Aufgabe 3 - HexMax

Teilnahme-ID: 60302

Bearbeitet von
Florian Bange

25. April 2022

Inhaltsverzeichnis

1 Einleitung

Dieses Problem moechte ich loesen, indem ich das gleiche, allgemeinere Problem fuer beliebige Basen (im Folgenden „a“ genannt) angehe. Das Programm ist fuer die Basis 16 geschrieben.

Im Folgenden werden die sieben Positionen einer Siebensegmentanzeige „Segmente“ genannt. Weiter sind „aktivierte Segmente“ Segmente, welche durch ein „Staebchen“ belegt sind und „deaktivierte Segmente“ Segmente, welche frei sind.

2 Ziffern veraendern

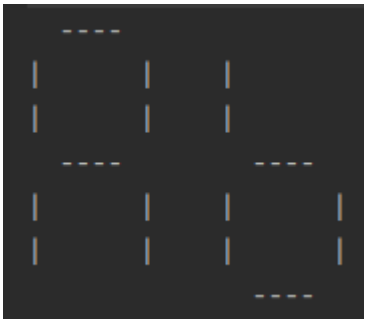
Um das Problem zu loesen, muss man die Ziffern der gegebenen Zahl systematisch veraendern.

Moechte man eine Siebensegmentanzeige d zu einer anderen Siebensegmentanzeige g veraendern, muss man dafuer Segmente in d aktivieren und/oder deaktivieren.
Aktivieren muss man die Segmente, welche in d nicht aktiviert, aber in g aktiviert sind und deaktivieren muss man die Segmente, welche in d aktiviert sind, aber in g nicht.

Der Prozess des Veraenderns einer Ausgangssiebensegmentanzeige d zu einer Zielsiebensegmentanzeige g wird im Folgenden schlicht Veraenderung genannt.
Die Anzahl der Segmente, die aktiviert werden muessen, nenne ich im Folgenden a und die Anzahl der Segmente, die deaktivieren werden muessen r.

Beispiele

1. Moechte man A zu B veraendern ...

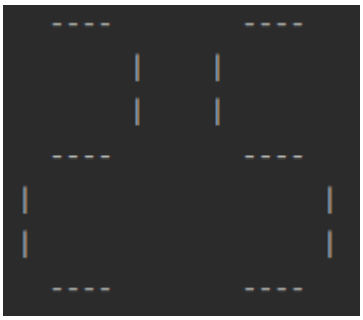


... muss man zwei Segmente, oben rechts deaktivieren und ein Segment unten aktivieren. Somit ist r=2 und a=1.

2. Moechte man A zu F veraendern ...

... muss man zwei Segmente rechts deaktivieren und keine Segmente aktivieren. Somit ist r=2 und a=0.

3. Moechte man 2 zu 5 veraendern ...



... muss man oben links und unten rechts ein Segment aktivieren und oben rechts und unten links ein Segment deaktivieren. Somit ist r=2 und a=2.

Selbstverstaendlich exestiert ebenfalls der Fall, dass r und a gleich 0 sind, falls d = g gilt.

Sei nun

$$changes : (d,\ g) \to (a,\ r)$$

eine Funktion, welche von Tupeln an Ziffern (d und g) abbildet auf Tupel der natuerlichen Zahlen mit Null (a, r).

Dabei stellt d die Ziffer der Ausgangssiebensegmentanzeige und g die Ziffer der Zielsiebensegmentanzeige dar. a ist die Anzahl der Segmente, welche aktiviert werden muessen und r die Anzahl der Segmente, welche deaktiviert werden muessen, um von d's Siebensegmentanzeige zu g's Siebensegmentanzeige zu kommen.

Pseudocode
Der Pseudocode zum Erhalten der Werte a und r fuer zwei Ziffern d und g sieht wie folgt aus:

```
1:  procedure GETADDANDREMOVE(d, g)
2:      segmentD_1, ... , segmentD_7 ← getSegments(d)
3:      segmentG_1, ... , segmentG_7 ← getSegments(g)
4:      a ← 0
5:      r ← 0
6:
7:      for i ← 1 to 7 do                          ▷ Go through segments
8:          if segmentD_i and not segmentG_i then
9:              r ← r + 1
10:         else not segmentD_i and segmentG_i
11:             a ← a + 1
12:         end if
13:     end for
14:
15:     return (a, r)
16: end procedure
```

Die Methode *getSegments* soll hier eine Methode darstellen, welche die sieben Segmente der Siebensegmentanzeige einer Ziffer als Wahrheitswerte zurueckgibt.

Die soeben beschriebene Vorgehensweise kann ebenfalls benutzt werden, um herrauszufinden, welche Segmente (bestimmt durch ihren Index) aktiviert/deaktiviert werden muessen, um von d zu g zu kommen.

3 Reihenfolge der Veraenderungen

Es ist leicht erkennbar, dass man versuchen sollte, die Ziffern von links nach rechts zu erhoehen. Denn fuer eine Zahl mit den Ziffern $d_n,\ d_{n-1},\ \dots\ ,\ d_2,\ d_1$ (von links nach rechts in dieser Reihenfolge - absteigend nummeriert) hat eine Ziffer an Index i $(1 \leq i \leq n)$ schon bei Erhoehung um eins mehr Einfluss auf den Wert der Zahl, als wuerde man alle anderen Ziffern von Index 1 bis $i-1$ auf den hoechsten Wert - $(a-1)$ - setzen.

Dies laesst sich wie folgt durch vollstaendige Induktion beweisen.

Seien

$$d_n,\ d_{n-1},\ \dots\ ,\ d_2,\ d_1$$

die Ziffern einer Zahl m mit Basis a, welche n Ziffern hat. Dabei liegt die Ziffer d_n ganz links in der Zahl und die Ziffer d_1 ganz rechts in der Zahl. Die Ziffern sind also von rechts nach links aufsteigend nummeriert.

Den Wert der Zahl m kann man als Summe mit Hilfe der Ziffern und der Basis wie folgt darstellen:

$$m = \sum_{i=1}^{n} d_i * a^{i-1}$$

Dabei hat jede Ziffer an Index i $(1 \leq i \leq n)$ den Stellenwert a^{i-1}.

Sei i eine natuerliche Zahl mit $1 \leq i \leq n$. Nun wird gezeigt, dass der Wert der Stelle i mit der Ziffer 1 groeszer ist als die Summe aller Stellen mit einem Index kleiner als i mit je der groeszten Ziffer, $a-1$.
Formal (Induktionsvoraussetzung):

$$1 * a^{i-1} > \sum_{k=1}^{i-1} (a-1) * a^{k-1}$$

Induktionsanfang $(i = 1)$

$$1 * a^{1-1} = a^{1-1} = a^0 = 1 > 0 = \sum_{k=1}^{0} (a-1) * a^{k-1} = \sum_{k=1}^{1-1} (a-1) * a^{k-1}$$

Induktionsbehauptung:

$$1 * a^{(i+1)-1} > \sum_{k=1}^{(i+1)-1} (a-1) * a^{k-1}$$

Induktionsschritt:

$$1*a^{(i+1)-1} = a^i = a*a^{i-1} = (a-1)*a^{i-1}+a^{i-1} \overset{IV}{>} (a-1)*a^{i-1}+\sum_{k=1}^{i-1}(a-1)*a^{k-1} = \sum_{k=1}^{i}(a-1)*a^{k-1} = \sum_{k=1}^{(i+1)-1}(a-1)*a^{k-1}$$

4 Eigenschaften einer korrekten Loesung

Damit eine Zahl eine korrekte Loesung ist, muss Folgendes gegeben sein:

1. Die Zahl ist gueltig (jede Ziffer existiert in der Basis)
2. Die Anzahl der Segmente ist die gleiche wie bei der gegebenen Zahl
3. Die gegebene Maximalzahl an Umlegungen ist nicht ueberschritten
4. Die Zahl ist groeszer als die gegebene Zahl oder gleich grosz

Aus diesen Eigenschaften lassen sich weitere Eigenschaften schlieszen.

Seien $d_n,\ \dots\ ,\ d_1$ die Ziffern der gegebenen Zahl von links nach rechts und $d'_n,\ \dots\ ,\ d'_1$ die jeweiligen Ziffern der korrekten Loesung. Weiter sind a_i und r_i die Anzahl an Segmenten, die man hinzufuegen, bzw. entfernen muss um von $d_i s$ Siebensegmentanzeige zu $d'_i s$ Siebensegmentanzeige zu kommen $(1 \leq i \leq n)$, bzw.

$$(a_i,\ r_i) = changes(d_i,\ d'_i)$$

fuer alle $i \in \mathbb{N}$ mit $(1 \leq i \leq n)$.

Weiter sei

$$A = \sum_{i=1}^{n} a_i$$

und

$$R = \sum_{i=1}^{n} r_i$$

Daraus, dass eine korrekte Loesung nach den zuvor beschriebenen Punkten gegeben ist, laesst sich Folgendes schlieszen:

1. $$\sum_{i=1}^{n} a_i - r_i = 0$$

   Dies liegt schlicht daran, dass somit nicht mehr Segmente entfernt oder hinzugefuegt werden muessen als existieren, so dass am Ende gleich viele Segmente vorhanden sind, wie bei der gegebenen Zahl.

2. A = R
   Dies laesst sich aus dem ersten Punkt wie folgt schlieszen:

$$\begin{aligned}
&\sum_{i=1}^{n} a_i - r_i = 0 \\
\Leftrightarrow\ &(a_1 - r_1) + \cdots + (a_n - r_n) = 0 \\
\Leftrightarrow\ &a_1 + (-r_1) + \cdots + a_n + (-r_n) = 0 \\
\Leftrightarrow\ &a_1 + \cdots + a_n + (-r_1) + \cdots + (-r_n) = 0 \\
\Leftrightarrow\ &a_1 + \cdots + a_n - r_1 - \cdots - r_n = 0 \\
\Leftrightarrow\ &a_1 + \cdots + a_n - (r_1 + \cdots + r_n) = 0 \\
\Leftrightarrow\ &\sum_{i=1}^{n} a_i - \sum_{i=1}^{n} r_i = 0 \\
\Leftrightarrow\ &A - R = 0 \\
\Leftrightarrow\ &A = R
\end{aligned}$$

3. Man benoetigt $\frac{A+R}{2}$ Umlegungen
   Dies wird ersichtlich, da A die Anzahl aller Segmente ist, die hinzugefuegt werden muessen (dort ist also noch kein aktiviertes Segment) und R die Anzahl der Segmente ist, die entfernt werden muessen (dort ist also ein aktiviertes Segment) (um von $d_1, \ldots, d_n$ zu $d'_1, \ldots, d'_n$ zu kommen). Nun kann man fuer jedes der R Segmente, die deaktiviert werden muessen, je eines der A Segmente, die aktiviert werden muessen, aktivieren. Dies ist moeglich, da $A = R$ gelten muss. Somit ergeben sich $\frac{A+R}{2}$ Umlegungen. Bzw., da A gleich R gilt, A = R Umlegungen.

Durch diese Schlussfolgerungen weisz man nun, welche Eigenschaften ein Loesungsweg ueberpruefen bzw. einhalten muss, um eine korrekte Loesung zu erzeugen.
Naemlich muss $A = R \leq m$ gelten, da wie erlaeutert A bzw. R Umlegungen benoetigt werden (so fern $A = R$ gilt).

5 Modellierung

Sei eine Zahl der Basis a mit n Ziffern gegeben. Die Ziffern seien nun $d_1, \ldots, d_n$ (d_1 ganz links, d_n ganz rechts).

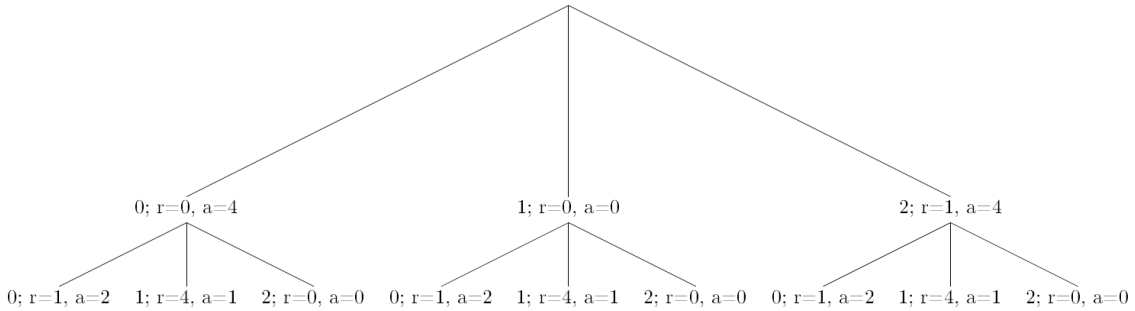
Alle moeglichen Belegungen der Ziffern $d_1, \ldots, d_n$ lassen sich als Baum darstellen.

Dabei ist die Wurzel auf Level 0 des Baums leer.
Fuer jeden Index i der Zahl ($1 \leq i \leq n$) gehen auf Level $i-1$ stets a Pfade von jedem Knoten auf Level $i-1$ zu je einem Knoten auf Level i ab. Diese a Knoten stellen je alle a Ziffern der Basis a ($0, \ldots, a-1$) dar.

Jeder Knoten auf Level i enthaelt eine Ziffer g, sowie das Tupel $changes(d_i, g)$

Die Anzahl an Blaettern ist demnach a^n und die Hoehe ist $n+1$.

Ein Beispielbaum zur Zahl 12 der Basis 3 (n ist somit 2) ist:



In einem solchen Baum waere nun die Aufgabe, den Pfad zu finden, bei dem die Summe A aller a's der Summe R aller r's entspricht.
Und fuer diese Summen gilt, dass sie beide nicht groeszer sind als die Maximalanzahl an Umlegungen (gemaesz der Eigenschaften einer korrekten Loesung).

Selbstverstaendlich soll dabei der Pfad gefunden werden, der als Zahl den groeszten Wert hat. Dazu koennte man die a Knoten, welche die a Ziffern der Basis a darstellen, immer absteigend oder immer aufsteigend (wie im Beispiel) hinzufuegen. Dann muesste man nur von links nach rechts bzw. von rechts nach links eine Tiefensuche durchfuehren und den ersten Pfad waehlen, welcher die zuvor beschriebenen Eigenschaften hat. Denn dann „erhoeht" man die Ziffern von links nach rechts, so dass man, wie zuvor bewiesen, die hoechstmoegliche Zahl findet.

6 Rekursiver Algorithmus

Im folgenden Loesungsweg wird ausschlieszlich die Berechnung der hoechsten Zahl umgesetzt ohne die Umlegungen zu beachten. Denn diese koennen deutlich einfacher im Nachhinein berechnet werden. Dazu spaeter mehr.

Aus der Darstellung als Baum kann man vermuten, dass sich das HexMax-Problem rekursiv formulieren laesst. Tatsaechlich ist dies wie folgt moeglich:

Geben ist erneut die Zahl der Basis a mit den n Ziffern $d_1, \ldots, d_n$ und die Maximalzahl an Veraenderungen m.

Fuer den rekursiven Algorithmus seien nun folgende Eingabeparameter gegeben:

1. Ein momentaner Index einer Ziffer: index ($1 \leq index \leq n$)
2. Die momentane Gesamtanzahl an Segmenten, die hinzugefuegt werden muessen: A
3. Die momentane Gesamtanzahl an Segmenten, die entfernt werden muessen: R

Der Rueckgabewert soll eine Liste an Ziffern der Groesze $n - index + 1$ oder NULL sein. NULL stellt den Fall dar, dass keine Loesung gefunden wird.

Die Aufgabe des Algorithmus lautet, ab dem gegebenen Index (inklusive index) die hoechste Belegung der Ziffern zu finden, welche die benoetigten Eigenschaften erfuellt: $A = R \leq m$ (wie unter Eigenschaften einer korrekten Loesung beschrieben).

Rekursion kann hier sehr schoen angewendet werden, indem an jedem Aufruf an einem Index i, alle moeglichen Ziffern g der Basis a absteigend durchgegangen werden. Dazu wird dann $(a,\ r) = changes(d_i,\ g)$ berechnet und die Funktion rekursiv benutzt, wobei als naechster Index index+1, als naechstes A A+a und als naechstes R R+r genommen werden.
Sollte der Rueckgabewert des rekursiven Aufrufs NULL sein, so wird die naechstkleinere moegliche Ziffer probiert. Sollte der Rueckgabewert nicht NULL sein, wird eine Liste erzeugt, welche an erster Position g hat und weiter mit dem Rueckgabewert gefuellt ist. Diese wird zurueckgegeben.

Sollte, nachdem alle Ziffern der Basis a absteigend durchgegangen wurden, jeder rekursive Aufruf NULL ergeben haben, so wird NULL zurueckgegeben.

Die Abbruchbedingungen des rekursiven Algorithmus stellen dar:

1. $A + R$ ist groeszer als $2 * m$
   In diesem Fall wuerden zu viele Umlegungen gebraucht, so dass NULL zurueckgegeben werden muss.
   Denn, wie aus korrekten Loesungen geschlossen wurde, muss $A = R \leq m$ gelten. Dies laesst sich zu $2A = 2R \leq 2m$ umformen. Dies wiederum entspricht $A + R \leq 2m$, da $A = R$ sein muss.
2. Ein zu groszer Index, bzw. ein Index der groeszer als n ist
   In diesem Fall wird ueberprueft, ob $A = R$ und $A \leq m$ bzw. $R \leq m$ gilt, nur dann wird eine leere Liste zurueckgegeben, sonst wird NULL zurueckgegeben.
   Wird die Reihenfolge, wie hier nummeriert, benutzt, kann die Abfrage nach $A \leq m$ bzw. $R \leq m$ natuerlich weggelassen werden, da sie durch Punkt 1 gegeben ist.
3. $A + R$ ist genau gleich $2 * m$
   Ist dies der Fall, so ist die Anzahl an Umlegungen genau erreicht. Weiter muss zusaetzlich ueberprueft werden, ob $A = R$ gilt.
   Ist dies gegeben, wird eine Teilmenge der Ziffern ab Index index (inklusive index) zurueckgegeben.

7 Loesung durch Dynamische Programmierung

Der soeben beschriebene Algorithmus loest zwar das Problem, indem die Uebergabeparameter $index = 1, A = 0$ und $R = 0$ benutzt werden, hat aber eine deutlich zu hohe Laufzeit von maximal ca. a^n.
Dies ergibt sich daraus, dass so maximal lange gesucht wuerde, bis das Ergebnis gleich der Eingabezahl ist.

Um diese Laufzeit stark zu senken, kann man einen Blick auf die Anzahl an moeglichen Eingaben fuer die jeweiligen Eingabeparameter werfen:

index	A	R
n	5n	5n

Dabei ergeben sich die Werte fuer A und R durch die Maximalzahl an Segmenten, die deaktiviert bzw. aktiviert werden koennen, um eine neue Ziffer zu erstellen: 5. Dies ist bei 1 und 8 der Fall. Multipliziert man diese Maximalzahl der Segmente, die aktiviert/deaktiviert werden koennen mit der Anzahl an Ziffern (n), ergibt sich 5n.

Multipliziert man die Anzahl an moeglichen Eingaben fuer jeden Eingabeparameter, ergibt sich

$$n * 5n * 5n = 25n^3.$$

Dieser Wert stellt die Maximalzahl an Eingaben fuer den Algorithmus dar.

Da dieser Wert deutlich langsamer waechst als a^n, kann man hier Memoisation bzw. Dynamische Programmierung benutzen. Dies bedeutet, dass man bereits berechnete Rueckgabwerte speichert, um sie ggf. wieder benutzen zu koennen.
Dazu wird am Anfang des Algorithmus ueberprueft, ob dieselbe Eingabe (index, A und R) bereits gespeichert wurde. Ist dies der Fall, wird das gespeicherte Ergebnis zurueckgegeben.
Damit dies moeglich ist, wird der Wert, welcher zurueckgegeben werden wird, gespeichert. Und zwar so, dass man den Rueckgabewert mit Hilfe der Eingabeparameter finden kann. Dies ist stets entweder eine Liste oder NULL.

Pseudocode
Nun folgt der Pseudocode fuer den zuvor beschriebenen rekursiven Algorithmus zusammen mit der Dynamischen Programmierung.

Gegeben, als feste Werte fuer den Algorithmus, sind die Ziffern der Zahl (nun von links nach rechts): $d_1, \ \dots \ , \ d_n$, wobei d_1 ganz links und d_n ganz rechts in der Zahl ist. Somit hat die Zahl n Ziffern - dieser Wert ist ebenfalls gegeben. Genauso, wie die Basis a der Zahl. Da diese nicht verwechselt werden sollte, mit dem a der *changes* Funktion, werden die „Rueckgabewerte“ dieser als $(adds, removes)$ bezeichnet.
Weiter ist die Maximalanzahl an Umlegungen m gegeben.
Die Eingabeparameter des Algorithmus sind wie beschrieben (index, A, R).

```
1:  memo ← [ ]                                              ▷ Memoisation object - dictionary
2:  procedure GETHIGHESTNUMBER(index, A, R)
3:      if memo.contains(index, A, R) then      ▷ Check if memo contains inputs meaning they were
    already calculated
4:          return memo.get(index, A, R)                                  ▷ Return them if so
5:      end if
6:
7:      if A + R > 2 * m then                ▷ Max amount of changes is overstepped, so returning null
8:          return NULL
9:      end if
10:
11:     if index > n then                              ▷ Index of digit is bigger than amount of digits (n)
12:         if A = R then          ▷ Check A = R, note that A + R ≤ 2 * m is given by previous check
13:             return [ ]                                                        ▷ Empty array
14:         else
15:             return NULL
16:         end if
17:     end if
18:
19:     if A + R = 2 * m then                          ▷ Max amount of changes exactly reached
20:         if A = R then                                                      ▷ Check A = R
21:             return [d_index, ... , d_n]
22:         else
23:             return NULL
24:         end if
25:     end if
26:
27:     for g ←(a-1) to 0 do                          ▷ Go through digits of base a, decreasing
28:         (adds, removes) ← changes(d_index, g)
29:         subResult ← getHighestNumber(index + 1, A + adds, R + removes)
30:         if subResult ≠ NULL then
31:             finalResult ← [ g ]
32:             add all subResult to finalResult
33:             memo.put(index, A, R, finalResult)      ▷ Put found result into memo for current inputs
34:             return finalResult
35:         end if
36:     end for
37:
```

```
38:     memo.put(index, A, R, NULL)                ▷ Put NULL into memo for current inputs
39:     return NULL
40: end procedure
```

Wichtig ist zu beachten, dass die Reihenfolge der Abbruchbedingung der Rekursion so eingehalten werden, dass stets sicher ist, dass $A = R \leq m$ gilt.

Weiter ist das Memoisation Objekt „memo“ zum Speichern der Ergebnisse hier vereinfacht dargestellt. Es sollen index, A und R als Schluessel (keys) und die Liste oder NULL als dazu assoziierter Wert (value) gesehen werden. Dazu unter Implementierung genaueres.

Weiter sei wiederholt, dass dieser Algorithmus mit den Uebergabeparametern Index = 1, A = 0 und R = 0 das eigentliche Problem loest, wobei der Rueckgabewert nicht die Zahl selbst sondern eine Liste ihrer Ziffern ist.

8 Berechnung der Umlegungen

Nach dem die hoechstmoegliche Zahl berechnet wurde, kann man mit dieser und der Ausgangszahl die benoetigten Umlegungen berechnen. Dabei muss beachtet werden, dass stets ein aktiviertes Segment mit einem deaktivierten Segment getauscht und niemals eine Siebensegmentanzeige vollstaendig geleert wird.

Gegeben seien die Ziffern der gegebenen Zahl $d_1, \ \dots \ , \ d_n$ und die Ziffern der resultierenden Zahl $d'_1, \ \dots \ , \ d'_n$.

Zunaechst wird fuer jedes Paar (d_i und d'_i) die Liste der Segmente, die entfernt werden muessen - $removes_i$ - und die Liste der Segmente, die hinzugefuegt werden muessen - $adds_i$, berechnet (wie unter Ziffern veraendern beschrieben).
Segmente werden dabei ueber ihren Index j ($0 \leq j < 7$) definiert.

Anschlieszend werden alle Indexe i der Hex-Zahl ($1 \leq i \leq n$) durchgegangen. Fuer jeden Index i werden so lange Elemente aus $adds_i$ mit Elementen aus $removes_i$ getauscht, wobei bei jedem Tausch die jeweiligen Segmente aus beiden Listen entfernt werden, bis eine Liste leer ist.

Als naechstes werden alle restlichen Segmente aus add- und remove-Listen miteinander getauscht, bis alle Listen leer sind.

Bei jedem der erwaehnten Taeusche wird dieser gespeichert.
Jeder Tausch von zwei Segmenten besteht dabei aus den Indexen der zwei Ziffern und aus den jeweiligen Indexen der Segmente in den Segmenten. Es ist moeglich, dass die Indexe der Ziffern dabei identisch sind.

Pseudocode
Nun wird der Pseudocode fuer den zuvor beschriebenen Algorithmus folgen.
Die Uebergabeparameter sind $d_1, \ \dots \ , \ d_n$ - die Ziffern der gegebenen Zahl - und $d'_1, \ \dots \ , \ d'_n$ - die Ziffern der resultierenden Zahl.

```
 1: procedure GETNEEDEDCHANGES(d_1, ... , d_n, d'_1, ... , d'_n)
 2:     adds_1, ... , add_n                                  ▷ Declare adds-lists
 3:     removes_1, ... , removes_n                           ▷ Declare removes-lists
 4:     for i ← 1 to n do                                    ▷ Get remove- and add-stacks for each index i
 5:         add_i ← neededAdds(d_i, d'_i)
 6:         removes_i ← neededRemoves(d_i, d'_i)
 7:     end for
 8:
 9:     for i ← 1 to n do                                    ▷ 1. Swap inside of digits
10:         while adds_i.size > 0 and removes_i.size > 0 do  ▷ Repeat until one stack is empty
11:             addSwap(add_i.pop(), removes_i.pop(), i)
12:         end while
13:     end for
14:
```

```
15:     addIndex ← 0
16:     removeIndex ← 0
17:     addStack ← [ ]
18:     removeStack ← [ ]
19:     isDone ← false
20:
21:     while !isDone do                                          ▷ 2. Swap left segments
22:         while addStack.size > 0 and addIndex ≤ n do ▷ Get next non-empty add stack if current is empty and it's possible
23:             addIndex ← addIndex + 1
24:             addStack ← adds_addIndex
25:         end while
26:
27:         while removeStack.size > 0 and removeIndex ≤ n do ▷ Get next non-empty remove stack if current is empty and it's possible
28:             removeIndex ← removeIndex + 1
29:             removeStack ← removes_removeIndex
30:         end while
31:
32:         if addStack.size = 0 then                             ▷ No swaps are left
33:             isDone ← true
34:         else
35:             addSwap(addIndex, add_i.pop(), removeIndex, removes_i.pop())
36:         end if
37:     end while
38:
39: end procedure
```

Die removes und adds Listen werden hier ueber die Methoden neededAdds und neededRemoves erhalten. Diese sollen Methoden dastellen, welche fuer zwei Ziffern ausgeben, welche Segmente (bestimmt durch ihren Index) entfernt bzw. hinzugefuegt werden muessen, um von der einen Ziffer zur anderen zu kommen. Dies soll geschehen, wie unter Ziffern veraendern beschrieben.

Die removes und adds Listen sollen hier des Weiteren Stacks sein, so dass das oberste Element des Stacks mit der pop-Methode bekommen und vom Stack entfernt werden kann.
Des Weiteren sollen die addSwap Methoden die Taeusche (swaps) speichern.
Zu beidem unter Implementierung genaueres.

Warum dieses Verfahren korrekt ist, wird unter Korrektheit argumentiert.

9 Optimierung

Optimieren kann man diesen Loesungsweg dadurch, dass alle Ergebnisse der changes-Funktion bereits vorgeneriert werden, so dass r und a nicht jedes Mal neu berechnet werden muessen. Genauer sollte fuer jedes Paar an Ziffern der Basis a die Liste an Segmenten (bzw. deren Indexe), welche hinzugefuegt/entfernt werden, vorgeneriert werden, so dass diese im Laufe des Programms nicht mehrmals berechnet werden.

10 Korrektheit

Die Korrektheit werde ich nun Punkt fuer Punkt begruenden.

Zunaechst zur Korrektheit der Berechnung der Umlegungen.
Hier ist zu beachten, dass eine Siebensegmentanzeige niemals vollkommen leer sein darf. Es muss also jede Siebensegmentanzeige immer mindestens ein aktiviertes Segment enthalten.

Im ersten Teil der Berechnung wird niemals eine Anzeige leer sein, da die Anzahl an aktivierten Segmenten in einer Anzeige die gleiche bleibt, wenn man eine Umlegung innerhalb der Anzeige vornimmt.
Im zweiten Teil der Berechnung gibt es fuer jede Ziffer an Index i drei Moeglichkeiten:

1. Es wird nichts nach auszen veraendert, da die Ziffer d'_i bereits erreicht ist.
   In diesem Fall kann die Anzeige nicht leer werden, da nichts veraendert wird.

2. $a_i < r_i$ - Es muessen $-(a_i - r_i)$ Segmente entfernt werden, so dass diese zu anderen Ziffern hinzugefuegt werden.
   In diesem Fall wird es niemals dazu kommen, dass die Anzeige an Index i vollkommen leer wird. Denn es ist sicher, dass durch das Entfernen der Segmente eine neue Ziffer entsteht. Und jede Ziffer besitzt aktivierte Segmente.

3. $a_i > r_i$ - Es muessen $a_i - r_i$ Segmente hinzugefuegt werden, sodass diese von anderen genommen werden muessen.
   In diesem Fall werden Segmente hinzugefuegt, sodass die Anzahl der aktivierten Segmente nicht kleiner wird und die Anzeige niemals leer wird.

Nun zur Korrektheit des rekursiven Algorithmus mit der Dynamischen Programmierung.

1. Die Anzahl an Umlegungen ist nicht groeszer als vorgegeben durch m.
   Dies wird durch die Abbruchbedingung $A + R > 2 * m$ gegeben, welche dafuer sorgt, dass niemals mehr Veraenderungen gebraucht werden als durch Umlegungen moeglich sind.

2. Die Anzahl der Ziffern ist nicht veraendert.
   Dies ist gegeben, da in dem Loesungsweg keine Moeglichkeit existiert, Ziffern hinzuzufuegen oder zu entfernen.

3. Die Anzahl der aktivierten Segmente ist die gleiche wie bei der gegebenen Zahl.
   Dies ist gegeben, da in den Abbruchbedingungen stets geprueft wird, dass $A = R$ ist, sodass gleich viele Segmente aktiviert und deaktiviert werden.

4. Die Loesung ist groeszer oder gleich grosz zur gegebenen Zahl.
   Dies ist gegeben, da der Algorithmus maximal eine Loesung sucht, bis $d_i = d_i$ fuer alle Indexe i gilt. In diesem Fall wird der Algorithmus den Abbruchfall des letzten Index erreichen und die Loesung zurueckgeben, da A und R immer 0 bleiben. Dies ist die kleinstmoegliche Loesung, die der Algorithmus finden kann, da die moeglichen Ziffern stets absteigend probiert werden.

11 Implementierung

Die Umsetzung des Algorithmus wurde in Java 8 vorgenommen. Im Programm wurde die Basis 16 implementiert, so dass Hex-Zahlen dargestellt werden koennen.
Bei der Umsetzung sind folgende Dinge wichtig.

Siebensegmentanzeige
Zunaechst benoetigt man eine Implementierung fuer eine hexadezimale Zahl, sowie fuer die Siebensegmentanzeigen.
Die Siebensegmentanzeige wurde als unveraenderliche Klasse (immutable class) implementiert, welche ein siebenelementiges Array an booleans enthaelt, welches die Segmente darstellt. True bedeutet, das Segment ist aktiviert, false bedeutet, das Segment ist deaktiviert. Des weiteren enthaelt die Klasse Methoden, um Segmente zu setzen und zu erhalten (als boolean; an einem index), sowie um zu ueberpruefen, ob die Anzeige leer ist. Methoden, welche das Objekt veraendern, geben eine neue Anzeige zurueck.

Auszerdem braucht man eine Moeglichkeit mehrere Siebensegmentanzeigen darzustellen, so dass man eine Hex-Zahl representieren kann. Dazu gibt es eine Klasse „SSDSet“, welche ein Array an Siebensegmentanzeigen enthaelt. Die Klasse ist ebenfalls immutable und enthaelt eine Methode, um das Array zu erhalten, sowie eine Methode, um eine Siebensegmentanzeige an einem index im Array neu zu setzen.

Die beiden Typen sind immutable, damit keine Werte unerwartet veraendert werden. Aufgrund des Java Garbage Collectors wird dadurch auch kein Speicherplatz unnoetig belegt, da dieser nicht mehr referenzierte/genutzte Objekt entfernt.

Die 16 hexadezimalen Zahlen von 0 bis F habe ich mit einer Enumeration umgesetzt, diese enthaelt fuer jede Ziffer den Namen der Ziffer (z.B. „0“ oder „F“), den Wert der Ziffer in dezimal, sowie ein

boolean-array fuer die Siebensegmentanzeige und Methoden, um diese Eigenschaften zu erhalten. Des Weiteren gibt es eine Utility (Hilfs-) Klasse um hexadezimale Zahlen durch den Namen oder den Wert zu erhalten, sowie um alle Hex-Zahlen sortiert nach einer bestimmten Eigenschaft als Liste zu erhalten.

Umsetzung der Optimierung
Das unter Optimierung erwaehnte Speichern der Listen an Segmenten, welche hinzugefuegt/entfernt werden muessen, fuer jedes Paar an Ziffern der Basis a, wird umgesetzt durch die Klasse HexDigitChanges, welche eine statische Singleton besitzt.
Diese Klasse speichert die Listen an Indexen der Segmente fuer alle moeglichen Paare an Ziffern und verfuegt ueber eine Methode diese Listen fuer zwei angegebene Ziffern zu erhalten.
Die Singleton wird benutzt, um sicher zu gehen, dass das Objekt nicht mehrmals erstellt wird, so dass die Berechnung nicht mehrfach vorgenommen werden.

Berechnung der hoechsten Zahl mit Hilfe Dynamischer Programmierung
Bei diesem Algorithmus wurde als Rueckgabewert ein Array an HexDigits (Hex-Ziffern) gewaehlt.
Ein Array ist hier sinnvoller als eine Liste, da die Laenge bereits bei der Dekleration des Arrays bekannt ist. Somit muessen keine Arrayvergroeszerungen vorgenommen werden (wie ggf. bei einer ArrayList).

Memoisation Objekt
Nun zur Umsetzung des Memoisation Objekts, welches Rueckgabewerte des Algorithmus speichert und bei erneutem Aufruf des Algorithmus mit gleichen Uebergabeparametern diese direkt zurueckgibt.
Das Memoisation Objekt wird implementiert als geschachtelte HashMap. Die aeuszerste HashMap zeigt von Indexen auf die zweite HashMap, die zweite HashMap zeigt von A-Werten auf die innere HashMap. Diese zeigt von R-Werten auf HexDigit Arrays.
Um den Quellcode simpel zu halten, wird vor Begin des Algorithmus die aeuszerste HashMap vollkommen gefuellt. Dadurch ist es einfacher zu ueberpruefen, welche inneren HashMaps welche Schluessel enthalten.

Soll fuer beliebige Werte Index, A, R ueberprueft werden, ob eine Eingabe bereits berechnet wurde, so wird mit dem Index als Schluessel in der aeuszerste HashMap die zweite HashMap erhalten. In dieser wird ueberprueft, ob sie A als Schluessel enthaelt. Ist dies nicht der Fall, wurden die Eingabeparameter noch nicht berechnet. Andernfalls muss ueberprueft werden, ob die innerste HashMap, welche man mit A erhaelt, R als Schluessel enthaelt.

Soll ein Rueckgabewert gefunden werden, muss nicht ueberprueft werden, ob die HashMaps die Schluessel enthalten, da dies nur passiert, wenn zuvor ueberprueft wurde, ob sie dies tun. Somit muss nur aus der ersten HashMap die zweite erhalten werden, aus der zweiten die dritte und aus dieser der Rueckgabwert.

Soll ein Rueckgabewert result fuer die Uebergabeparameter index, A, R gespeichert werden, muss aus der ersten HashMap die zweite erhalten werden. Bei der zweiten muss ueberprueft werden, ob diese A als Schluessel enthaelt. Ist dies der Fall, kann die innere HashMap erhalten werden. Ist dies nicht der Fall, muss eine neue HashMap erstellt werden, welche mit dem Schluessel A in der zweiten HashMap gespeichert wird. Anschlieszend wird entweder in der erhaltenen HashMap oder der neu erstellten HashMap result mit dem Schluessel R gespeichert.

Berechnung der Umlegungen
Bei der Berechnung der Umlegungen von der Ausgangszahl zur resultierenden Zahl werden die Listen der Segmente, welche entfernt/hinzugefuegt werden muessen (bzw. deren Indexe), in ArrayDeques gespeichert. Diese ermoeglichen die stacktypische Methode pop, welche das „oberste“ Element der Liste zurueckgibt und aus der Liste entfernt.
Diese Datenstruktur wird verwendet, da jedes Element aller Listen genau einmal benoetigt wird, wenn es getauscht wird.
Dadurch muss man nicht zusaetzlich Elemente aus den Listen entfernen.

Die erwaehnten Taeusche werden objektorientiert umgesetzt mit einer eigenen Klasse, welche die Indexe der Ziffern und die Indexe der Segmente in den Ziffern speichert.
Bei der Berechnung der Umlegungen erhaelt man also am Ende eine Liste an Taeuschen.

Moechte man die Taeusche auf die wirklichen Hex-Ziffern anwenden, um Tests einfacher zu machen

oder die Veraenderungen grafisch auszugeben, so benoetigt man eine Moeglichkeit, die Veraenderungen bzw. Umlegungen der Ausgangszahl zu der resultierenden Zahl zu speichern.
Dafuer gibt es eine Klasse, welche eine LinkedList (verkettete Liste) enthaelt in welcher SSDSets gespeichert werden. In dieser Klasse gibt es eine Methode, um ein neues SSDSet Objekt zur LinkedList hinzuzufuegen. Diese wird benutzt um neue, veraenderte SSDSet Objekte waehSrend der Berechnung der Umlegungen zu speichern.
Eine LinkedList wird genutzt, da kein sofortiger Zugriff auf bestimmte Elemente in der Liste benoetigt wird. Durch eine LinkedList ist zudem das Vergroeszern eines Array (wie bei einer ArrayList) nicht noetig.

12 Laufzeitanalyse

Die worst-case Laufzeit des gesamten Algorithmus setzt sich zusammen aus folgenden Teilen:

1. Die Vorgenerierung der r's und a's von allen Ziffern zu allen Ziffern.
   Diese laeuft in konstanter Zeit, da stets genau a^2 Durchlaeufe benoetigt werden, wobei a kein Teil der Eingabe ist.
   Dieser Teil laeuft somit in $\mathcal{O}(1)$.

2. Berechnung der hoechsten Zahl mit Hilfe Dynamischer Programmierung
   Diese Laufzeit stellt sich als etwas komplizierter heraus.

   Zunaechst sei zu wissen, dass die Laufzeit eines Algorithmus mit Memoisation berechnet wird, indem die Anzahl der Teilprobleme mit der Laufzeit pro Teilproblem multipliziert wird, wobei der rekursive Teil als konstant ($\mathcal{O}(1)$) gewertet wird.

   Die Laufzeit pro Teilproblem besteht aus ...

   a) 4 Abfragen konstanter Zeit, $\mathcal{O}(1)$
   b) einer Schleife mit maximal a Durchgaengen, $\mathcal{O}(a) = \mathcal{O}(1)$
   c) einem Kopieren von maximal n Elementen pro Schleifendurchgang, $\mathcal{O}(a * n) = \mathcal{O}(n)$
   d) einem Speichern des Ergebnisses in konstanter Zeit, $\mathcal{O}(1)$

   a feallt hier als Faktor weg, da a kein Teil der Eingabe ist.

   Weiter hat das Speichern und Erhalten der bereits gespeicherten Rueckgabewerte eine konstante Laufzeit, da die Schluessel der HashMaps ausschlieszlich 32-bit-Integer sind, so dass fuer die hash-Funktion der Klasse HashMap, welche den hashcode der Integer durch bitweise Operationen bearbeitet, eine konstante Zeit gebraucht wird.
   Genauer sind diese bitweisen Operationen das XOR und bitweise Verschiebungen.

   Insgesamt erhaelt man also eine Laufzeit von

   $$\mathcal{O}(1) + \mathcal{O}(1) + \mathcal{O}(n) + \mathcal{O}(1) = \mathcal{O}(n)$$

   pro Teilproblem.

   Die Anzahl der Teilprobleme kann man angehen ueber die Eingabeparameter des Algorithmus: index, A und R
   Wie bereits beschrieben, erhaelt man theoretisch

   $$n * 5n * 5n = 25 * n^3$$

   moegliche verschiedene Eingabeparameter.

   Die gesamte Laufzeit des eigentlichen Algorithmus ist somit

   $$25 * n^3 * \mathcal{O}(n) = O(25n^3 * n) = O(n^4)$$

3. Berechnung der Umlegungen
   Ein simpler Weg die Laufzeitkomplexitaet der Berechnung der Umlegungen anzugehen ist, dass fuer

die bereits beschriebenen Werte A und R (aller zu entfernenden und hinzuzufuegenden Segmente) gilt, dass

$$\frac{A+R}{2}$$

die Anzahl an Umlegungen ist (siehe Punkt 5 - Schluesse aus der korrekten Loesung).

Da, wie soeben beschrieben, A und R je maximal $5n$ sein koennen, laesst sich dies umformen zu

$$\frac{5n+5n}{2}$$
$$= \frac{10n}{2}$$
$$= 5n$$

Die Laufzeit ist somit

$$\mathcal{O}(5n) = \mathcal{O}(n).$$

Demnach ist die gesamte Laufzeitkomplexitaet

$$\mathcal{O}(1) + \mathcal{O}(n^4) + \mathcal{O}(n) = \mathcal{O}(n^4).$$

13 Platzkomplexitaet

Die Platzkomplexitaet ist sehr aehnlich zur Laufzeitkomplexitaet und besteht aus denselben folgenden Teilen:

1. Die Vorgenerierung der r's und a's von allen Ziffern zu allen Ziffern
   Diese hat auch hier eine Komplexitaet von $\mathcal{O}(1)$, da a^2 Ziffern mit je sieben Segmenten gespeichert werden muessen.

2. Berechnung der hoechsten Zahl mit Hilfe Dynamischer Programmierung
   Bei dieser wird zum einem pro Aufruf des Algorithmus ein Array mit maximal n Elementen erzeugt.

   Weiter muessen aufgrund der Rekursion maximal n Ruecksprungadressen und eine bestimmte Anzahl an lokalen Variablen, welche von n unabhaengig sind, gespeichert werden. Zu beachten ist dabei, dass das erwaehnte Array keine dieser lokalen Variablen ist, da es erst entsteht, wenn der rekursive Teil des Algorithmus vorbei ist.

   Diese beiden Teile ergeben also eine Platzkomplexitaet von O(n).

   Weiter werden maximal $25n^3$ Rueckgabewerte gespeichert, welche je eine maximal Groesze von n haben.
   Dadurch entsteht eine Platzkomplexitaet von

   $$\mathcal{O}(25n^3 * n) = \mathcal{O}(n^4).$$

3. Berechnung der Umlegungen
   Bei der Berechnung der Umlegungen werden zum einen die Indexe der Segmente, die entfernt bzw. hinzugefuegt werden muessen, fuer jede der n Ziffern gespeichert.
   Dadurch entsteht eine Platzkomplexitaet von maximal

   $$\mathcal{O}(2 * 5 * n) = \mathcal{O}(n).$$

   Denn es gibt n Ziffern und fuer jede maximal 5 Segmente, die entfernt oder hinzugefuegt werden koennen.

   Zum anderen werden die Taeusche gespeichert.
   Wie bereits erlaeutert, gibt es maximal $5n$ Taeusche. Bei jeder dieser werden 4 Werte gespeichert.
   Somit ist die Platzkomplexitaet hier

   $$\mathcal{O}(4 * 5n) = \mathcal{O}(n).$$

Somit ist die gesamte Platzkomplexitaet ebenfalls

$$\mathcal{O}(1) + \mathcal{O}(n^4) + \mathcal{O}(n) = \mathcal{O}(n^4).$$

14 Beispiele

Nun folgen die Ergebnisse des Programms fuer die Eingaben der bwinf website.
Dabei wurden stets die Eingabe, die maximale Anzahl an Umlegungen, die Ausgabe, die Anzahl an genutzten Umlegungen und die benoetigte Zeit angegeben. Zusaetzlich wurde bei den Eingabedateien „hexmax0.txt“, „hexmax1.txt“ und „hexmax2.txt“ die Umlegungen angegeben.

14.1 hexmax0.txt

Eingabe:
Zahl:

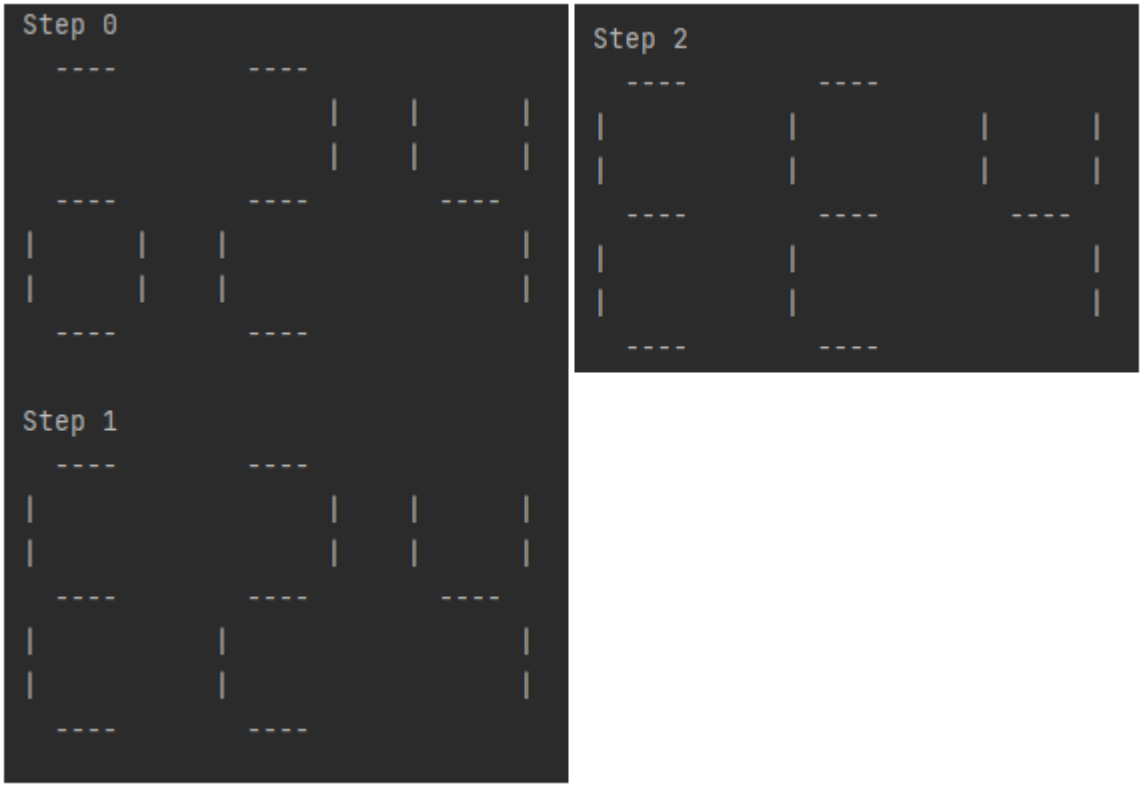
D24

Maximale Anzahl an Umlegungen: 3
Ausgabe:

EE4

Anzahl genutzter Umlegungen: 3
Zeit (mit Ausgabe der Umlegungen): 2 ms

Umlegungen:



14.2 hexmax1.txt

Eingabe:
Zahl:

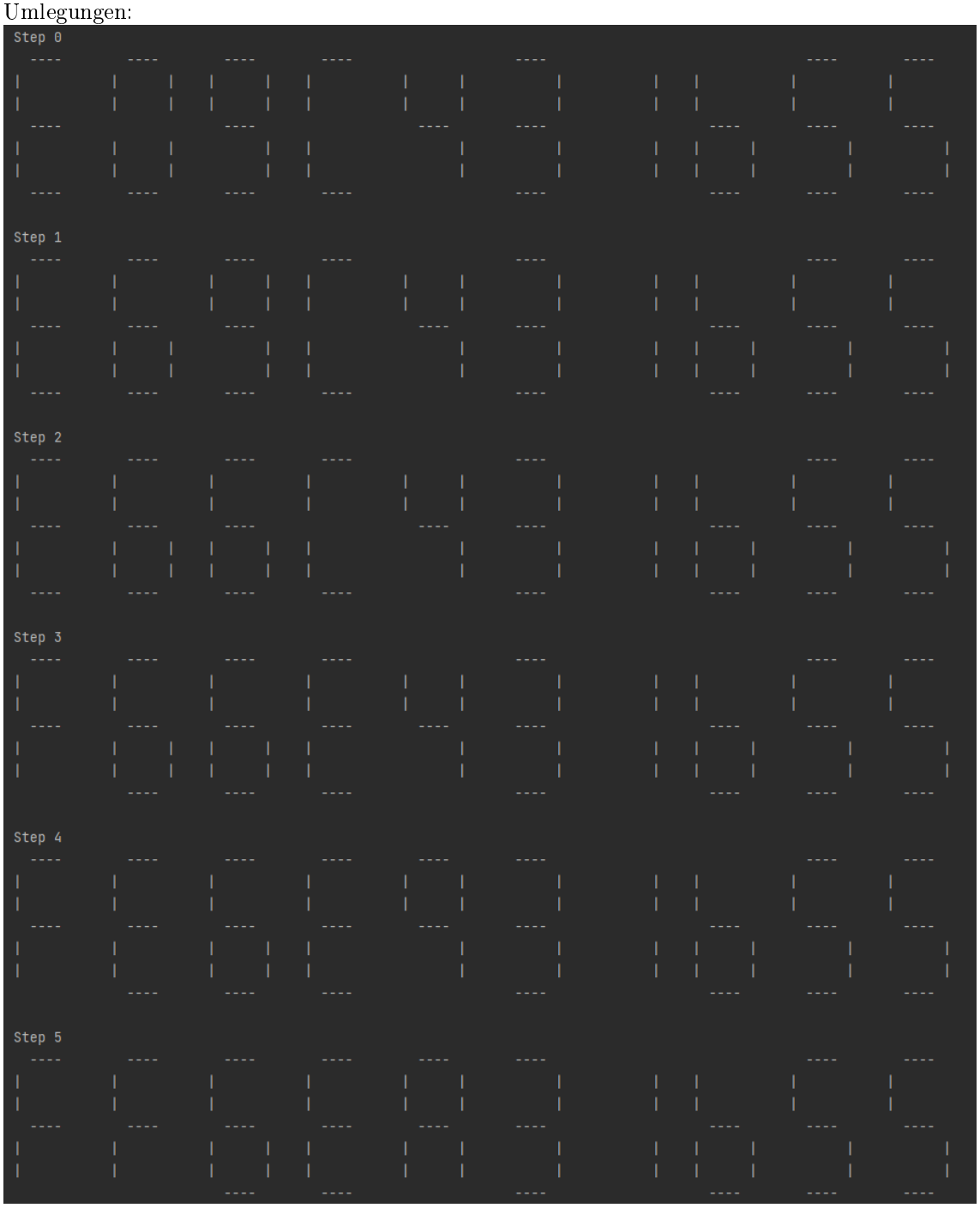
509C431B55

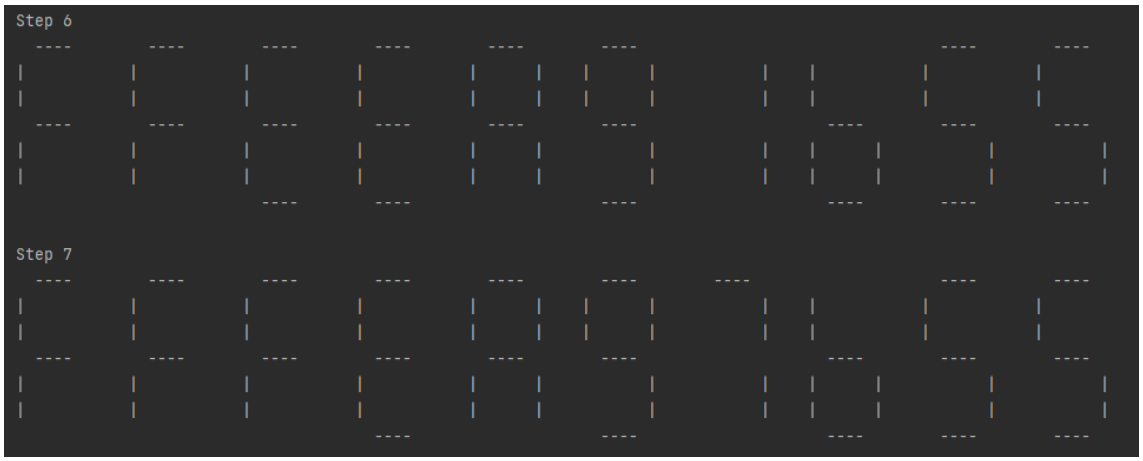
Maximale Anzahl an Umlegungen: 8
Ausgabe:

FFFEA97B55

Anzahl genutzter Umlegungen: 8
Zeit (mit Ausgabe der Umlegungen): 16 ms

Umlegungen:





14.3 hexmax2.txt

Eingabe:
Zahl:

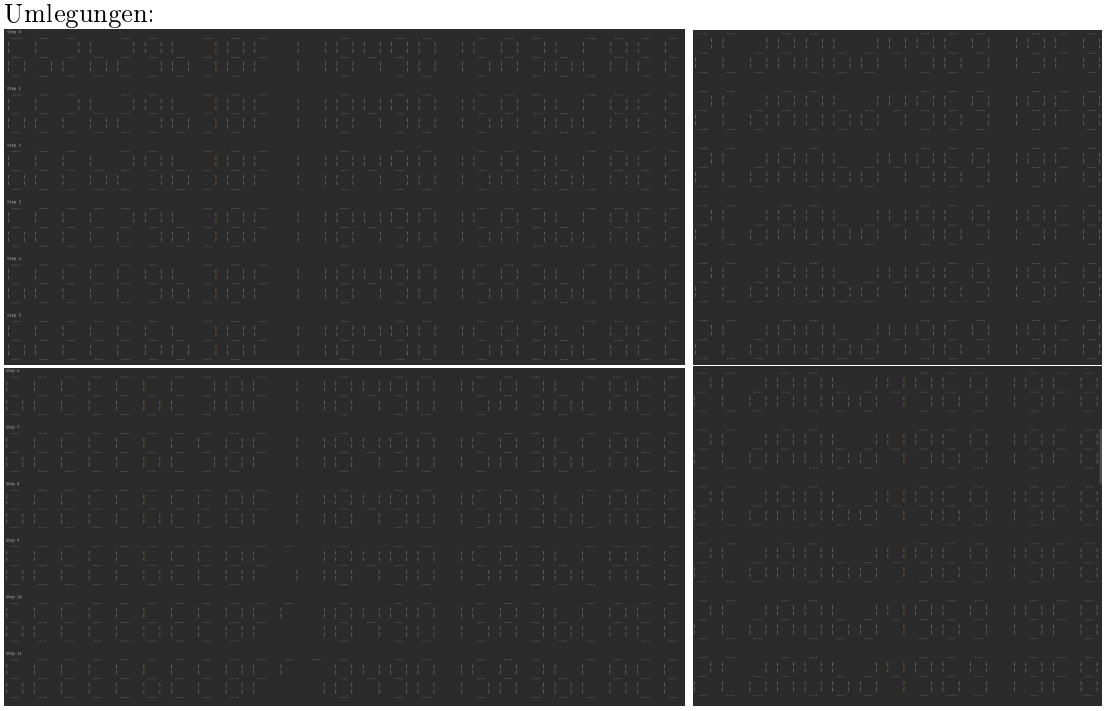
632B29B38F11849015A3BCAEE2CDA0BD496919F8

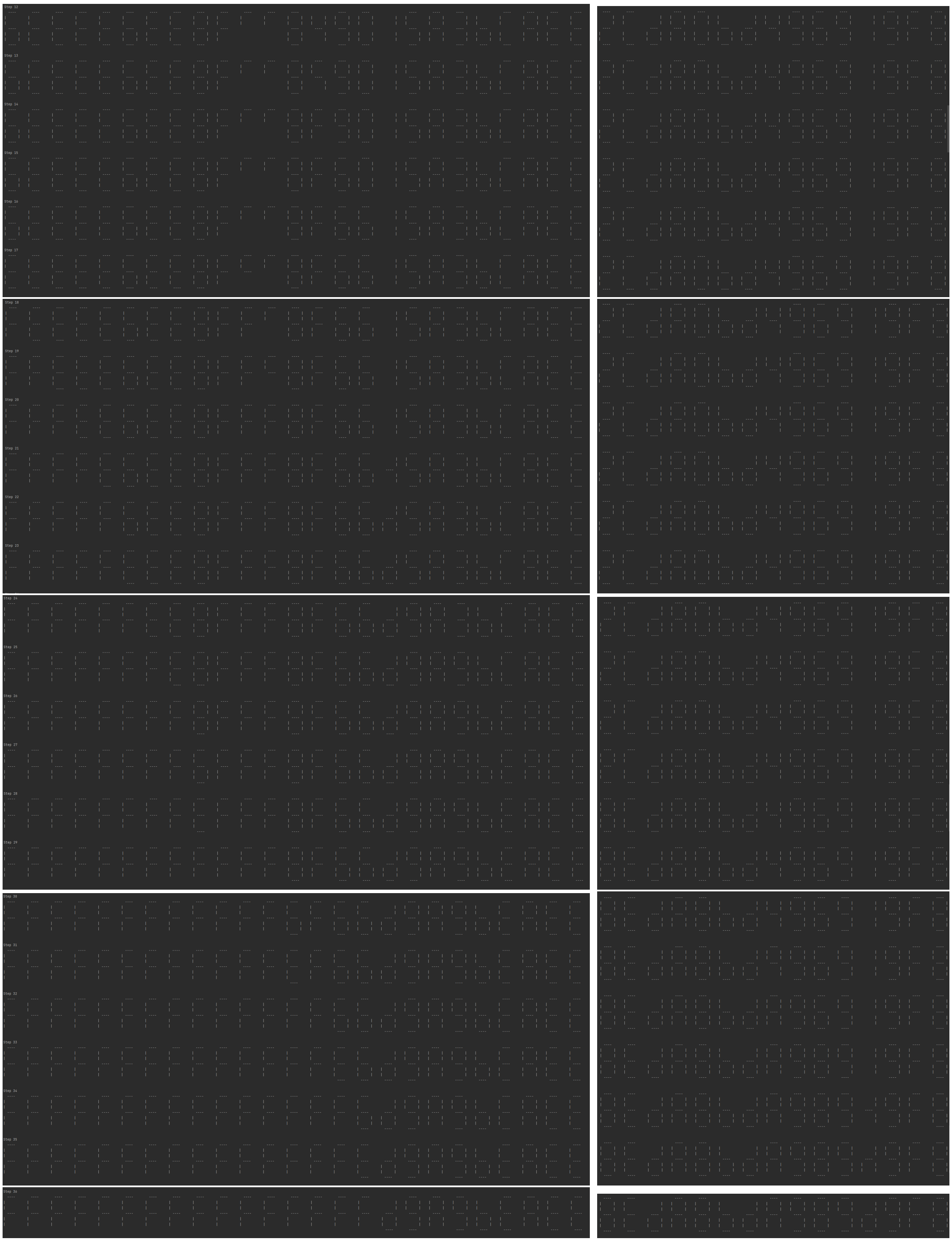
Maximale Anzahl an Umlegungen: 37
Ausgabe:

FFFFFFFFFFFFFFFFD9A9BEAEE8EDA8BDA989D9F8

Anzahl genutzter Umlegungen: 37
Zeit (mit Ausgabe der Umlegungen): 32 ms

Umlegungen:





14.4 hexmax3.txt

Eingabe:
Zahl:

0E9F1DB46B1E2C081B059EAF198FD491F477CE1CD37EBFB65F

8D765055757C6F4796BB8B3DF7FCAC606DD0627D6B48C17C09

Maximale Anzahl an Umlegungen: 121
Ausgabe:

FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF

FFFFFFFFFFFFFFFAA98BB8B9DFAFEAE888DD888AD8BA8EA8888

Anzahl genutzter Umlegungen: 121
Zeit (ohne Ausgabe der Umlegungen): 48 ms

14.5 hexmax4.txt

Eingabe:
Zahl:

1A02B6B50D7489D7708A678593036FA265F2925B21C28B4724

DD822038E3B4804192322F230AB7AF7BDA0A61BA7D4AD8F888

Maximale Anzahl an Umlegungen: 87
Ausgabe:

FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEB8DE88BAA8A

DD888898E9BA88AD98988F898AB7AF7BDA8A61BA7D4AD8F888

Anzahl genutzter Umlegungen: 87
Zeit (ohne Ausgabe der Umlegungen): 77 ms

14.6 hexmax5.txt

Eingabe:
Zahl:

EF50AA77ECAD25F5E11A307B713EAAEC55215E7E640FD263FA

529BBB48DC8FAFE14D5B02EBF792B5CCBBE9FA1330B867E330

A6412870DD2BA6ED0DBCAE553115C9A31FF350C5DF99382488

6DB5111A83E773F23AD7FA81A845C11E22C4C45005D192ADE6

8AA9AA57406EB0E7C9CA13AD03888F6ABEDF1475FE9832C66B

FDC28964B7022BDD969E5533EA4F2E4EABA75B5DC119728248

96786BD1E4A7A7748FDF1452A5079E0F9E6005F040594185EA

03B5A869B109A283797AB31394941BFE4D38392AD12186FF6D

233585D8C820F197FBA9F6F063A0877A912CCBDCB14BEECBAE

C0ED061CFF60BD517B6879B72B9EFE977A9D3259632C718FBF

45156A16576AA7F9A4FAD40AD8BC87EC569F9C1364A63B1623

A5AD559AAF6252052782BF9A46104E443A3932D25AAE8F8C59

F10875FAD3CBD885CE68665F2C826B1E1735EE2FDF0A196514

9DF353EE0BE81F3EC133922EF43EBC09EF755FBD740C8E4D02

4B033F0E8F3449C94102902E143433262CDA1925A2B7FD01BE

F26CD51A1FC22EDD49623EE9DEB14C138A7A6C47B677F033BD

EB849738C3AE5935A2F54B99237912F2958FDFB82217C17544

8AA8230FDCB3B3869824A826635B538D47D847D8479A88F350

E24B31787DFD60DE5E260B265829E036BE340FFC0D8C05555E

75092226E7D54DEB42E1BB2CA9661A882FB718E7AA53F1E606

Maximale Anzahl an Umlegungen: 1369
Ausgabe:

FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF

FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFF88EFA9EBE89EFA99FBDAA8E8EAD88
AB899F8E8F9AA9E9AD88988EDA9A99888EDAD989A8BAFD8A88
88888888888888888888888888888888888888888888888888
88888888888888888888888888888888888888888888888888
88888888888888888888888888888888888888888888888888
88888888888888888888888888888888888888888888888888
88888888888888888888888888888888888888888888888888

Anzahl genutzter Umlegungen: 1369
Zeit (ohne Ausgabe der Umlegungen): 62,6 s

14.7 Eigene Beispiele

Die folgenden drei Beispiele zeigen, dass das Programm auch fuer Eingaben funktioniert, bei welchen die Ausgabe ebenfalls die Eingabe ist.

1. Beispiel
Eingabe:
Zahl:

FFFFFFFFFFFFFFFFFFFFFFF

Maximale Anzahl an Umlegungen: 420
Ausgabe:

FFFFFFFFFFFFFFFFFFFFFFF

Anzahl genutzter Umlegungen: 0

2. Beispiel
Eingabe:
Zahl:

32AA438430CCFF327DE6F78A

Maximale Anzahl an Umlegungen: 0
Ausgabe:

32AA438430CCFF327DE6F78A

Anzahl genutzter Umlegungen: 0

3. Beispiel
Eingabe:
Zahl:

88888888888888888888888888888888

Maximale Anzahl an Umlegungen: 187
Ausgabe:

88888888888888888888888888888888

Anzahl genutzter Umlegungen: 0

15 Quellcode

Nun folgen die wichtigsten Bestandteile des Quellcodes.

15.1 Methode zum Erhalten der zu veraendernden Segmente

Diese private Methode ist Teil einer Klasse und erhaelt zwei Hex-Zahlen. Sie soll zwei Listen „remove“ und „add“, welche Attribute der Klasse sind, befuellen mit den Indexen der Segmente, welche aktiviert bzw. deaktiviert werden muessen, um von der ersten Zahl zur zweiten zu kommen.

```
/**
 * Add indexes of segment that have to be removed or added, to get from digit to goal,
 * into their lists
 * @param digit Start digit
 * @param goal Goal digit
 */
private void findChanges(HexDigit digit, HexDigit goal) {
    // Numbers' segments
    boolean[] digitSegments = digit.getSegments(), goalSegments = goal.getSegments();

    // Get segments indexes: stay/remove/add
    for (int i = 0; i < SevenSegmentDisplay.SEGMENT_AMOUNT; i++) {
        boolean numberSegment = digitSegments[i], goalSegment = goalSegments[i];

        // Segment has to be removed because it's enabled in number but not in goal
        if (numberSegment && !goalSegment) {
            this.remove.add(i);
        }
        // Segment has to be enabled/added because it's disabled in number but enabled in goal
        else if (!numberSegment && goalSegment) {
            this.add.add(i);
        }

        // If segment already is on right position, it can be ignored
    }
}
```

15.2 Loesung mit Dynamischer Programmierung

Diese Methode ist die Umsetzung des Algorithmus zum Finden der hoechst moeglichen Zahl durch Tiefensuche mit Dynamischer Programmierung.

```
/**
 * Implementation of solving.
 *
 * @param index The current index in digits
 * @param A     The currently used amount of adds (segment which will be activated)
 * @param R     The currently used amount of removes (segment which will be deactivated)
 * @return An array of the biggest hex-number as hex-digits from given list (digits; constructor)
 * starting at index or null if there is no
```

```
     */
    private HexDigit[] solveImp(int index, int A, int R) {
        // Check in memo
        if (this.isInMemo(index, A, R)) {
            return this.getFromMemo(index, A, R);
        }

        int AR = A + R; // Used twice

        // A + R > 2*m
        // Max amount of changes is overstepped, so returning null
        if (AR > this.doubledMaxChanges) {
            return null;
        }

        // index >= n
        // Index is bigger than or equal size of digits;
        // index was out of range
        if (index >= this.n) {
            // Check A = R, meaning a's can be swapped with r's
            // s.t. solution is valid
            if (A != R) return null;
            // A == R given and A <= m and R <= m given by previous condition

            // Return empty array
            return new HexDigit[0];
        }

        // A+R = 2*m
        // Amount of changes exactly reached
        if (AR == this.doubledMaxChanges) {
            // If dif is not zero, meaning no solution, return null
            if (A != R) return null; // Check A == R

            // Return digits from index (inclusive) to end (n, exclusive)
            return Arrays.copyOfRange(this.digits, index, this.n);
        }

        HexDigit current = this.digits[index];
        HexDigit[] result = null; // Default result is null

        int nextIndex = index + 1;

        // Go through all hex-digits in decreasing order
        for (HexDigit digit : HexDigitUtils.getHexDigitsDecreasing()) {
            // a and r in int array obtained HexDigitChanges object
            int[] values = this.hexDigitChanges.getChanges(current, digit).getValues();
            // Amount of segments to add (a) / remove (r) to get from current to digit
            int a = values[0], r = values[1];

            // Recursive part to get the best solution for next index with new A and R
            HexDigit[] subResult = this.solveImp(nextIndex, A + a, R + r);

            if (subResult != null) {
                int newSize = this.n - index;
                result = new HexDigit[newSize]; // Set result to new HexDigit Array

                // Fill result with digit on first index and subResult on the others
                result[0] = digit;
                System.arraycopy(subResult, 0, result, 1, newSize - 1);

                // Break loop because best result was found
                break;
            }
        }

        // Put result in memo (might be null)
        this.putInMemo(index, A, R, result);

        // Return the found result (might be null)
        return result;
```

```
80      }
```

15.3 Implementierung des Memoisation Objekts

Die folgenden Methoden, welche bereits im vorherigen Quellcode benutzt wurden, sind die unter Implementierung beschriebenen Vorgehensweisen, um bereits berechnete Rueckgabewerte des Hauptalgorithmus (welche zuvor beschrieben wurden) zu speichern und zu erhalten.

Der Typ der Variable memo ist dieser:

```
Map<Integer, Map<Integer, Map<Integer, HexDigit[]>>>
```

Initialisiert wird die Variable als HashMap:

```
1 this.memo = new HashMap<>();
```

Die ebenfalls unter Implementierung beschriebene Vorbefuellung des Memoisation Objekts sieht wie folgt aus:

```
1     // Prefill memo on indexes
          for (int index = 0; index <= this.n; index++) {
3             this.memo.put(index, new HashMap<>());
          }
```

Nun zu den bereits benutzten Methoden.

isInMemo-Methode

```
  /**
2      * Methode returning if given solveImp input was already saved/is contained in memo
       *
4      * @return If given solveImp input was already saved/is contained in memo
       */
6      private boolean isInMemo(int index, int A, int R) {
           Map<Integer, Map<Integer, HexDigit[]>> subMap = this.memo.get(index);
8          if (!subMap.containsKey(A)) return false;

10         Map<Integer, HexDigit[]> subSubMap = subMap.get(A);
           return subSubMap.containsKey(R);
12     }
```

getFromMemo-Methode

```
  /**
2      * Methode returning result to given solveImp input from memo
       *
4      * @return Result to given solveImp input
       */
6     private HexDigit[] getFromMemo(int index, int A, int R) {
           Map<Integer, Map<Integer, HexDigit[]>> subMap = this.memo.get(index);
8          Map<Integer, HexDigit[]> subSubMap = subMap.get(A);
           return subSubMap.get(R);
10     }
```

putInMemo-Methode

```
/**
   * Methode for putting given solveImp input in memo with given result
   */
  private void putInMemo(int index, int A, int R, HexDigit[] result) {
      // Get subMap (will exist because of prefilling in constructor) by index
      Map<Integer, Map<Integer, HexDigit[]>> subMap = this.memo.get(index);

      // Get subSubMap by A:
      // If subMap already contains A, get it from subMap.
      // Otherwise, create new HashMap and put it into subMap
      Map<Integer, HexDigit[]> subSubMap;

      if (subMap.containsKey(A)) {
          subSubMap = subMap.get(A);
      } else {
          subSubMap = new HashMap<>();
          subMap.put(A, subSubMap);
      }

      // Put result into subSubMap
      subSubMap.put(R, result);
  }
```

15.4 Berechnung der Umlegungen

Die Berechnung der Umlegungen geschieht in einer Klasse, welche die Berechnungen im Konstruktor umsetzt und eine oeffentliche Methode besitzt, um eine erzeugte Liste an Taeuschen (Swaps) zu erhalten. Im Konstruktor wird zuneachst die Methode loadEnableDisable und anschlieszend die Methode fillSwaps aufgerufen.

loadEnableDisable-Methode
Diese Methode soll alle die Segmente jeder Ziffer finden, welche aktiviert und deaktiviert werden muessen. Diese Indexe werden in zwei HashMaps („enable“ und „disable“) gespeichert.

```
/**
   * Methode for filling of enable and disable map
   */
  private void loadEnableDisable() {
      // Get all segments that have to be enabled/activated and deactivated/disabled
      // by looping through all digits
      for (int i = 0; i < this.size; i++) {
          // Get changes needed to get from given at i to result at i
          DigitChanges changes = HexDigitChanges.getSingleton()
                  .getChanges(this.given[i], this.result[i]);

          // Lists of segments that have to be enabled/disabled by index
          Deque<Integer> adds = changes.getAdd(ArrayDeque::new), removes = changes
                                                  .getRemove(ArrayDeque::new);

          // Add arrays to their maps
          this.enable.put(i, adds);
          this.disable.put(i, removes);
      }
  }
```

fillSwaps-Methode und Untermethoden
Die Methode fillSwaps ruft diverse Untermethoden auf welche alle im folgenden Codeblock enthalten sind.
Nach Aufruf der Methode soll die Liste an Taeuschen („swaps“) vollkommen befuellt sein.

```
/**
   * Methode for filling the swaps list containing the swaps from given to result number
   */
  private void fillSwaps() {
```

```
        // 1. Only swap inside digits
        this.swapInside();

        // 2. Swap each enable with next disable (they are same size)
        this.swapOutside();
    }

    /**
     * Swap adds with removes in all numbers (not just in one at a time)
     */
    private void swapOutside() {
        Deque<Integer> enableDeque = new ArrayDeque<>(), disableDeque = new ArrayDeque<>();
        int enableIndex = -1, disableIndex = -1;

        boolean isDone = false;
        while (!isDone) {
            // Set enable Deque to next non-empty Deque in enable map if needed and possible (by index)
            while (enableDeque.isEmpty() && enableIndex < this.size - 1) {
                enableIndex++;
                enableDeque = this.enable.get(enableIndex);
            }

            // Set disable Deque to next non-empty Deque in disable map if needed
            // and possible (by index)
            while (disableDeque.isEmpty() && disableIndex < this.size - 1) {
                disableIndex++;
                disableDeque = this.disable.get(disableIndex);
            }

            // Check Deques not being updated, meaning no changes left
            // They should have the same amount of elements, so just testing one
            if (enableDeque.isEmpty()) {
                isDone = true;
            } else {
                int enableSegment = enableDeque.pop();
                int disabledSegment = disableDeque.pop();

                this.addSwap(enableIndex, enableSegment, disableIndex, disabledSegment);
            }
        }
    }

    /**
     * Swap adds with removes until one list is empty inside of all digits
     */
    private void swapInside() {
        for (int i = 0; i < this.size; i++)
            this.swapInsideDigit(i);
    }

    /**
     * Swap adds with removes until one list is empty at index
     * @param index Current index
     */
    private void swapInsideDigit(int index) {
        // Get which segments have to be enabled/disabled
        Deque<Integer> enables = this.enable.get(index);
        Deque<Integer> disables = this.disable.get(index);

        // Swap inside the digit until one list is empty
        while (!enables.isEmpty() && !disables.isEmpty()) {
            int enableSegment = enables.pop();
            int disableSegment = disables.pop();

            this.addSwap(enableSegment, disableSegment, index);
        }
    }

    /**
     * Methode for saving swap of two given segments at given (different!) indexes in number
     * into changingRow
     *
     * @param indexA   Index of first display
```

```
78       * @param segmentA Index of segment in first display
         * @param indexB   Index of second display
80       * @param segmentB Index of segment in second display
         */
82      private void addSwap(int indexA, int segmentA, int indexB, int segmentB) {
            Swap swap = new Swap(indexA, segmentA, indexB, segmentB);
84          this.swaps.add(swap);
        }
86
        /**
88       * Methode for saving swap of two given segments at given index in number into changingRow;
         * here the segments are in the same digit
90       *
         * @param segmentA Index of fist segment
92       * @param segmentB Index of second segment
         * @param index    Display's index
94       */
        private void addSwap(int segmentA, int segmentB, int index) {
96          Swap swap = new Swap(index, segmentA, index, segmentB);
            this.swaps.add(swap);
98      }
```