*(3+3))-(3/3))+33)	9	0,004	(((((3)!)!-3)*3)-((((3)!)!+(3)!)/(3)!))	6	0,197
(((((((4+4)*4)*4)*4)-4)*4)-((4+4)/4))	10	0,035	((((4)^(4)*(4+4))+((4)!/4))-(4)!)	7	0,205
(((((55+(5*5))*5)+5)*5)+5)	8	0,004	((((5)!*(((5)!/(5+5))+5))-5)-5)	7	0,035
(((66*6)+((66-6)/6))*(6-(6/6)))	10	0,043	((((((6)^(6)+(6)!)/6)/6)+(6)!)-6)	7	0,048
((((((7*7)-7)*7)+7)*7)-77)	8	0,007	(((((7)!/(7+7))-77)+7)*7)	7	0,028
(((((((8+8)*8)*8)-8)*(8+8))-8)-8)/8)	10	0,020	(((88*(8+8))+(((8)!/8)/8))-8)	8	0,438
(((9*9)-(99/9))*(((99/9)+9)+9))	10	0,073	(((((9+9)/9))^((99/9))-9)-9)	8	0,193

6.1.3 Jahr 2080

Term A	n	Zeit (s)	Term B	n	Zeit (s)
(((((111-11)*(1+1))-11)*11)+1)	12	0,011	((((1+1))^(11)+(11*((1+1)+1)))-1)	10	0,064
(((((22+2)+2)*(22-2))*2)*2)	9	0,002	((((((2+2))!+2)*(22-2))*2)*2)	8	0,022
(((33*(((3+3)*3)+3))*3)+(3/3))	9	0,008	((((3)!)!*3)-((((3)!)!/3)/3))	5	0,014
(((((4*4)*4)*4)+4)*(4+4))	7	0,000	(((4)^(4)+4)*(4+4))	5	0,002
((((5*5)*5)+5)*((55/5)+5))	8	0,002	((((((5)!+55)+(5)!)+(5)!)*5)+5)	7	0,037
((66-(6/6))*(((6*6)+((6+6)/6))-6))	10	0,055	(((((((6)!+((6)!/6))*(6)!)/6)/6)/6)-(6)!)	8	0,729
((((((7+7)*(7+7))-7)*77)+7)/7)	9	0,010			2,079
((((((8*8)*8)+8)*8)/(8+8))*8)	8	0,001			0,150
(((((99+99)-9)*99)+9)/9)	9	0,007	((((9*9)*9)*9)-((((9)!/9)+9)/9))	8	1,366

6.1.4 Jahr 2980

Term A	n	Zeit (s)	Term B	n	Zeit (s)
(((((111*((1+1)+1))-1)-1)*((11-1)-1))+1)	13	0,032	(((((((((1+1)+1))!+1))!-((1+1))^(11))-11)-1)	11	0,249
((((((22*22)*2)-222)*2)-2)*2)	11	0,015	(((((((2+2)+2))!+((2+2))!)*2)+2)*2)	8	0,012
(((((333-3)*3)+3)*3)+(3/3))	9	0,009	((((((3)!)!+33)*3)+(3/3))+((3)!)!)	7	2,074
((((((44*4)*4)+44)-4)*4)+4)	9	0,007	((((((4)!/4))!+(4)!)*4)+4)	5	0,004
(((((((5*5)*5)-5)*5)-5)*5)+5)	8	0,006	(((((5)!*5)-5)*5)+5)	5	0,004
(((((((66+6)*6)-6)*((6*6)+6))-6)-6)/6)	11	0,077	(((((((((6+6))!/(6)!)/6)+(6)!)/6)-(6)!)/6)	8	0,470
(((((((77-7)*7)+7)*((7*7)-7))-7)-7)/7)	11	0,091	(((((777-7)*(7+7))+(7)!)+(7)!)/7)	9	3,753
(((((((((8*8)*8)*8)*8)*8)+88)+8)/88)	11	0,147	((((((8)^(8)/8)+(8*8))/88)+8)/8)	9	2,746
(((((999*((9+9)+9))+9)/9)-9)-9)	10	0,022	((((((99/9))!*((9+9)+9))+(9)!)/(9)!)+9)	9	2,738

6.1.5 Auswertung

Es stellt sich heraus, dass die Beispiele laufzeittechnisch kein Problem darstellen. Über alle fünf Beispiele (inklusive 2019) hinweg liegt die durchschnittliche Rechenzeit für Teil A bei 28 Millisekunden mit einer Standardabweichung von 38,9 ms. Für Teil B beträgt sie 1011 Millisekunden mit einer Standardabweichung von 1791 ms. Es fällt auf, dass die Standardabweichung absolut als auch prozentual zunimmt. Dies liegt daran, dass je mehr Terme theoretisch erstellt werden können, desto unvorhersehbarer ist die Anzahl an tatsächlich erstellten Termen sowie die Laufzeit.

In keinem der Beispiele wird in Teil A ein Term aufgrund eines zu hohen Wertes verworfen. Somit sind alle ermittelten Lösungsterme (Teil A) allgemein und nicht nur innerhalb des Datentypes long eine optimale Lösung.

6.2 Sonderfall: Jahr 0

Wie bereits in dem Kapitel *Umsetzung* im Abschnitt *Sonderfall: Jahreszahl = 0* beschrieben stellt dieses Beispiel eine Ausnahme da. Das Programm gibt in diesem Fall die Subtraktion der eingegebenen Ziffer von sich selber zurück (Bsp: Ziffer: 9 Term: 9-9).

7. Erweiterungen

7.1 Modulo

Der Modulo-Operator kann wahlweise vom Benutzer hinzugefügt werden.

7.1.1 Bessere Ergebnisse für die Beispiele

Wenn die fünf Beispiele nun nochmals mit aktivierter Modulo Erweiterung durchgerechnet werden, fällt auf, dass in Teil A sich die Länge der resultierenden Terme nicht ändert. In Teil B wird hingegen in rund 50 % der Fälle ein durchschnittlich um 2 Ziffern kürzerer Term ausgegeben. Der Modulo-Operator benötigt für “brauchbare zusätzliche Werte” tendenziell Terme mit einem hohen Wert. Diese stellt Teil B früher bereit. Die neuen Terme befinden sich in der folgenden Tabelle.

Jahr	Ziffer	Zeit (s)	Term	n	Reduzierung um
2019	5	0,132	(((((5)!*(5)!)%(5)^(5))-(5/5))+(5)!)	7	1
	7	0,044	((7777-((((7)!-7)-7)/7))-(7)!)	7	2
	8	0,071	((8)!%(((((8+8))!/(8)!)%((8)!+8))/8))	7	2
2020	5	0,001	((((5)!*(5)!)%(5)^(5))+(5)!)	5	2
	7	0,104	(((7)^(7)-(((7+7)+7)/7))%(7)!)	7	2
	8	0,596	((((8)!*8)%(((8)!+(8*8))/(8+8)))+8)	8	1
	9	0,937	((9)!%((((99*9)*9)*9)+(9/9)))	8	1
2030	7	0,000	(((7)^(7)%(7)!)+7)	4	3
2080	5	0,021	(((5+5))^(5)%(((5)!*(5)!)/5))	6	1
	6	0,103	(((6)!+(((((6)!/6)/6))^(6)%(6)!))+(6)!)	7	1
	7	0,017	((((7+7))!%((7)!-7))+(7/7))	6	3
	8	0,012	((8)^(8)%((((8)!/8)-8)-8))	6	2
	9	0,007	((9)!%((9)!%((((9)!/9)+9)/9)))	6	2
2980	7	0,782	(((((7)^(7)+7)*7)%((7)!+(7/7)))-7)	8	1
	8	0,456	(((8)!%((8)^(8)%(((8)!*8)-(8/8))))-8)	8	1
	9	0,035	(((((9)^(9)*99)%(9)!)+9)/9)	7	2

7.1.2 Termwachstum

Diagramm 2 zeigt das Termwachstum mit aktivierter Erweiterung (Kapitel *Analyse des Termwachstums*).

Es fällt auf, dass es anfänglich sehr dem durchschnittlichen Wachstum ohne Erweiterung gleicht, es flacht später aber nicht ab und befindet sich knapp unterhalb des maximal möglichen Termwachstums mit Termbeschränkung und Termüberprüfung.

7.2.3 Bewertung

Es ist erstaunlich, dass trotz mehr möglicher Terme die durchschnittliche Laufzeit über die fünf Beispiele mit allen neun Ziffern hinweg als auch die Standardabweichung sich mehr als vierteln. Die Laufzeit beträgt nun 241 Millisekunden mit einer Standardabweichung von 420 ms. Der dazu verwendete Datensatz befindet sich im Anhang.

Da die Erweiterung laufzeitsparender als die normale Berechnung in Teil B operiert und oftmals kürzere Terme generiert, wird sie als sinnvoll erachtet. Sie kann kein schlechteres Ergebnis produzieren.

8. Literaturverzeichnis

Aufgabenblatt (o. J.): in: *BwInf*, [online] https://bwinf.de/fileadmin/bundeswettbewerb/38/aufgaben382.pdf [28.12.2019].

SortedDictionary Class (System.Collections.Generic) (o. J.): in: *Microsoft Docs*, [online] https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.sorteddictionary-2?view=netframework-4.8 [13.04.2020].

SortedList Class (System.Collections.Generic) (o. J.): in: *Microsoft Docs*, [online] https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.sortedlist-2?view=netframework-4.8 [13.04.2020].

9. Anhang

Die beiliegenden Dokumente befinden sich in folgender Reihenfolge:

1. Statistisch erhobenes Termwachstum Teil A
2. Statistisch erhobenes Termwachstum Teil B
3. Statistisch erhobenes Termwachstum Teil B mit Modulo-Erweiterung
4. Quellcode: TermClasses
5. Quellcode: CalculateTerm Methode
6. Quellcode: CreateTerms Methode
7. Quellcode: CheckTerm Methode

Statistisch erhobenes Termwachstum für Teil A ohne Modulo Erweiterung

Ziffer	Jahr	nDigit	Zeit (ms)	Summe	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
1	4567890	20	75377	15426230	1	1	4	8	20	46	108	259	630	1565	3898	9816	25054	64589	167482	438275	1152904	3055931	8140439	2365200
2	4567890	18	118258	19568840	1	2	7	21	50	134	354	968	2615	7265	20562	58526	168311	488198	1427837	4209798	12480819	703372		
3	4567890	17	135501	30220720	1	3	9	27	70	189	559	1605	4683	13850	41512	126149	386389	215973	3723910	11688273	13038395			
4	4567890	16	135198	28922080	1	3	11	30	87	257	751	2247	6794	20940	65279	206397	658221	2081682	6868492	18974015				
5	4567891	16	90592	21247795	1	3	11	31	91	254	762	2348	7231	22599	71892	230846	750472	2460014	8130884	9570356				
6	4567890	16	140071	31497062	1	3	11	35	104	307	929	2889	9260	29938	97921	323856	1084017	3659560	12461568	13826663				
7	4567890	16	235547	58651850	1	3	12	40	112	357	1109	3614	11780	39347	131952	376014	1547741	5373292	18826802	32265831				
8	4567890	16	230907	52310645	1	3	12	39	118	355	1145	3762	12420	42135	143298	497611	1736821	2065818	21834011	21901762				
9	4567890	16	548231	90993909	1	3	11	36	111	334	1091	3498	11766	39416	135760	470105	1649295	4837626	20871344	61968215				

*unvollständig

AVG	1	2,7	9,8	30	85	248	756	2354	7464	24117	79119	255480	889591	2360750	10479148	*durschschnittliches Termwachstum mit Termüberprüfung*	
STDEV	0	0,71	2,7	10,1	32,7	107	360	1227	4233	14726	51499	174205	648043	1950988	8388870		
STD%AVG	1	0,73	0,73	0,66	0,61	0,57	0,52	0,48	0,43	0,39	0,35	0,32	0,27	0,17	0,20		

Min	1	1	4	8	20	46	108	259	630	1565	3898	9816	25054	64589	167482	*minimales Termwachstum mit Termüberprüfung*	
Max	1	3	12	40	118	357	1145	3762	12420	42135	143298	497611	1736821	5373292	21834011	*maximales Termwachstum mit Termüberprüfung*	

Statistisch erhobenes Termwachstum für Teil B ohne Modulo Erweiterung

Ziffer	nDigit	Zeit (ms)	Summe	1	2	3	4	5	6	7	8	9	
1	9	32	1749323	1	1	6	26	109	413	1554	5841	22282	
2	9	453	38354903	1	2	15	85	436	2069	9952	49335	246145	
3	9	76041	20023577	1	22	216	2054	19469	187840	1831358	17982604	177488302	angenähert
4	9	82216	23155516	1	12	102	805	5889	44775	340993	2613437	20149502	
5	9	37354	11284247	1	11	79	542	3738	25995	184549	1332635	9736697	
6	9	29193	9172121	1	11	78	528	3500	23508	161209	1119662	7863624	
7	9	23006	7250580	1	10	68	437	2783	18487	126573	882464	6219757	
8	9	19062	6386702	1	10	69	437	2761	17979	119568	803991	5441886	
9	9	13898	4795114	1	10	67	411	2498	15528	98528	630181	4047890	

	1	2	3	4	5	6	7	8	9	
AVG	1	9,9	78	592	4576	37399	319365	2824461	25690676	*durschschnittliches Termwachstum mit Termüberprüfung*
STDEV	0	6	60	597	5845	57922	575801	5736508	57237868	
STD%AVG	1	0,39	0,22	-0,01	-0,28	-0,55	-0,80	-1,03	-1,23	
Min	1	1	6	26	109	413	1554	5841	22282	*minimales Termwachstum mit Termüberprüfung*
Max	1	22	216	2054	19469	187840	1831358	17982604	177488302	*maximales Termwachstum mit Termüberprüfung*

Statistisch erhobenes Termzunahme für die Modulo Erweiterung

Ziffer	nDigit	Zeit (ms)	Summe	1	2	3	4	5	6	7	8	9	
1	9	35		0	0	0	0	1	6	38	237	1240	
2	9	509		0	0	0	0	0	10	116	1267	10818	
3	9	94582		0	0	1	16	321	6735	113872	1671854	22986970	angenähert
4	9	109928		0	0	0	4	115	2030	27021	309453	3341676	
5	9	47335		0	0	0	3	51	814	10528	116916	1169550	
6	9	39182		0	0	1	12	132	1351	14426	142040	1321998	
7	9	34867		0	0	2	31	324	2835	23487	198653	1668446	
8	9	31912		0	0	1	17	237	2499	22801	196474	1675332	
9	9	28414		0	0	2	31	378	3967	38180	326986	2655977	

AVG	#	###	1	16	195	2250	27830	329320	3870223	*durschschnittliche Zunahme*
STDEV	0	0	1	12	146	2134	34595	516433	7250175	
STD%AVG	#	###	0,40	0,25	0,25	0,05	-0,24	-0,57	-0,87	
Min	0	0	0	0	0	6	38	237	1240	*minimale Zunahme*
Max	0	0	2	31	378	6735	113872	1671854	22986970	*maximale Zunahme*

AVG	1	9,9	79	604	4749	39649	347195	3153781	29560899	*durschnittliches Termwachstum*
Min	1	1	6	26	109	419	1592	6078	23522	
Max	1	22	218	2085	19847	194575	1945230	19654458	200475272	

```
 1  using System;
 2  using System.Collections.Generic;
 3  using System.Linq;
 4  using System.Text;
 5  using System.Threading.Tasks;
 6
 7  namespace BwInf38Runde2Aufgabe2
 8  {
 9      public abstract class Term
10      {
11          private static double LogLong = Math.Log10(long.MaxValue);
12          private static double LogLogLong = Math.Log10(Math.Log10
              (long.MaxValue));
13
14          public virtual long GetResult()
15          {
16              return 0;
17          }
18          public virtual string PrintTerm()
19          {
20              return string.Empty;
21          }
22          public static double GetLogLong()
23          {
24              return LogLong;
25          }
26          public static double GetLogLogLong()
27          {
28              return LogLogLong;
29          }
30      }
31      public class Literal : Term
32      {
33          long Value;
34          public Literal(long _Value) : base()
35          {
36              Value = _Value;
37          }
38
39          public override long GetResult()
40          {
41              return Value;
42          }
43          public override string PrintTerm()
44          {
45              return Value.ToString();
46          }
47      }
48      public class FactorialOperator : Term
49      {
50          protected long result;
51          protected Term Term1;
52          public FactorialOperator(Term PriorTerm) : base()
```

```
53          {
54              Term1 = PriorTerm;
55              result = 1;
56              long Number = PriorTerm.GetResult();
57              while (Number != 1)
58              {
59                  result *= Number;
60                  Number--;
61              }
62          }
63
64          public override long GetResult()
65          {
66              return result;
67          }
68          public override string PrintTerm()
69          {
70              return "(" + Term1.PrintTerm() + ")!";
71          }
72          public static bool IsCalculatable(Term Term1)
73          {
74              if (Term1.GetResult() <= 20)
75              {
76                  return true;
77              }
78              else
79              {
80                  return false;
81              }
82          }
83      }
84
85      public class AddOperator : Term
86      {
87          long result;
88          protected Term Term1;
89          protected Term Term2;
90          public AddOperator(Term _Term1, Term _Term2)
91          {
92              Term1 = _Term1;
93              Term2 = _Term2;
94              result = Term1.GetResult() + Term2.GetResult();
95          }
96
97          public override long GetResult()
98          {
99              return result;
100         }
101         public override string PrintTerm()
102         {
103             return "(" + Term1.PrintTerm() + "+" + Term2.PrintTerm() + ")";
104         }
105         public static bool IsCalculatable(Term Term1, Term Term2)
```

```
106         {
107             long Check = long.MaxValue;
108             Check -= Term1.GetResult() + Term2.GetResult();
109             if (Check > 0)
110             {
111                 return true;
112             }
113             else
114             {
115                 return false;
116             }
117         }
118     }
119
120     public class SubtractOperator : Term
121     {
122         long result;
123         protected Term Term1;
124         protected Term Term2;
125         public SubtractOperator(Term _Term1, Term _Term2)
126         {
127             Term1 = _Term1;
128             Term2 = _Term2;
129             result = Term1.GetResult() - Term2.GetResult();
130         }
131
132         public override long GetResult()
133         {
134             return result;
135         }
136         public override string PrintTerm()
137         {
138             return "(" + Term1.PrintTerm() + "-" + Term2.PrintTerm() + ")";
139         }
140     }
141
142     public class MultiplyOperator : Term
143     {
144         long result;
145         protected Term Term1;
146         protected Term Term2;
147         public MultiplyOperator(Term _Term1, Term _Term2)
148         {
149             Term1 = _Term1;
150             Term2 = _Term2;
151             result = Term1.GetResult() * Term2.GetResult();
152         }
153
154         public override long GetResult()
155         {
156             return result;
157         }
158         public override string PrintTerm()
```

```
159         {
160             return "(" + Term1.PrintTerm() + "*" + Term2.PrintTerm() + ")";
161         }
162         public static bool IsCalculatable(Term Term1, Term Term2)
163         {
164             if (GetLogLong() - 0.00000001 > Math.Log10(Term1.GetResult()) +
                  Math.Log10(Term2.GetResult()))
165             {
166                 return true;
167             }
168             else
169             {
170                 return false;
171             }
172         }
173     }
174
175     public class DivisionOperator : Term
176     {
177         long result;
178         protected Term Term1;
179         protected Term Term2;
180         public DivisionOperator(Term _Term1, Term _Term2)
181         {
182             Term1 = _Term1;
183             Term2 = _Term2;
184             result = Term1.GetResult() / Term2.GetResult();
185         }
186
187         public override long GetResult()
188         {
189             return result;
190         }
191         public override string PrintTerm()
192         {
193             return "(" + Term1.PrintTerm() + "/" + Term2.PrintTerm() + ")";
194         }
195         public static bool IsCalculatable(Term Term1, Term Term2)
196         {
197             int Remainder = (int)(Term1.GetResult() % Term2.GetResult());
198             if (Remainder == 0)
199             {
200                 return true;
201             }
202             else
203             {
204                 return false;
205             }
206         }
207     }
208
209     public class ModuloOperator : Term
210     {
```

```
211         long result;
212         protected Term Term1;
213         protected Term Term2;
214         public ModuloOperator(Term _Term1, Term _Term2)
215         {
216             Term1 = _Term1;
217             Term2 = _Term2;
218             result = Term1.GetResult() % Term2.GetResult();
219         }
220 
221         public override long GetResult()
222         {
223             return result;
224         }
225         public override string PrintTerm()
226         {
227             return "(" + Term1.PrintTerm() + "%" + Term2.PrintTerm() + ")";
228         }
229 
230     }
231 
232     public class PowerOperator : Term
233     {
234         long result;
235         protected Term Term1;
236         protected Term Term2;
237         public PowerOperator(Term _Term1, Term _Term2)
238         {
239             Term1 = _Term1;
240             Term2 = _Term2;
241             result = (long)Math.Pow(Term1.GetResult(), Term2.GetResult());
242         }
243 
244         public override long GetResult()
245         {
246             return result;
247         }
248         public override string PrintTerm()
249         {
250             return "(" + Term1.PrintTerm() + ")^(" + Term2.PrintTerm() +
                  ")";
251         }
252 
253         public static bool IsCalculatable(Term Term1, Term Term2)
254         {
255             if (GetLogLogLong() - 0.0000001 > Math.Log10(Term2.GetResult())
                  + Math.Log10(Math.Log10(Term1.GetResult())))
256             {
257                 return true;
258             }
259             else
260             {
261                 return false;
```

```
262                 }
263             }
264         }
265 }
266
```

```
 1  using System.Collections.Generic;
 2  using System.Windows;
 3
 4  namespace BwInf38Runde2Aufgabe2
 5  {
 6      public partial class MainWindow : Window
 7      {
 8          private void CalculateTerm()
 9          {
10              //Erstelle den ersten Digit
11              Literal FirstLiteral = new Literal(Digit);
12
13              //Lege eine Liste an Index 0 an und speicher in ihr First
                  Literal
14              ListTerms.Add(new List<Term>(100000));
15              ListTerms[0].Add(FirstLiteral);
16              DictionaryResult.Add(FirstLiteral.GetResult(), FirstLiteral);
17
18              //Schaue, ob Literal GoalNumber ist
19              if (FirstLiteral.GetResult() == GoalNumber)
20              {
21                  LabelResult1Term.Content = FirstLiteral.GetResult().ToString
                      ();
22                  NeededNumberOfDigits1 = 1;
23                  return;
24              }
25
26              //Erstelle für jede Ziffernlänge (alle) Terme
27              for (nDigit = 1; true; nDigit++)
28              {
29                  //Lege Liste für nDigit an
30                  ListTerms.Add(new List<Term>());
31
32                  //Erstelle Literal für nDigit
33                  long LiteralValue = ListTerms[nDigit - 1][0].GetResult();
34                  LiteralValue = LiteralValue * 10 + Digit;
35                  Literal NewLiteral = new Literal(LiteralValue);
36                  ListTerms[nDigit].Add(NewLiteral);
37
38                  //Muss theoretisch noch überprüft werden, ob wert nicht
                      schon erreicht
39                  if (LiteralValue == GoalNumber)
40                  {
41                      GoalNumber1Reached = true;
42                      LabelResult1Term.Content = LiteralValue.ToString();
43                  }
44
45                  if (Task == 2)
46                  {
47                      //Wende Fakultät für nDigit-1 an
48                      Term OldTerm;
49                      int Lenght = ListTerms[nDigit - 1].Count;
50
```

```
51                      for (int i = 0; i < Lenght; i++)
52                      {
53                          OldTerm = ListTerms[nDigit - 1][i];
54                          while (FactorialOperator.IsCalculatable(OldTerm))
55                          {
56                              Term NewTerm = new FactorialOperator(OldTerm);
57
58                              if (CheckTerm(NewTerm))
59                              {
60                                  ListTerms[nDigit - 1].Add(NewTerm);
61                                  DictionaryResult.Add(NewTerm.GetResult(),
                        NewTerm);
62                              }
63                              else if (NewTerm.GetResult() ==
                        OldTerm.GetResult())
64                              {
65                                  break;
66                              }
67                              OldTerm = NewTerm;
68                          }
69                      }
70                  }
71                  if (Task == 2 && NeededNumberOfDigits1 - 1 <= nDigit &&
                      BoolAB)
72                  {
73                      LabelResult2Term.Content = "Keine kürzere Lösung
                        gefunden";
74                      return;
75                  }
76
77                  //Gehe Ziffernlänge bis zur Hälfte der aktuellen hoch
78                  for (int DigitLenght = 0; DigitLenght < (nDigit + 1) / 2;
                      DigitLenght++)
79                  {
80                      //Für jede dieser Ziffernlänge gehe alle ihre Terme
                        durch
81                      int UpperBound = ListTerms[DigitLenght].Count;
82                      for (int ElementsOfDigitLength = 0;
                        ElementsOfDigitLength < UpperBound; ElementsOfDigitLength
                        ++)
83                      {
84                          //Für jede dieser Terme verknüpfe sie mit mit allen
                        nDigit-DigitLenght Termen
85                          int RemainingDigitDifference = nDigit - DigitLenght
                        - 1;
86                          for (int ElementsOfRemainingDigitDifference = 0;
                        ElementsOfRemainingDigitDifference < ListTerms
                        [RemainingDigitDifference].Count;
                        ElementsOfRemainingDigitDifference++)
87                          {
88                              //Erstelle alle sinnvollen Terme aus den zwei
                        aktuellen Termen
89                              CreateTerms(DigitLenght, ElementsOfDigitLength,
```

```
                        RemainingDigitDifference,
                        ElementsOfRemainingDigitDifference);

                            //Breche ab, wenn GoalNumber erreicht

                            if (Task == 2 && GoalNumber2Reached)
                            {
                                return;
                            }
                            else if (Task == 1 && GoalNumber1Reached)
                            {
                                NeededNumberOfDigits1 = nDigit + 1;
                                return;
                            }
                        }
                    }
                }
            }
        }
    }
}

```

```
1  using System.Collections.Generic;
2  using System.Windows;
3
4  namespace BwInf38Runde2Aufgabe2
5  {
6      public partial class MainWindow : Window
7      {
8          private void CreateTerms(int IndexA1, int IndexA2, int IndexB1, int ↪
             IndexB2)
9          {
10             Term NewTerm;
11             Term Term1 = ListTerms[IndexA1][IndexA2];
12             Term Term2 = ListTerms[IndexB1][IndexB2];
13
14             //Schaue ob Term2 größer ist als Term1, wenn ja tausche diese
15             if (Term1.GetResult() < Term2.GetResult())
16             {
17                 Term1 = ListTerms[IndexB1][IndexB2];
18                 Term2 = ListTerms[IndexA1][IndexA2];
19             }
20
21             //Modulo
22             if (BoolModulo)
23             {
24                 NewTerm = new ModuloOperator(Term1, Term2);
25                 if (CheckTerm(NewTerm))
26                 {
27                     ListTerms[nDigit].Add(NewTerm);
28                     DictionaryResult.Add(NewTerm.GetResult(), NewTerm);
29                 }
30             }
31
32             //Addition
33             if (AddOperator.IsCalculatable(Term1, Term2))
34             {
35                 NewTerm = new AddOperator(Term1, Term2);
36                 if (CheckTerm(NewTerm))
37                 {
38                     ListTerms[nDigit].Add(NewTerm);
39                     DictionaryResult.Add(NewTerm.GetResult(), NewTerm);
40                 }
41             }
42
43             //Subtraction
44             NewTerm = new SubtractOperator(Term1, Term2);
45             if (CheckTerm(NewTerm))
46             {
47                 ListTerms[nDigit].Add(NewTerm);
48                 DictionaryResult.Add(NewTerm.GetResult(), NewTerm);
49             }
50
51             //Multiplication
52             if (MultiplyOperator.IsCalculatable(Term1, Term2))
```

```
53              {
54                  NewTerm = new MultiplyOperator(Term1, Term2);
55                  if (CheckTerm(NewTerm))
56                  {
57                      ListTerms[nDigit].Add(NewTerm);
58                      DictionaryResult.Add(NewTerm.GetResult(), NewTerm);
59                  }
60              }
61
62              //Division
63              if (DivisionOperator.IsCalculatable(Term1, Term2))
64              {
65                  NewTerm = new DivisionOperator(Term1, Term2);
66                  if (CheckTerm(NewTerm))
67                  {
68                      ListTerms[nDigit].Add(NewTerm);
69                      DictionaryResult.Add(NewTerm.GetResult(), NewTerm);
70                  }
71              }
72
73              //Power - Wird nur ausgeführt, wenn GoalNumber1 schon erreicht ↵
                  wurde
74              if (Task == 2)
75              {
76                  if (PowerOperator.IsCalculatable(Term1, Term2))
77                  {
78                      NewTerm = new PowerOperator(Term1, Term2);
79                      if (CheckTerm(NewTerm))
80                      {
81                          ListTerms[nDigit].Add(NewTerm);
82                          DictionaryResult.Add(NewTerm.GetResult(), NewTerm);
83                      }
84                  }
85
86                  if (PowerOperator.IsCalculatable(Term2, Term1))
87                  {
88                      NewTerm = new PowerOperator(Term2, Term1);
89                      if (CheckTerm(NewTerm))
90                      {
91                          ListTerms[nDigit].Add(NewTerm);
92                          DictionaryResult.Add(NewTerm.GetResult(), NewTerm);
93                      }
94                  }
95              }
96          }
97
98      }
99  }
100
```

```
1  using System.Collections.Generic;
2  using System.Windows;
3
4  namespace BwInf38Runde2Aufgabe2
5  {
6      public partial class MainWindow : Window
7      {
8          private bool CheckTerm(Term NewTerm)
9          {
10             long TermResult = NewTerm.GetResult();
11
12             if (DictionaryResult.ContainsKey(TermResult))
13             {
14                 NumberOfDoubleTerms++;
15                 return false;
16             }
17             else if (TermResult <= 0)
18             {
19                 NumberOfNotNaturalTerms++;
20                 return false;
21             }
22             else if (Task == 2 && TermResult == GoalNumber)
23             {
24                 GoalNumber2Reached = true;
25                 LabelResult2Term.Content = NewTerm.PrintTerm();
26                 return false;
27             }
28             else if (Task == 1 && TermResult == GoalNumber)
29             {
30                 GoalNumber1Reached = true;
31                 LabelResult1Term.Content = NewTerm.PrintTerm();
32                 return false;
33             }
34             return true;
35         }
36     }
37 }
38
```