Aufgabe 3: Hex-Max

Teilnahme-ID: 62344

Manuel Frohn

17.04.2022

Inhalt

Lösungsansatz

Das Problem: Herauszufinden, wie die Streichhölzer umzulegen sind, um die möglichst größte Zahl zu formen, lässt sich in zwei Teile teilen. Zum einen in den Teil, herauszufinden, was die höchstmöglichste Zahl ist, die sich aus der Ausgangszahl und der Anzahl an Umlegungen bilden lässt, und zum anderen in den Teil herauszufinden, wie die Streichhölzer umgelegt werden müssen um diese Zahl zu bilden. Der erste Teil ist der, der wirklich komplexe Berechnungen benötigt. Der zweite Teil ist, insofern man den ersten Teil bereits berechnet hat, nur ein einfacher Abgleich der Ausgangszahl mit der verbesserten Zahl. Da der zweite Teil, bei guter Modellierung, nur einen sehr kleinen Teil der Berechnung ausmacht, wird dieser, in diesem Abschnitt, nicht weiter betrachtet.

Dieser Abschnitt beschäftigt sich im Folgenden daher ausschließlich mit der theoretischen Lösung des ersten Teils. Um dieses Problem zu lösen, macht sich der Algorithmus eine, der fundamentale Eigenschaft, einer Zahl zu Nutze: Das eine Verbesserung einer Ziffer höherer Ordnung, beispielsweise bei einer Drei-Ziffer Zahl die Hunderterstelle (für das Beispiel wird hier, der Einfachheit halber, das Dezimalsystem verwendet), die Zahl stärker verbessert als es eine Veränderung der Zehner- und Einerstelle jemals könnten. Das heißt, um bei dem Drei-Ziffern Beispiel zu bleiben, dass eine Verbesserung der Hunderterstelle, auch nur um eins, es rechtfertig sowohl die Zehner-, als auch die Einerstelle auf null zu verändern. Durch diese Eigenschaft lässt sich nun grundlegen ein Algorithmus formulieren, bei dem von der höchsten Ziffer zur niedrigsten Ziffer, durch die Zahl durch, iteriert wird und Schritt für Schritt jede Ziffer verbessert wird. Solch ein Algorithmus kann aber nur funktionieren, wenn man eine Möglichkeit hat, festzustellen, ob die Veränderung der Ziffer möglich ist. Hierzu brauchen

wir zunächst ein neues Konzept: Einen Zwischenspeicher. Dadurch ist es möglich, eine Ziffer zu verändern, ohne Streichhölzer direkt woanders hinzulegen bzw. wegzunehmen. Verändert man nun eine „8“, zu einem „F“, so legen man die drei Streichhölzer, die man weggenommen hat, nicht direkt wieder in die Zahl, sondern zunächst in den besagten Zwischenspeicher. Kann man jetzt noch feststellen, ob man diesen Zwischenspeicher, nach dessen Veränderung noch abbauen kann. Also ob man die Streichhölzer, die in diesem Zwischenspeicher liegen, oder, die man dem Zwischenspeicher „schuldet“, in den nachfolgenden Ziffern wieder unterbringen, bzw. wegnehmen kann, so ließe sich das Problem mit dem benannten iterativen Ansatz lösen, indem man wie beschrieben von der höchsten, zur niedrigsten Ziffer iteriert und für jede Zahl jede Verbesserungsmöglichkeit daraufhin prüft, ob die notwendige Veränderung am Zwischenspeicher möglich ist, und ob noch genügend Umlegungen übrig sind um jene Verbesserung durchzuführen. Nachdem dies für alle Ziffern durchgeführt wurde, muss man, wenn der Zwischenspeicher nicht null ist, diesen Abbauen. Dies kann man, indem man, für den unveränderten Teil der Zahl alle Veränderungsmöglichkeit durchprobiert. Dieses Durchprobieren lässt sich noch optimieren, indem man nur alle relevanten Möglichkeiten prüft. Dies ist möglich, indem man die Veränderungsmöglichkeiten der Zahl, Ziffer für Ziffer, berechnet, und nur jene Möglichkeiten speichert, die noch nicht gespeichert sind und die, die nicht schlechter, als die Gespeicherten, sind. Man erstellt also einen Baum der Abbaumöglichkeiten.

Dieser letzte Abschnitt beschäftigt sich nun damit, wie festgestellt wird, ob der Zwischenspeicher abgebaut werden kann. Hierzu verwendet der Algorithmus sogenannte „Potenziale“. Ein Potential gibt einen Wert an, der, hypothetisch (solange genügend Umlegungen übrig sind) abbaubar wäre. Solche „Potenziale“ lassen sich für jede Ziffer bestimmen. Die Potenziale von „0“, zum Beispiel, sind 1:0, -1:1, -2:0, - 3:0 und -4:0 (die erste Zahl ist der Wert, der hypothetisch abbaubar wäre und der zweite, wie viele, zifferninterne, Umlegungen dazu nötig wären). Das allein reicht aber noch nicht um, festzustellen, ob eine Zwischenablage abgebaut werden kann oder nicht. Hat man zum Beispiel den Zwischenablagewert „2“ und möchte diesen mit der Ziffernfolge „00“ abbauen, so sind beide Ziffern von Nöten, um dies zu tuen. Hier kommt jetzt das Addieren von Potenzialen ins Spiel. Um die Potentiale nun addieren zu können müssen sie zunächst anders dargestellt werden, um die internen Umlegungen besser berechnen zu können. Ein Potenzial speichert nun, die Anzahl an streichholzleeren Stellen die, nach der Veränderung durch Streichhölzer belegt sind und die Anzahl der Stellen mit Streichhölzern, die nach der, vom Potenzial beschriebenen, Veränderung leer sind. Die Potenziale der Ziffer „0“ sind dann 0:1, 2:1, 2:0, 3:0 und 4:0 (der erste Wert gibt die Anzahl der geleerten Stellen an und der zweite Wert die, der Gefüllten). Die Werte die für die Prüfung der Abbaubarkeit der Zwischenablage relevant sind lassen sich nun wie folgt berechnen: Die Anzahl der internen Umlegungen ist immer der kleiner der beiden Werte. Ist das Potential also 2:1, so ist die Anzahl der internen Umlegungen 1. Ist das Potential 3:0, so ist die Anzahl der internen Umlegungen 0. Dies lässt sich dadurch erklären, dass zum Beispiel bei dem Potential 2:1, eines der Weggenommenen Streichhölzer innerhalb der Ziffer direkt wieder hingelegt werden würde, wodurch hier eine interne Umlegung auftreten würde. Das eigentliche „Potential“, also der Wert, der hypothetisch abgebaut werden, lässt sich generell für ein Potential $n:p$ durch folgende Formel berechnen: $P= p – n$.

P ist dabei der tatsächliche Potentialwert, *p* ist die Anzahl der gefüllten Stellen und *n* die Anzahl der gelehrten Stellen. Mit diesem Format ist nun die Addition von Potentialen sehr einfach. Hat man nun die Potentiale *n1:p1* und *n2:p2* so berechnet sich die Summe der beiden, wie folgt *n1 + n2:p1 + p2*. Um nun das Potential einer Ziffernfolge zu berechnen, kann man einen rekursiven Algorithmus formulieren. Dieser berechnet die Potenziale einer Ziffernfolge, indem er für alle Potenziale der ersten Ziffer der Folge, die Summe mit allen Potentialen der Ziffernfolge, der Zeichenkette ohne diese erste Ziffer bildet, (hier muss man sich eine verschachtelte foreach-Schleife vorstellen) und prüft, ob diese neugebildeten Potentiale relevant sind, also ob ihr Wert noch nicht in der Liste der Potentiale vorhanden ist, oder wenn er vorhanden ist, ob das neue Potential weniger interne Umlegungen benötigt, als das bereits bekannte. Ist ein neugebildetes Potential relevant, so wird es der Liste hinzugefügt, bzw. überschreibt das Potential das denselben Wert hat. Für das „00"- Beispiel heißt das, dass man zuerst die Potentiale von „0" bestimmen muss.

Um nun festzustellen, ob eine Zwischenspeicher abgebaut werden kann, berechnet man nun die Potentiale der Ziffernkette, die auf die zu verbessernde Ziffer folgt, und prüft ob der Zwischenspeicherwert, in den Potentialen enthalten ist und ob die übrige Anzahl an möglichen Umlegungen ausreicht, um dieses Potential zu nutzen. Trift beides zu, so kann der Zwischenspeicher abgebaut werden. Wen nicht, dann nicht.

Komplexitätsanalyse

Die Zeitkomplexität des beschriebenen Algorithmus lässt sich nun in zwei Teile teilen, die der Verbesserungsphase und die, der Abbauphase. Betrachten wir zuerst die Verbesserungsphase. Hier beschäftigen wir uns zuerst mit der Berechnung der Potentiale. Hier stellt man fest, dass der oben beschriebene Algorithmus zur Ermittlung der Potentiale einer Ziffernfolge, auch immer die Potentiale aller kleineren Ziffernfolgen bestimmen muss. Deswegen bietet es sich an, zu Beginn der Verbesserungsphase die Potentiale für die gesamte Zahl zu bestimmen und die jeweiligen Zwischenergebnisse, also die Potentiale für die kleineren Ziffernfolgen zu speichern. Dadurch muss man diesen Algorithmus, zur Potentialbestimmung, nur einmal anwenden. Die Komplexität von diesem, lässt sich, durch den Ausdruck $O(\sum_{a=1}^{n} a)$, beschreiben. *n* ist dabei die Anzahl der Ziffern in der Zahl. Dieser Ausdruck stimmt, da die Anzahl der Potentiale, um die die Liste der Potentiale, pro Ziffer, erweitert wird, konstant ist. Sie ist immer Fünf, da eine Ziffer immer fünf Potentiale hat. Diese Potentiale haben maximal den Wert „5" und minimal den Wert „-5". Der oben beschriebene Term lässt sich wie folgt umformen $O(\sum_{a=1}^{n} a) = O(n^2 + n)$. Nachdem die Potenziale berechnet worden sind, iteriert der Algorithmus nun über die Zahl und probiert, je Ziffer, die Verbesserungsmöglichkeiten aus. Unter Verwendung der richtige Datenstruktur, zur Speicherung der Potentiale, lässt sich nun in *O(1)* feststellen ob solch eine Veränderung möglich ist. Da die Anzahl der maximalen Verbesserungsmöglichkeiten Konstant ist, beträgt die Komplexität, dieser Iteration, *O(n)*. Die gesamte Komplexität der Verbesserungsphase beträgt also $O(n^2 + n) + O(n)$. Die Laufzeitkomplexität der Verbesserungsphase ist also Polynomial.

Kommen wir nun zur Abbauphase. Da wir hier, im schlimmsten Fall alle möglichen Ziffernkombinationen, des unveränderten Teils der Zahl, durchprobieren müssen, beträgt die

Zeitkomplexität dieser Phase $O(15^k)$. k ist dabei die Länge des unveränderten Teils der Ziffer, dieser beginnt nach der letzten Verbesserten Ziffer und endet bei der letzten Ziffer der Zahl.

Die Zeitkomplexität, des gesamten Algorithmus, beträgt also $O(n^2 + n) + O(n) + O(15^k)$.

Verbesserung des Lösungsansatzes

Der oben beschrieben Algorithmus mag zwar für Zahlen mit geringer Ziffernmenge effizient genug sein, aber die exponentiell steigende Laufzeit, der Abbauphase, ist für Zahlen, mit großer Ziffernmenge, problematisch. Glücklicherweise befinden sich die Grundsteine, für eine Algorithmus, für die Abbauphase, von polynomialer Laufzeit, bereits im vorher beschrieben Algorithmus.

Der neue Abbaualgorithmus funktioniert wie folgt: Man nehme die erste Ziffer, bei der, die Ziffernfolge, die von der darauffolgenden Ziffer, bis zur letzten Ziffer läuft, nicht ausreicht, um den Zwischenspeicher abzubauen. Man geht dann alle Veränderungen dieser Ziffer, beginnend bei der höchsten, durch und prüft, ob solch eine Veränderung möglich. Ist sie möglich, sind also noch ausreichend Umlegungen vorhanden, muss man prüfen, ob der, durch diese Veränderung entstehende, Wert des Zwischenspeichers, in der auf die ausgewählte Ziffer folgenden, Ziffernkette, abbaubar ist. Ist dem so wird die Veränderung durchgeführt, und dieselbe Prozedur wird nun mit der Ziffer gemacht, die auf die gerade betrachtete Ziffer folgt. Und da das Potential einer leeren Ziffernfolge 0:0 ist, funktioniert dieser Algorithmus auch für die letzte Ziffer, sollte der Algorithmus zu dieser kommen.

Durch diesen Algorithmus ist die Laufzeitkomplexität der Abbauphase nun nicht mehr $O(15^k)$, sondern $O(k)$. Dadurch ist die gesamte Laufzeitkomplexität, des Algorithmus, nun $O(n^2 + n) + O(n) + O(k)$.

Umsetzung

Den, im letzten Abschnitt beschriebenen, Algorithmus habe ich in Java umgesetzt. Die Implementation besteht aus drei Klassen und einer Enumeration. Der „HexMaxApplication“-Klasse, der „Digit“-Enumeration, der „Potential“-Klasse und der „Number“-Klasse. Die „HexMaxApplication“-Klasse enthält nur die GUI, dient also nur zur Darstellung der Ergebnisse und wird deshalb, in diesem Text, nicht weiter erläutert. Die „Potential“-Klasse (Codeausschnitt 1) repräsentiert das Oben beschriebene Konzept. Ein Objekt dieser Klasse ist also ein Potenzial. Es enthält die Anzahl der gefüllten Stellen(„additions“) und der geleerten Stellen(„subtractions“). Desweiter enthält es die Anzahl an internen Umlegungen(„cost“) und den Wert des Potenzials(„value“). Diese Variablen werden, dem Objekt bei der Initialisierung übergeben. Die Addition von zwei Potentialen läuft über eine Überladung, des Konstruktors. Dieser zweite Konstruktor nimmt zwei „Potential“-Objekte als Parameter. Die „additions“, des neuen Potentials, werden bestimmt, indem die „additions“, der beiden Parameterobjekte, summiert werden. Für die „subtractions“ wird dasselbe gemacht. Danach werden der Wert des

Potentials und seine Kosten (die Anzahl der internen Umlegungen wird, von hier aus, als „Kosten“ bezeichnet), wie im letzten Abschnitt beschrieben, berechnet.

Als nächstes werden wir uns mit der „Digit“-Enumeration. Ein Objekt dieser Enumeration repräsentiert eine, aus Streichhölzern gelegt, Ziffer, im Hexadezimalsystem. Ein Objekt, dieser Enumeration, enthält drei finale Variablen. Die erste ist die „code“-Zeichenkette, welche die Ziffer als belegte und unbelegte Streichholzstellen kodiert. Eine null repräsentiert eine Stelle, an der kein Streichholz liegt, eine eins, eine Stelle, an der ein Streichholz liegt. Welche Stelle gemeint ist hängt vom Index ab. Der Index des Zeichens korrespondiert mit der Stelle in Abbildung 1.

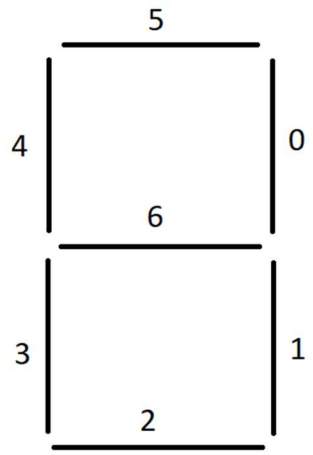


Abbildung 1: Kodierung der Ziffern

Die Zweite Variable, das „character“-Zeichen, ist das Zeichen, mit dem der Wert des Objekts, im Hexadezimalsystem beschreiben wird. Die dritte Variable, die „value“-Zahl, ist der Wert der Ziffer im Dezimalsystem. Jedes Objekt hat des Weiteren eine „getBetterDigits“-Methode, welche ein Array zurückgibt, welches alle Ziffern, mit einem höheren Wert, enthält. Die Klasse, an sich, hat zwei statische Methoden. Die erste, ist die „getChangeCost“-Methode (Codeausschnitt 2). Diese Methode berechnet die „Kosten“ einer Veränderung einer Ziffer, zu einer Anderen. Diese „Kosten“ werden als Array zurückgegeben. Das erste Element ist die Anzahl der „additions“, das Zweite die Anzahl der „subtractions“, das dritte, ist die Differenz in der Anzahl, der Streichhölzern, die, durch diese Veränderung, entsteht. Das vierte ist die Anzahl, an internen Umlegungen die nötig sind, für diese Veränderung. Um die „additions“ und „subtractions“ zu errechnen gleicht die Methode die Codes der beiden Ziffern ab. Danach berechnet sie die Differenz, indem sie die Differenz zwischen den „additions“ und den „subtractions“ berechnet. Für die Anzahl der internen Umlegungen wird der kleinere der ersten beiden Werte gewählt. Nach der Berechnung dieser Werte, werden diese in einem Array gespeichert. Dadurch müssen sie immer nur einmal berechnet werden. Die zweite Methode, die „getPotentials“-Methode (Codeausschnitt 3), bestimmt alle relevanten Potentiale, der als Parameter mitgegebene, Ziffer. Hierzu bildet die Methode alle Potentiale, der Ziffer, und gibt nur die relevanten zurück. Ein Potential wird dabei aus den Kosten zur Veränderung der Ziffer zu einer anderen gebildet. Dies macht die Methode für alle Veränderungen, der Ziffer, zu einer anderen Ziffer. Nachdem die Methode fertig ist, mit ihren Berechnungen, speichert sie, die Ergebnisse, in einem Array.

Kommen wir nun zur letzten Klasse, der „Number“-Klasse. Ein Objekt dieser Klasse, repräsentiert eine, mit Streichhölzern gelegte, Hexadezimalzahl. Dazu besitzt ein Objekt dieser Klasse ein Array, des Typs „Digit“. Des Weiteren enthält, jedes Objekt, dieser Klasse, eine Liste, welche Listen, für die Potentiale, der Ziffernfolgen der Zahl enthält. Die Klasse besitzt dazu noch vier wichtige Methoden. Die Erste, die wir betrachten werden, ist die „calculateNextPotentials“-Methode (Codeausschnitt 4). Diese macht eine, von zwei, Sachen. Ist die Liste, die die Listen der Potentiale enthält, leer, erstellt die Methode eine neue Liste, fügt dieser das Potential 0:0, und die Potentiale, der letzten Ziffer, der Zahl hinzu, und fügt die erstellte Liste, der „potentials“-Liste hinzu. Ist die Liste nicht leer, dann erstellt die

Methode eine leere Liste und begibt sich in eine foreach-Schleife, die über die Liste iteriert, die der Liste, mit Potentialen, als letztes hinzugefügt wurde. In dieser foreach-Schleife, wird eine neue foreach-Schleife aufgerufen. Diese iteriert, über die Potentiale, der ersten Ziffer, die nicht in der Ziffernfolge, deren Potentiale in der ausgewählten Liste, aus der erste foreach-Schleife, enthalten sind. Haben wir, beispielsweise, die Zahl „F431“ und die Potentiale der Liste, durch die die erste foreach-Schleife iteriert, sind von der Zahlenfolge „31“, dann iteriert die zweite foreach-Schleife, über die Potentiale der Ziffer „4“. Die Zeichenfolge, für die hier die Potentiale errechnet werden, ist dementsprechend „341“. Beim nächsten Aufruf, dieser Methode, würde die Methode also die Potentiale der Ziffernfolge „F341“ bestimmen. Innerhalb dieser foreach-Schleifen wird zu Beginn erst einmal ein neues „Potential“-Objekt, aus den momentanen Objekten, der foreach-Schleifen, erstellt. Es wird geprüft, ob bereits ein „Potential“-Objekt, in der neuen Liste enthalten ist, dass denselben Wert hat. Wenn nicht wird es der neuen Liste hinzugefügt. Wenn doch, wird geprüft, ob das neu erstellte Objekt, einen geringeren, „cost“-Wert hat, als das, dass bereits in der neuen Liste enthalten ist. Trifft diese Bedingung zu, so wird das Objekt, das bereits in der Liste enthalten ist, mit dem Neuen überschrieben. Am Ende der Methode wird die neue erstellte Liste, der „potentials“-Liste hinzugefügt.

Die zweite Methode, die wir betrachten werden, ist die „canMatchDifferenceBeDesolved“-Methode (Codeausschnitt 5). Dies Methode, dient dazu festzustellen, ob die Differenz, in der Anzahl der Streichhölzer, die durch eine Veränderung einer Ziffer, entsteht, abbaubar ist. Hierzu berechnet die Methode, zu Beginn, den invertierten Index, der Ziffer, an der Veränderungen vorgenommen werden sollen. Invertiert heißt, dass der Index null nicht bei der ersten, sondern bei der letzten Ziffer liegt. Danach wird der Potential-Index, als die Länge der „potentials“-Liste minus eins, berechnet. Danach wird das „wantedPotential“-Objekt erstellt. Es ist ein Objekt der Klasse „Potential“, dass nur einen Wert hat. Dieser wird bei der Initialisierung auf „matchDifference * -1“ gesetzt. Danach wird geprüft, ob „invertedDigitsIndex“ gleich null ist. Ist dem so wird geprüft, ob Streichholzdifferenz null ist. Wenn ja wird „true“ zurückgegeben, sonst „false“. Ist „invertedDigitsIndex“ nicht null, wird geprüft, ob „potentialsIndex“ größer oder gleich „invertedDigitsIndex – 1“ ist. Ist dem so, wird geprüft, ob das „wantedPotential“-Objekt in der Liste, mit dem Index „invertedDigitsIndex - 1“ enthalten ist. Ist dem so wird geprüft, ob die Anzahl an internen Umlegungen, die zur Nutzung dieses Potentials nötig sind, geringer, oder gleich der Anzahl an übrigen Umlegungen ist. Ist dem so, wird „true“ zurückgegeben, sonst „false“. Ist „potentialsIndex“ kleiner als „invertedDigitsIndex - 1“, dann wird, Anstelle der Rückgabe von „false“, die „calculateNextPotentials“-Methode aufgerufen und der Rückgabewert eines rekursiven Aufrufs, der hier beschriebenen Methode, mit unveränderten Parametern, zurückgegeben.

Die dritte Methode ist die „improve“-Methode. Diese lässt sich intern nochmal in zwei Teile aufteilen. Der Erste (Codeausschnitt 6) dient zur Verbesserung der Zahl, der Zweite (Codeausschnitt 7) dient dazu, Differenzen, in der Anzahl der Streichhölzer, vor und nach der Verbesserung, auszugleichen. Beschäftigen wir uns zuerst mit dem ersten Teil. Dieser beginnt mit der Initialisierung der „additions“- und der „subtractions“-Integer, auf null. Danach

iteriert die Methode über das „digits“-Array. Für jede Ziffer wird ein „betterDigits“-Array definiert. Es enthält alle Ziffern, im Hexadezimalsystem, die einen höheren Wert haben, als die Ziffer, die momentan, an der ausgewählte Stelle, in der Zahl, steht. Danach wird über dieses Array iteriert. Es wird geprüft, ob eine Veränderung der Ziffer, zu dieser bessern Ziffer, möglich ist, indem, mit der „canMatchDifferenceBeDesolved“-Methode geprüft wird ob die dadurch entstehende Differenz abbaubar ist, und geprüft wird, ob die Anzahl, an übrigen Umlegungen, ausreicht, um solche eine Veränderung durchzuführen. Treffen beide Bedingungen zu, wird die Veränderung durchgeführt, und die beiden Integer entsprechend verändert.

Kommen wir nun zum zweiten Teil. Hier wird zu Beginn, erst einmal, die momentane „matchDifference“ bestimmt. Dazu wird, „additions - subtractions“ gerechnet. Danach wird die Anzahl an übrigen Umlegungen berechnet hierzu wird, entweder Umlegungen minus „additions“, oder Umlegungen minus „subtractions“ gerechnet, je nachdem welche der beiden Variablen größer ist. Danach iteriert die Methode, rückwärts, über das Ziffernarray. Es wird, mit der „canMatchDifferenceBeDesolved“-Methode, der Index der Ziffer gefunden, bei der die „matchDifference“ noch abbaubar ist, bei dem nächsten Index aber nicht mehr. Dieser Index wird als „startDigitIndex“-Integer gespeichert. Danach wird eine for-Schleife, bei diesem Integer, plus eins, gestartet, welche solange läuft, wie die Variable der for-Schleife, kleiner als, die Länge des Ziffernarrays, ist. Innerhalb dieser Schleife wird über alle Ziffern, des Hexadezimalsystems, beginnend bei „F“ und endend bei „0“ iteriert. Es wird geprüft, ob die Ziffer, am Index, der Variable, der for-Schleife, zu dieser Ziffer verändert werden kann, wie im ersten Teil. Dadurch wird die Differenz in der Streichholzzahl Stück für Stück abgebaut.

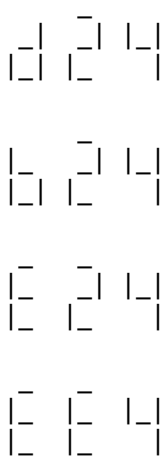
Die letzte wichtige Methode, der „Number“-Klasse, ist die „getInterimResults“-Methode (Codeausschnitt 8). Diese dient dazu, die einzelnen Umlegungen darzustellen, die nötig waren, um das „Number“-Objekt, zu der diese Methode gehört, aus dem „Number“-Objekt, das der Methode als Parameter mitgegeben wurde zu formen. Dazu erstellt die Methode, zu beginn, zwei neue Schlangen, für Integer. Danach iteriert die Methode über die Ziffernarrays der Ausgangszahl, und der verbesserten Zahl. Innerhalb dieser Iteration, iteriert die Methode, über die Codes, der beiden ausgewählten Ziffern, und vergleicht, die beiden, ausgewählten, Werte. Ist der Wert, der alten Zahl, null, und der, der Neuen, eins, wird der momentane Index in die „addititons“-Schlange eingereiht. Ist es andersherum wird der momentane Index in die „subtractions“-Schlange eingereiht. Der momentane Index errechnet, aus dem Ziffernindex, mal sieben plus den Codeindex. Danach wird ein neues Character-Array erstellt, dass alle Codes, aneinandergereiht und dann aufgespalten enthält. Als nächstes wird eine leere Zeichenkette erstellt. Zu dieser wird eine Darstellung, im Siebensegment-Still, des momentanen Character-Arrays hinzugefügt. Danach beginnt eine while-Schleife, die solange läuft, bis die „addition“-Schlange und damit auch die „substractions“-Schlange leer ist. Innerhalb dieser Schlange werden die ersten Objekte, beider Schlangen, aus diesen herausgenommen. Am Index, im Character-Array, der durch den Integer, aus der „additions“-Schlange, beschrieben wird, wird der Arraywert, auf ‚1‘ und am Index, aus der

„subtractions“-Schlange, auf ‚0‘ gesetzt. Danach wird das Character-Array, in der Siebensegmentdarstellung umgewandelt, und dem String hinzugefügt.

Beispiele

HexMax 0

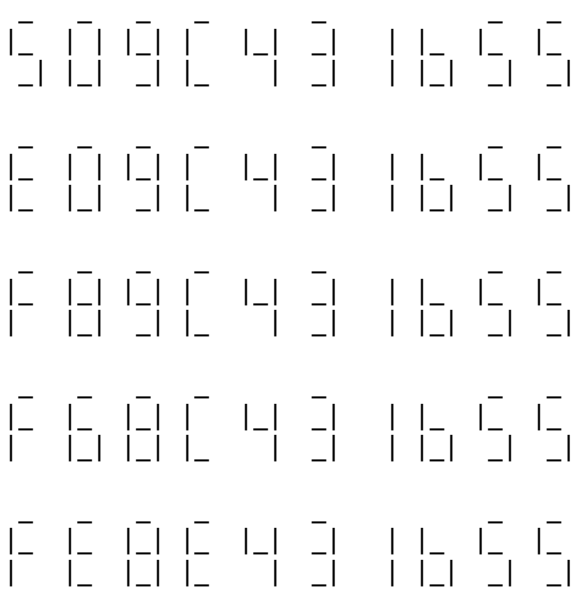
Ausgangszahl: D24

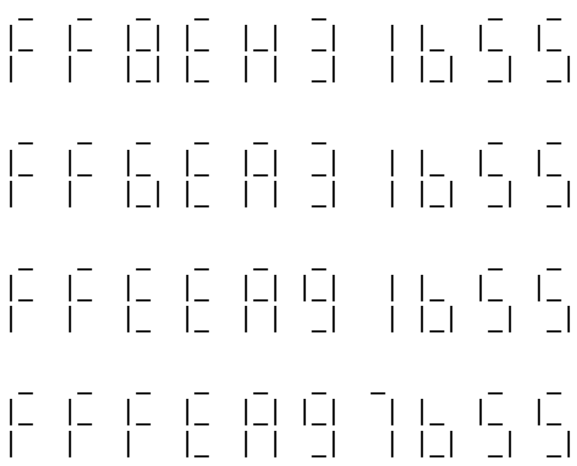


Verbesserte Zahl: EE4

HexMax 1

Ausgangszahl: 509C431B55





Verbesserte Zahl: FFFEA97B55

HexMax 2

Ausgangszahl: 632B29B38F11849015A3BCAEE2CDA0BD496919F8

Verbesserte Zahl: FFFFFFFFFFFFFFFFD9A9BEAEE8EDA8BDA989D9F8

HexMax 3

Ausgangszahl:
0E9F1DB46B1E2C081B059EAF198FD491F477CE1CD37EBFB65F8D765055757C6F4796
BB8B3DF7FCAC606DD0627D6B48C17C09

Verbesserte Zahl:
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFAA9
8BB8B9DFAFEAE888DD888AD8BA8EA8888

HexMax 4

Ausgangszahl:
1A02B6B50D7489D7708A678593036FA265F2925B21C28B4724DD822038E3B480419232
2F230AB7AF7BDA0A61BA7D4AD8F888

Verbesserte Zahl:
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEB8DE88BAA8ADD888898E9BA88A
D98988F898AB7AF7BDA8A61BA7D4AD8F888

HexMax 5

Ausgangszahl:
EF50AA77ECAD25F5E11A307B713EAAEC55215E7E640FD263FA529BBB48DC8FAFE
14D5B02EBF792B5CCBBE9FA1330B867E330A6412870DD2BA6ED0DBCAE553115C9
A31FF350C5DF993824886DB5111A83E773F23AD7FA81A845C11E22C4C45005D192AD
E68AA9AA57406EB0E7C9CA13AD03888F6ABEDF1475FE9832C66BFDC28964B7022B
DD969E5533EA4F2E4EABA75B5DC11972824896786BD1E4A7A7748FDF1452A5079E0
F9E6005F040594185EA03B5A869B109A283797AB31394941BFE4D38392AD12186FF6D
233585D8C820F197FBA9F6F063A0877A912CCBDCB14BEECBAEC0ED061CFF60BD51
7B6879B72B9EFE977A9D3259632C718FBF45156A16576AA7F9A4FAD40AD8BC87EC5
69F9C1364A63B1623A5AD559AAF6252052782BF9A46104E443A3932D25AAE8F8C59F
10875FAD3CBD885CE68665F2C826B1E1735EE2FDF0A1965149DF353EE0BE81F3EC13
3922EF43EBC09EF755FBD740C8E4D024B033F0E8F3449C94102902E143433262CDA19
25A2B7FD01BEF26CD51A1FC22EDD49623EE9DEB14C138A7A6C47B677F033BDEB84
9738C3AE5935A2F54B99237912F2958FDFB82217C175448AA8230FDCB3B3869824A82
6635B538D47D847D8479A88F350E24B31787DFD60DE5E260B265829E036BE340FFC0
D8C05555E75092226E7D54DEB42E1BB2CA9661A882FB718E7AA53F1E606

Verbesserte Zahl:
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
F88EFA9EBE89EFA99FBDAA8E8EAD88AB899F8E8F9AA9E9AD88988EDA9A99888E
DAD989A8BAFD8A88888888888888888888888888888888888888888888888888888888
8888888888888888888888888888888888888888888888888888888888888888888888888

88888888888888888888888888888888888888888888888888888888888888888888888888888888
88888888888888888888888888888888888888

Quellcode

```java
package com.hexMax;

public class Potential {
    private int subtractions;
    private int additions;

    private int value;
    private int cost;

    public Potential(int value) {
        this.value = value;
    }

    public Potential(int subtractions, int additions, int value, int cost) {
        this.subtractions = subtractions;
        this.additions = additions;
        this.value = value;
        this.cost = cost;
    }

    public Potential(Potential oldPotential, Potential digitPotential) {
        subtractions = oldPotential.getSubtractions() + digitPotential.getSubtractions();
        additions = oldPotential.getAdditions() + digitPotential.getAdditions();

        value = additions - subtractions;

        if(subtractions < additions) {
            cost = subtractions;
        }
        else {
            cost = additions;
        }
    }

    @Override
    public boolean equals(Object arg0) {
        Potential potential = (Potential) arg0;

        if(potential.getValue() == value) {
            return true;
        }
        else {
            return false;
        }
    }

    public int getSubtractions() {
        return subtractions;
    }

    public int getAdditions() {
        return additions;
    }

    public int getValue() {
        return value;
    }

    public int getCost() {
        return cost;
    }
}

```

Codeausschnitt 1: Potential-Klasse

```
91      public static int[] getChangeCosts(Digit from, Digit to) {
92          if(changeCosts[from.getValue()] [to.getValue()] [4] != 0) {
93              return changeCosts[from.getValue()] [to.getValue()];
94          }
95          else {
96              int additions = 0;
97              int subtractions = 0;
98
99              char[] fromCodeArray = from.getCode().toCharArray();
100             char[] toCodeArray = to.getCode().toCharArray();
101
102             for(int i = 0; i < fromCodeArray.length; i++) {
103                 if(fromCodeArray[i] == '0' && toCodeArray[i] == '1') {
104                     additions++;
105                 }
106                 else if(fromCodeArray[i] == '1' && toCodeArray[i] == '0') {
107                     subtractions++;
108                 }
109             }
110
111             int matchDifference  = additions - subtractions;
112             int cost;
113
114             if(subtractions < additions) {
115                 cost = subtractions;
116             }
117             else {
118                 cost = additions;
119             }
120
121             int[] changeCost = new int[5];
122
123             changeCost[0] = additions;
124             changeCost[1] = subtractions;
125             changeCost[2] = matchDifference;
126             changeCost[3] = cost;
127             changeCost[4] = 1;
128
129             changeCosts[from.getValue()] [to.getValue()] = changeCost;
130
131             return changeCost;
132         }
133     }
```

Codeausschnitt 2: getChangeCost-Methode

```java
public static Potential[] getPotentials(Digit digit) {
    if(potentials[digit.getValue()] [0] != null) {
        return potentials[digit.getValue()];
    }
    else {
        ArrayList<Potential> digitPotentials = new ArrayList<Potential>();

        Digit[] values = values();

        for(int n = 0; n < values.length; n++) {
            if(!values[n].equals(digit)) {
                int[] changeCost = getChangeCosts(digit, values[n]);

                int additions = changeCost[0];
                int subtractions = changeCost[1];
                int value = changeCost[2];
                int cost = changeCost[3];

                Potential potential = new Potential(subtractions, additions, value, cost);

                if(potential.getValue() != 0) {
                    if(!digitPotentials.contains(potential)) {
                        digitPotentials.add(potential);
                    }
                    else {
                        int exsistingPotentialIndex = digitPotentials.indexOf(potential);
                        Potential exsitingPotential = digitPotentials.get(exsistingPotentialIndex);

                        if(potential.getCost() < exsitingPotential.getCost()) {
                            digitPotentials.set(exsistingPotentialIndex, potential);
                        }
                    }
                }
            }
        }

        for(int n = 0; n < 5; n++) {
            potentials[digit.getValue()] [n] = digitPotentials.get(n);
        }

        return potentials[digit.getValue()];
    }
}
```

Codeausschnitt 3: getPotentials-Methode

```java
public void calculateNextPotentials() {
    if(potentials.size() == 0) {
        ArrayList<Potential> potentialsAtIndex = new ArrayList<Potential>();

        potentialsAtIndex.add(new Potential(0, 0, 0, 0));
        for(Potential potential : Digit.getPotentials(digits[digits.length - 1])) {
            potentialsAtIndex.add(potential);
        }

        potentials.add(potentialsAtIndex);
    }
    else {
        ArrayList<Potential> potentialsAtLastIndex = potentials.get(potentials.size() - 1);
        Potential[] digitPotentials = Digit.getPotentials(digits[digits.length - 1 - potentials.size()]);

        ArrayList<Potential> potentialsAtIndex = new ArrayList<Potential>();
        potentialsAtIndex.addAll(potentialsAtLastIndex);

        for(Potential lastIndexPotential : potentialsAtLastIndex) {
            for(Potential digitPotential : digitPotentials) {
                Potential newPotential = new Potential(lastIndexPotential, digitPotential);

                if(!potentialsAtIndex.contains(newPotential)) {
                    potentialsAtIndex.add(newPotential);
                }
                else {
                    int exsitingPotentialIndex = potentialsAtIndex.indexOf(newPotential);
                    Potential exsistingPotential = potentialsAtIndex.get(exsitingPotentialIndex);

                    if(newPotential.getCost() < exsistingPotential.getCost()) {
                        potentialsAtIndex.set(exsitingPotentialIndex, newPotential);
                    }
                }
            }
        }

        potentials.add(potentialsAtIndex);
    }
}
```

Codeausschnitt 4: calculateNextPotentials-Methode

```java
public boolean canMatchDifferenceBeDesolved(int matchDiffrence, int changesLeft, int digitIndex) {
    int invertedDigitsIndex = digits.length - 1 - digitIndex;
    int potentialIndex = potentials.size() - 1;

    Potential wantedPotential = new Potential(matchDiffrence * -1);
    if(invertedDigitsIndex == 0) {
        if(matchDiffrence == 0) {
            return true;
        }
        else {
            return false;
        }
    }
    else if(potentialIndex >= invertedDigitsIndex - 1) {
        if(potentials.get(invertedDigitsIndex - 1).contains(wantedPotential)) {
            int foundPotentialIndex = potentials.get(invertedDigitsIndex - 1).indexOf(wantedPotential);
            Potential foundPotential = potentials.get(invertedDigitsIndex - 1).get(foundPotentialIndex);

            if(foundPotential.getCost() <= changesLeft) {
                return true;
            }
            else {
                return false;
            }
        }
        else{
            return false;
        }
    }
    else {
        if(potentials.get(potentialIndex).contains(wantedPotential)) {
            int foundPotentialIndex = potentials.get(potentialIndex).indexOf(wantedPotential);
            Potential foundPotential = potentials.get(potentialIndex).get(foundPotentialIndex);

            if(foundPotential.getCost() <= changesLeft) {
                return true;
            }
            else {
                calculateNextPotentials();
                return canMatchDifferenceBeDesolved(matchDiffrence, changesLeft, digitIndex);
            }
        }
        else{
            calculateNextPotentials();
            return canMatchDifferenceBeDesolved(matchDiffrence, changesLeft, digitIndex);
        }
    }
}
```

Codeausschnitt 5: canMatchDifferenceBeDesolved-Methode

```java
public void improve(int changes) {
    int additions = 0;
    int subtractions = 0;

    for(int digitIndex = 0; digitIndex < digits.length; digitIndex++) {
        Digit[] betterDigits = digits[digitIndex].getBetterDigits();

        for(Digit betterDigit : betterDigits) {
            int[] changeCosts = Digit.getChangeCosts(digits[digitIndex], betterDigit);

            int additionsAfterChange = additions + changeCosts[0];
            int subtractionsAfterChange = subtractions + changeCosts[1];

            int matchDifferenceAfterChange = additionsAfterChange - subtractionsAfterChange; // This value descr
            int changesAfterChange = 0;

            if(subtractionsAfterChange > additionsAfterChange) {
                changesAfterChange = changes - subtractionsAfterChange;
            }
            else {
                changesAfterChange = changes - additionsAfterChange;
            }

            if(changesAfterChange < 0) {
                continue;
            }
            else {
                if(canMatchDifferenceBeDesolved(matchDifferenceAfterChange, changesAfterChange, digitIndex)) {
                    digits[digitIndex] = betterDigit;
                    additions += changeCosts[0];
                    subtractions += changeCosts[1];
                    break;
                }
                else {
                    continue;
                }
            }
        }
    }

```

Codeausschnitt 6: improve-Methode Teil 1

```
192         /*
193          * Desolving Matchdifference
194          */
195         int matchDifference = additions - subtractions;
196         int changesLeft = 0;
197 
198         if(subtractions > additions) {
199             changesLeft = changes - subtractions;
200         }
201         else {
202             changesLeft = changes - additions;
203         }
204 
205         int startDigitIndex = 0;
206 
207         for(int  digitIndex = digits.length - 1; digitIndex > -1; digitIndex--) {
208             if(canMatchDifferenceBeDesolved(matchDifference, changesLeft, digitIndex)) {
209                 startDigitIndex = digitIndex;
210                 break;
211             }
212         }
213 
214         for(int digitIndex = startDigitIndex + 1; digitIndex < digits.length; digitIndex++) {
215             for(Digit changedDigit : Digit.values()) {
216                 int[] changeCosts = Digit.getChangeCosts(digits[digitIndex], changedDigit);
217 
218                 int additionsAfterChange = additions + changeCosts[0];
219                 int subtractionsAfterChange = subtractions + changeCosts[1];
220 
221                 int matchDifferenceAfterChange = additionsAfterChange - subtractionsAfterChange; // This value describe
222                 int changesAfterChange = 0;
223 
224                 if(subtractionsAfterChange > additionsAfterChange) {
225                     changesAfterChange = changes - subtractionsAfterChange;
226                 }
227                 else {
228                     changesAfterChange = changes - additionsAfterChange;
229                 }
230 
231                 if(changesAfterChange < 0) {
232                     continue;
233                 }
234                 else {
235                     if(changesAfterChange < 0) {
236                         continue;
237                     }
238                     else {
239                         if(canMatchDifferenceBeDesolved(matchDifferenceAfterChange, changesAfterChange, digitIndex)) {
240                             digits[digitIndex] = changedDigit;
241                             additions += changeCosts[0];
242                             subtractions += changeCosts[1];
243                             break;
244                         }
245                         else {
246                             continue;
247                         }
248                     }
249                 }
250             }
251         }
252     }
```

Codeausschnitt 7: improve-Methode Teil 2

```
26⊖    public String getInterimResults(Number before) {
27
28         Queue<Integer> additions = new LinkedList<Integer>();
29         Queue<Integer> subtractions = new LinkedList<Integer>();
30
31         for(int digitIndex = 0; digitIndex < digits.length; digitIndex++) {
32             Digit unchangedDigit = before.getDigits()[digitIndex];
33             Digit changedDigit = digits[digitIndex];
34
35             char[] unchangedDigitCode = unchangedDigit.getCode().toCharArray();
36             char[] changedDigitCode = changedDigit.getCode().toCharArray();
37
38             for(int n = 0; n < 7; n++) {
39                 if(unchangedDigitCode[n] == '1' && changedDigitCode[n] == '0') {
40                     subtractions.offer((digitIndex * 7) + n);
41                 }
42                 else if(unchangedDigitCode[n] == '0' && changedDigitCode[n] == '1') {
43                     additions.offer((digitIndex * 7) + n);
44                 }
45             }
46         }
47
48         int numberCodeLenght = 7 * digits.length;
49         char[] numberCode = new char[numberCodeLenght];
50
51         for(int digitIndex = 0; digitIndex < digits.length; digitIndex++) {
52             char[] digitCode = before.getDigits()[digitIndex].getCode().toCharArray();
53             for(int n = 0; n < 7; n++) {
54                 numberCode[(digitIndex * 7) + n] = digitCode[n];
55             }
56         }
57
58         String interimResults = new String();
59
60         interimResults += convertCodeToString(numberCode);
61
62         while(!additions.isEmpty()) {
63             Integer addition = additions.poll();
64             Integer subtraction = subtractions.poll();
65
66             numberCode[addition] = '1';
67             numberCode[subtraction] = '0';
68             interimResults += convertCodeToString(numberCode);
69         }
70
71         System.out.println(interimResults);
72         return interimResults;
73     }
```

Codeausschnitt 8: getInterimResults-Methode