Aufgabe 3: Hex-Max

Teilname-ID: 62454

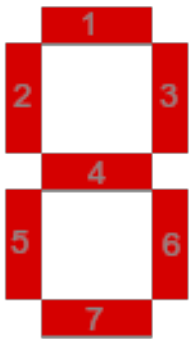
Bearbeiter dieser Aufgabe:
Philip Gilde

19. April 2022

Inhaltsverzeichnis

1 Lösungsidee

Zunächst sollten die Dynamiken einer Transformation, also einer Folge von Verschiebungen, bei der aus einer Hexzahl eine andere wird, betrachtet werden, unabhängig davon, wie groß die sich ergebende Zahl ist. Im Folgenden markiert das Apostroph $'$ eine Zahl nach der Transformation. Eine einzelne Ziffer lässt sich als Liste von 7 booleschen Werte darstellen: Für jedes Segment ein Wert, der aussagt, ob ein Stäbchen vorhanden ist oder nicht. Die Reihenfolge wird folgendermaßen definiert:



Wenn man nun eine Ziffer Z in eine andere Ziffer Z' umlegen möchte, ist das nur möglich, wenn die Ausgangs- und Zielziffer gleich viele Stäbchen haben, also gleich viele Werte wahr sind. Das ist die erste Bedingung ($C(X)$ ist die Anzahl wahrer Werte in der Liste X):

$$C(Z) = C(Z') \quad | - C(Z') \tag{1}$$

$$\leftrightarrow \quad C(Z) - C(Z') = 0 \tag{2}$$

$$D(Z, Z') \coloneqq C(Z) - C(Z') \tag{3}$$

$$D(Z, Z') = 0 \tag{4}$$

$$\tag{5}$$

Die Anzahl der verschobenen Stäbchen lässt sich finden, in dem man für jedes Stäbchen von Z prüft, ob dieses auch bei Z' vorhanden ist. Ist dies nicht der Fall, muss dieses Stäbchen umgelegt werden. Die Anzahl $R(Z, Z')$ an zu verschiebenden Stäbchen für die beiden Ziffern Z und Z' darf nicht größer als m sein. Das ist die zweite Bedingung:

$$R(Z, Z') \coloneqq C(Z \wedge \neg Z') \tag{6}$$
$$R(Z, Z') \leq m \tag{7}$$

Eine mehrstellige Zahl setzt sich aus den l Ziffern $Z_0, Z_1, Z_2, ... Z_{l-1}, Z_l$ vor der Transformation beziehungsweise $Z'_0, Z'_1, Z'_2, ... Z'_{l-1}, Z'_l$ nach der Transformation zusammen. Die Konkatenation dieser Ziffern wird im folgenden als Q beziehungsweise Q' bezeichnet. Die beiden Bedingungen gelten genauso für Q und Q' anstelle von Z und Z'.
Die Anzahl an wahren Werten einer Konkatenation ist gleich der Summe der Anzahlen an wahren Werten der konkatenierten Listen. Wenn beispielsweise eine Liste 5 wahre Werte enthält und eine andere 3, wird die Konkatenation dieser 8 wahre Werte enthalten ($A \circ V$ steht für die Konkatenation der Listen A und B):

$$C(A \circ B) = C(A) + C(B) \tag{8}$$

Wenn also gilt:

$$C(Q) = C(Q') \tag{9}$$

Dann gilt auch, weil Q ja die Konkatenation von Z ist:

$$\sum_{n=0}^{l} C(Z_n) = \sum_{n=0}^{l} C(Z'_n) \qquad | - \sum_{n}^{l} C(Z'_n) \tag{10}$$
$$\leftrightarrow \qquad \sum_{n=0}^{l} (C(Z_n) - C(Z'_n)) = 0 \tag{11}$$
$$\sum_{n=0}^{l} D(Z_n, Z'_n) = 0 \tag{12}$$

Eine einzelne Ziffer Z_k lässt sich abkapseln:

$$\sum_{n=0}^{l} D(Z_n, Z'_n) = 0 \qquad | - D(Z_k, Z'_k)) \tag{13}$$
$$\leftrightarrow \qquad \sum_{n=0|n\neq k}^{l} D(Z_n, Z'_n) = -D(Z_k, Z'_k) \tag{14}$$

So lässt sich die erste Bedingung zu einer Ziffer hin umstellen. Dadurch kann man eine einzelne Ziffer transformieren und erhält daraus eine Bedingung für die restlichen Ziffern.
Bei der zweiten Bedingung verhält es sich genauso:

$$R(Q, Q') \leq m \tag{15}$$
$$\sum_{n=0}^{l} R(Z_n, Z'_n) \leq m \qquad | - R(Z_k, Z'_k) \tag{16}$$
$$\leftrightarrow \qquad \sum_{n=0|n\neq k}^{l} R(Z_n, Z'_n) \leq m - R(Z_k, Z'_k) \tag{17}$$

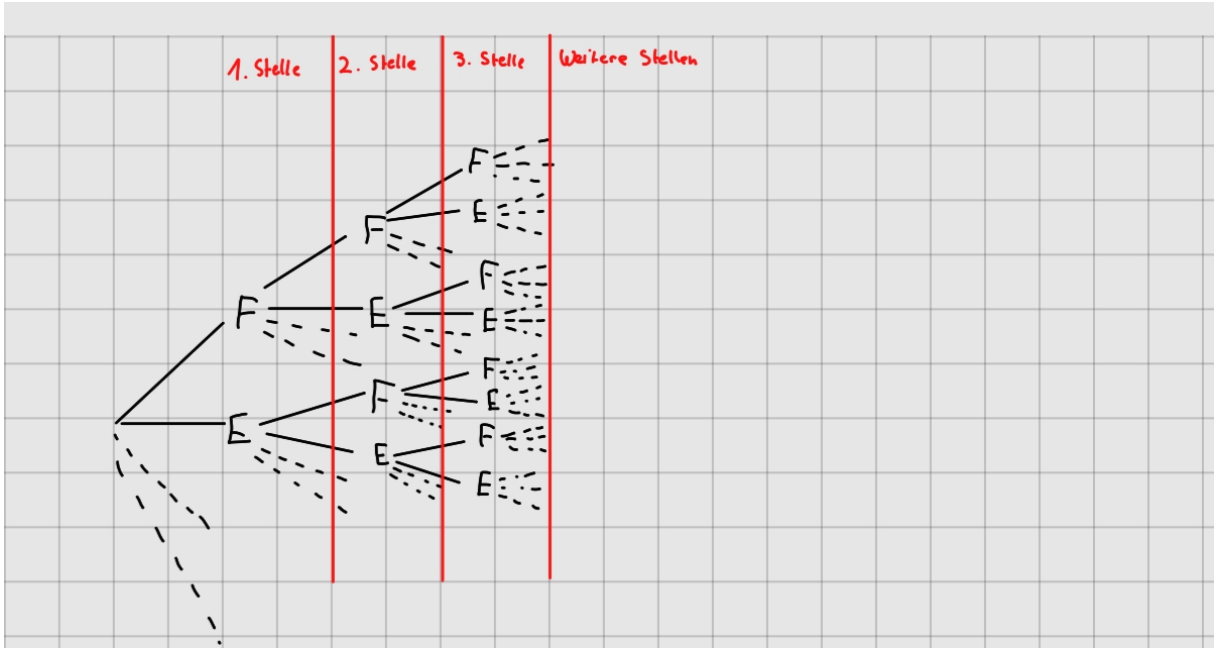
Auf diese Weise lässt sich eine rekursive Suche nach der höchstmöglichen Zahl, die die Bedingungen erfüllt, konstruieren. Dabei wird zuerst die ganz links stehende Ziffer auf den höchsten Wert gesetzt und für die restlichen Ziffern eine möglichst hohe Zahl gesucht, mit der die Bedingungen erfüllt wird. Es wird also mit den restlichen Ziffern genauso verfahren bis zur letzten Ziffer. Die von links erste Ziffer Z'_0 wird

auf den höchstmöglichen Wert F gesetzt. Für die restlichen Ziffern gilt dann

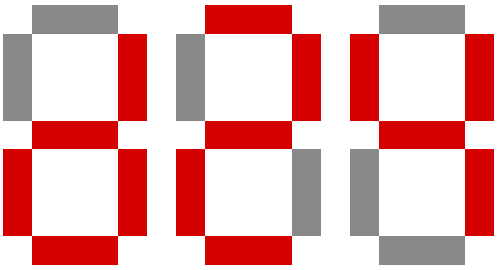
$$\sum_{n\neq 0}^{l} D(Z_n, Z'_n) = -D(Z_0, Z'_0) \tag{18}$$

$$\sum_{n\neq 0}^{l} R(Z_n, Z'_n) \leq m - R(Z_0, Z'_0) \tag{19}$$

Wenn bei der zweiten Bedingung $m - R(Z_0, Z'_0)$ kleiner als 0 ist, würden durch die Transformation mehr Stäbchen bewegt als erlaubt ist, weil daraus folgen würde, dass $m < R(Z_0, Z'_0)$. Es wird dann mit dem nächst kleineren Wert für die Stelle weiter gesucht. Wenn $m - R(Z_0, Z'_0)$ gleich 0 ist, wurden schon so viele Stäbchen weggenommen, wie erlaubt ist. Wenn die Bedingung (18) so schon erfüllt ist, wurden auch so viele Stäbchen an anderen Stellen hingelegt. Wenn nicht, müssen diese Stäbchen noch weiter hinten hingelegt werden. Im ersten Fall wird die Suche beendet, da die so gebildete Zahl die höchstmögliche ist. Sie ist die höchstmögliche, weil die signifikantesten Stellen auf die höchstmöglichen Werte gesetzt wurden und die restlichen nicht mehr änderbar sind. Im zweiten wird für die restlichen Ziffern mit den veränderten Bedingungen (18) und (19) weiter gesucht.
Wenn $m - R(Z_0, Z'_0)$ größer als Null ist, wird mit den restlichen Ziffern der gleiche Prozess durchgeführt, um die höchstmögliche Zahl zu finden. Sie muss die Bedingungen (18) und (19) erfüllen.
Wenn sich mit den restlichen Ziffern die Bedingung erfüllende Zahlen finden lassen, ist die höchste dieser in Verbindung mit Z'_0 die höchstmögliche Zahl. Wenn nicht, wird die erste Ziffer um 1 reduziert und dasselbe wiederholt. Es ist nicht möglich, dass die Zahl kleiner wird, als sie vorher war, weil eine Transformation einer Zahl zu sich selbst zwangsläufig beide Kriterien erfüllt.
Das Verfahren lässt sich mit einem Suchbaum modellieren:



Für die erste Stelle gibt es 16 Äste für jede mögliche Ziffer. In der Zeichnung wurden nur die obersten 2 dargestellt, der Rest soll mit den gestrichelten Linien angedeutet werden. Von jedem dieser 16 Äste gehen jeweils 16 weitere Äste ab, für jede mögliche zweite Ziffer (nicht eingezeichnete Äste sind wieder durch gestrichelte Linien angedeutet). So geht es immer weiter bis zur letzten Ziffer. Das Verfahren geht dabei erst den obersten Ast entlang, dann den (am Ende) zweithöchsten und so weiter. Dabei kann die Suche abgebrochen werden, sobald eine passende Ziffer gefunden wurde, weil ja alle weiteren Ziffern kleiner sind beziehungsweise alle größeren schon durchsucht wurden, ohne die Bedingungen zu erfüllen.
Das Verfahren soll an der Beispieleingabe `D24` mit $m = 3$ illustriert werden. Die Zahl sieht im 7-Segment-Display so aus:



Es wird die erste Ziffer von D auf F gesetzt. Dabei werden 3 Stäbchen von ihrer aktuellen Position weggenommen: $R(Z, Z') = 3$. Die Differenz an Stäbchen nach der Transformation liegt bei -1, es bleibt also ein Stäbchen übrig. Für die beiden restlichen Ziffern dürfen also keine Stäbchen von ihrer Position vor der Transformation weggenommen werden. Die Differenz muss dagegen bei 1 liegen, damit die Summe 0 ist. Es muss also ein Stäbchen aufgebraucht werden, damit das bei der ersten Ziffer übrig gebliebene einen Platz findet. Mit diesen beiden Bedingungen wird nun also die größtmögliche Transformation der übrigen Ziffern `24` mit dem selben Verfahren gesucht. Die erste Ziffer wird auf F gesetzt, dabei werden allerdings Stäbchen von ihrer Ursprungsposition weggenommen, womit $R(Z, Z') > 3$ wäre, es wird also bei der nächst-kleineren Ziffer weiter gesucht. Bei allen kleineren Ziffern bis zur 8 werden Stäbchen von ihrer Position weggenommen. Bei der 8 werden zwei Stäbchen zugelegt, die Differenz für die einzelne Ziffer ist also 2, die gesamte Differenz beträgt $-1 + 2 = 1$. Damit die Anzahl der Stäbchen sich nicht ändert, muss bei der letzten Ziffer die Differenz -1 sein, es muss also ein Stäbchen weggenommen werden. Da aber kein Stäbchen mehr weggenommen werden darf (sonst wäre $R(Z, Z') > 3$), lässt sich nach Durchprobieren aller keine letzte Ziffer finden, welche die Bedingungen erfüllt. Es wurde keine Lösung gefunden, also wird bei der Ziffer an der vorherigen Stelle weiter probiert. Die nächste Zahl, bei der kein Stäbchen weggenommen wird, ist die 2. Für die zwei ist die Differenz 0, es muss ja nichts umgelegt werden. Bei der letzten Ziffer muss die Differenz 1 sein, also ein Stäbchen hinzu gelegt werden. Es darf aber kein Stäbchen umgelegt werden, da dann $R(Z, Z') > 3$ wäre. Nur durch hinlegen eines Stäbchens lässt sich aus der 4 keine andere Ziffer bilden, es muss also bei der Stelle davor weiter gesucht werden. Dort gibt es keine weiteren Ziffern, zu denen transformiert werden kann, ohne ein Stäbchen irgendwo wegzunehmen. Weil keine Lösung gefunden wurde, wird also bei der ersten Ziffer weiter gesucht. Diese wird auf den nächst-kleineren Wert E gesetzt. Bei dieser Umwandlung werden nur zwei der ursprünglichen Stäbchen weggenommen. Die Differenz der Stäbchenzahlen ist 0. Bei den restlichen Ziffern darf also ein Stäbchen von seiner ursprünglichen Position weggenommen werden, während die Differenz der Stäbchenzahlen 0 sein muss. Wie vorher wird nun die größtmögliche Zahl für die restlichen Ziffern gesucht. Die Ziffer an zweiter Stelle wird auf von 2 auf F gesetzt. Dabei wird allerdings mehr als 1 Stäbchen von seiner ursprünglichen Position entfernt, die Bedingungen sind nicht erfüllt. Bei der Transformation von 4 zu E hingegen muss nur ein Stäbchen von seiner ursprünglichen Position entfernt werden. Damit wurden insgesamt genau $3 = m$ Stäbchen von ihrer ursprünglichen Position entfernt. Die Gesamtdifferenz beträgt 0, es können also bei der letzten Ziffer keine Stäbchen zugelegt werden. Somit ist die Lösung `EE4` gefunden, welche laut Aufgabenblatt richtig ist.
Um die Anzahl von zu durchsuchenden Zahlenfolgen zu reduzieren, können weitere Bedingungen für die Fortsetzung der Suche mit den restlichen Ziffern eingeführt werden. Eine solche Bedingung ist, dass der Überschuss an weggenommenen beziehungsweise hingelegten Stäbchen mit den restlichen Ziffern ausgeglichen werden kann. Wenn die ursprüngliche Zahl beispielsweise 88888 ist, mit $m = 10$, lässt sich keine größere Zahl finden, die durch Umlegung erreicht werden kann, weil jede andere Zahl weniger Stäbchen hätte und somit bei der Transformation Stäbchen übrig blieben. Damit nicht jede Zahlenfolge von FFFFF bis 88888 durchprobiert werden muss, um das zu "bemerken", wird nach jeder Transformation einer einzelnen Ziffer geprüft, ob die Differenz der Stäbchenzahlen der bisherigen Transformation D durch Transformation der restlichen Ziffern kompensiert werden könnte. Dafür werden mit den restlichen Ziffern die beiden Differenzen der Stäbchenzahlen D_{min} bei einer Transformation jeder Ziffer zu einer 8, und D_{max} bei Transformation jeder Ziffer zu einer 1, berechnet. 8 hat mit 7 Stäbchen die größte Zahl an Stäbchen und 1 mit 2 die kleinste.
Wenn $D > -D_{min}$, wurden mehr Stäbchen von den transformierten Ziffern weggenommen, als bei den restlichen Ziffern hingelegt werden könnten. Wenn $D < -D_{max}$, würden für die transformierten Ziffern mehr zusätzliche Stäbchen benötigt, als von den restlichen Ziffern weggenommen werden könnten. Ein Beispiel für den ersten Fall wäre die Transformation von `111` zu `F11`. Ein Beispiel für den zweiten Fall ist das obige mit 88888 als ursprüngliche Zahl.
Eine weitere Möglichkeit, die Anzahl der zu durchsuchenden Zahlenfolgen zu reduzieren, ist zu prüfen,

ob $R(Z', Z) \leq m$ und die Suche nur zu vertiefen, wenn es der Fall ist. Durch das Tauschen der Reihenfolge wird hier nicht gezählt, wie viele Stäbchen von ihrer ursprünglichen Position weggenommen wurden, sondern wie viele Stäbchen in vorherigen Lücken liegen. Bei einer gültigen Transformation haben beide den gleichen Wert, weil jedes weggenommene Stäbchen an einer anderen Stelle hingelegt wird, wo vorher keines war. Da die Transformationen bei der Suche allerdings nicht vollständig sind, kann es sein, dass einer der beiden Werte noch kleiner gleich m ist, während der andere schon größer ist.
Weiterhin kann geprüft werden, ob die Differenz der Stäbchenzahlen noch ausgeglichen werden kann, ohne das m dabei überschritten wird. Wenn also beispielsweise bei einer Teiltransformation, die noch vertieft werden muss, $m = 10$, $R(Z, Z') = 6$ und $D = -5$, dann kann die Differenz nicht mehr kompensiert werden, weil bei den restlichen Ziffern 5 Stäbchen übrig bleiben müssen, aber nicht nur noch 4 Stäbchen weggenommen werden dürfen, damit $R(Z, Z') \leq m$ ist. Wenn also $-D > m - R(Z, Z')$ gilt, kann die Differenz nicht kompensiert werden und es wird nicht vertieft. Umgekehrt darf nicht $D > m - R(Z, Z')$ sein, dann sind schon mehr Stäbchen übrig geblieben, als noch hingelegt werden dürfen.
Zuletzt muss bei einer Vertiefung der Suche diese nicht immer nochmal durchgeführt werden. Wenn beispielsweise die erste Ziffer eine 8 ist, kann es sein, dass diese bei der Suche irgendwann in ein E transformiert wird. Dabei werden zwei Stäbchen von ihrer Ursprungsposition weggenommen und bleiben übrig. Wenn die Vertiefung der Suche ergibt, dass mit dieser Transformation keine passende Anordnung für die restlichen Ziffern gefunden werden kann. Es wird also mit Transformation zu D weitergesucht. Bei dieser werden, genau wie beim E, zwei Stäbchen weggenommen und bleiben übrig. Die Bedingungen für die restlichen Zahlen sind also gleich und es muss nicht nochmal die ganze Suche durchgeführt werden, um zu wissen, dass die Bedingungen nicht erfüllt werden kann.
Wenn Z' gefunden ist, muss die Reihenfolge der Umlegungen ermittelt werden. Dabei muss laut Aufgabenstellung zu jedem Zeitpunkt bei jeder Ziffer mindestens ein Stäbchen liegen. Dafür wird eine zunächst leere Liste von Umlegungen L und der Zwischenzustand Z^* eingeführt, welcher anfangs gleich der Ausgangszahl ist. Nun wird für jedes Segment $Z^*_{j,k}$ von Z^* geprüft (j ist die Zahl der Ziffer und k des Segments in dieser), dort ein Stäbchen liegt, aber nicht bei $Z'_{j,k}$, also $Z^*_{j,k} \wedge \neg Z'_{j,k}$. Wenn ja, muss dieses Stäbchen verlegt werden. Wenn das Stäbchen das einzige Stäbchen seiner Ziffer ist, wird bei der nächsten Ziffer weitergemacht und das Stäbchen wird später verlegt. Wenn nicht, wird bei Z' nach einem Segment gesucht, bei dem ein Stäbchen liegt, an dessen Stelle aber bei Z^* noch keines liegt. Es werden also q, r gesucht, für die gilt $Z'_{q,r} \wedge \neg Z^*_{q,r}$. Diese Verschiebung des Stäbchens bei j, k nach q, r wird in Z^* durchgeführt und an das Ende von L gehangen. $Z^*_{j,k}$ wird weiter durch das veränderte Z^* iteriert. Wenn das letzte Segment erreicht ist, wird wieder von vorne begonnen. Es wird so lange iteriert, bis $Z^* = Z'$. Dann wurden alle Schritte gefunden.

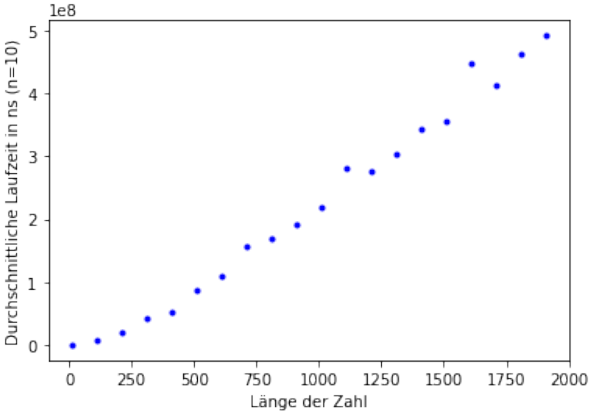
2 Umsetzung

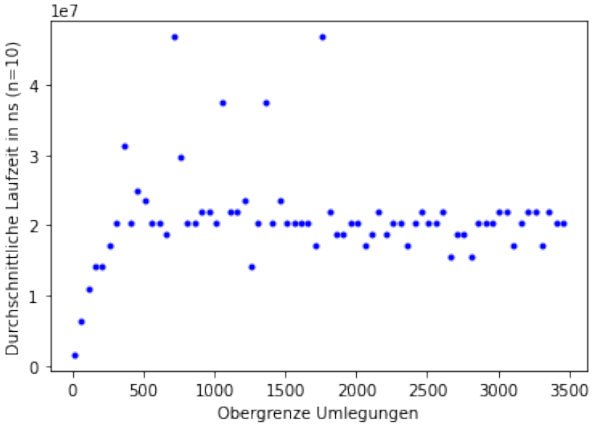
Die Aufgabe wurde in Python implementiert. Dabei wurde die Klasse `HashableArray` eingeführt, die eine oder mehrere Ziffern repräsentiert. Die Klasse ermöglicht die notwendigen elementweisen Operationen *und* und *nicht*, so wie das Zählen der Stäbchen. Diese Operationen werden auch von NumPy-Arrays unterstützt, allerdings sind diese veränderbar und können deshalb nicht gehasht und somit als Schlüssel in einem `dict` verwendet werden. Das ist notwendig, weil der zur Python-Standardbibliothek gehörende Dekorator `functools.lru_cache` sonst nicht verwendet werden kann. Dieser Dekorator speichert die Ergebnisse von Funktionen, damit diese bei gleichen Argumenten nicht nochmal ihre ganze Laufzeit brauchen, sondern das Ergebnis (in der Regel) innerhalb von $\mathcal{O}(1)$ abgerufen werden kann. Damit wird die in der Lösungsidee beschriebene nicht-Durchführung der Vertiefung der Suche bei gleichen Bedingungen implementiert. Weil das Verfahren rekursiv implementiert wurde, musste die Begrenzung der Rekursionstiefe von Python mit `sys.setrecursionlimit(3000)` auf 3000 erhöht werden. Mit dieser Tiefe konnte selbst die Beispieleingabe auf der BwInf-Webseite mit 1000 Ziffern gelöst werden, für höhere Werte wurde das Programm nicht getestet.
Wenn das Programm aufgerufen wird, muss erst der Pfad zur Datei mit der Eingabe angegeben werden, und dann, ob die Zwischenstände ausgegeben werden sollen.

3 Laufzeit

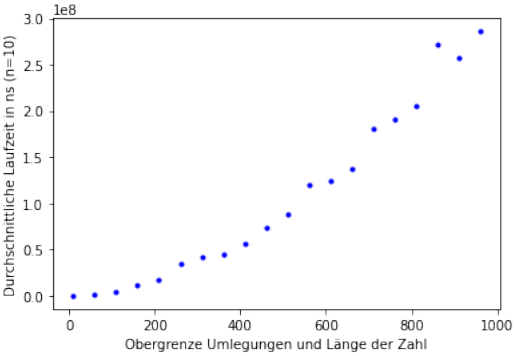
Es wenn die eingegebene Zahl Z die Länge n hat, müssen höchstens $16^n - Z$ verschiedene Zahlen durchprobiert werden. Eine wirkliche Abschätzung der Laufzeit ist schwierig, weil der Einfluss der Bedingungen für die Vertiefung der Suche in komplexem Zusammenhang mit der eingegebenen Zahl steht. Zur besseren

Bestimmung der Laufzeitordnung habe ich auch die Ausführungszeit für zufällig generierte Eingaben mit verschiedenen Parametern untersucht.





Die durchschnittliche Laufzeit ist scheint linear abhängig von der Länge der Zahl (n) bei gleichbleibendem $m = 200$ zu sein (Linkes Diagramm). Die Obergrenze an Umlegungen (m) hat hingegen nur begrenzten Einfluss auf die Laufzeit. Das macht Sinn, denn ab einem bestimmten Wert ist die Obergrenze so hoch, dass innerhalb der Zahl nicht genügend Umlegungen möglich sind, um sie zu erreichen, womit der Einfluss dieser schwindet. Wenn man hingegen gleichzeitig m und n erhöht, hat die Kurve eine eher Runde Form:



Der Speicherbedarf wächst linear zu n, da immer nur die aktuelle Teiltransformation, welche aus höchstens n Ziffern besteht, gespeichert werden muss. Beim Speichern der Ergebnisse von Vertiefungen der Suche kann der Speicherbedarf theoretisch schnell ansteigen, allerdings wurde die Größe des `functools.lru_cache`, welcher dies implementiert, begrenzt. Bei der längsten Beispieleingabe lag der maximale Arbeitsspeicherbedarf bei circa 58 MB. Diesen Wert erachte ich als akzeptabel, weil die meisten aktuellen Computer wenigstens 1 GB Arbeitsspeicher für Programme zur Verfügung stellen.

4 Beispiele

Um Platz zu sparen, wurden die Eingabedateien hier weggelassen, sie sind im Ordner "beispieleßu finden. Die Beispiele sind die gleichen, wie auf der BwInf-Webseite zu finden sind. Bei den Beispieleingaben, bei denen die Zwischenstände ausgegeben werden sollen, sind diese im Ordner `zwischenschritte` zu finden, da das Dokument sonst sehr lang werden würde. Diese stünden an Stelle von `gekürzt` in der Ausgabe.
Beispiel 1:
Eingabe:

```
beispiele/hexmax0.txt
J
```

Ausgabe:

```
Ursprüngliche Zahl:
D24
Höchstmögliche Zahl:
EE4
3 Umlegungen
```

```
Zwischenstände:
[Gekürzt]
```

Beispiel 2:
Eingabe:

```
beispiele/hexmax1.txt
J
```

Ausgabe:

```
Ursprüngliche Zahl:
509C431B55
Höchstmögliche Zahl:
FFFEA91B95
8 Umlegungen
Zwischenstände:
```

Beispiel 3:
Eingabe:

```
beispiele/hexmax2.txt
J
```

Ausgabe:

```
Ursprüngliche Zahl:
632B29B38F11849015A3BCAEE2CDA0BD496919F8
Höchstmögliche Zahl:
FFFFFFFFFFFFFFFFD9A9BEAEE8EDA8BDA989D9F8
37 Umlegungen
Zwischenstände:
[Gekürzt]
```

Beispiel 4:
Eingabe:

```
beispiele/hexmax3.txt
N
```

Ausgabe:

```
Ursprüngliche Zahl:
0E9F1DB46B1E2C081B059EAF198FD491F477CE1CD37EBFB65F8D765055757C6F4796BB8B3DF
7FCAC606DD0627D6B48C17C09
Höchstmögliche Zahl:
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFAA98BB8B9DF
AFEAE888D8888A88888888888
121 Umlegungen
```

Beispiel 5:
Eingabe:

```
beispiele/hexmax4.txt
N
```

Ausgabe:

```
Ursprüngliche Zahl:
1A02B6B50D7489D7708A678593036FA265F2925B21C28B4724DD822038E3B4804192322F230
AB7AF7BDA0A61BA7D4AD8F888
Höchstmögliche Zahl:
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEB8DE88BAA8ADD888898E9BA88AD98988F898
ABAAFABDA8A81BA7D4AD8F888
87 Umlegungen
```

Beispiel 6:
Eingabe:

```
beispiele/hexmax4.txt
N
```

Ausgabe:

```
Ursprüngliche Zahl:
EF50AA77ECAD25F5E11A307B713EAAEC55215E7E640FD263FA529BBB48DC8FAFE14D5B02EBF
792B5CCBBE9FA1330B867E330A6412870DD2BA6ED0DBCAE553115C9A31FF350C5DF99382488
6DB5111A83E773F23AD7FA81A845C11E22C4C45005D192ADE68AA9AA57406EB0E7C9CA13AD0
3888F6ABEDF1475FE9832C66BFDC28964B7022BDD969E5533EA4F2E4EABA75B5DC119728248
96786BD1E4A7A7748FDF1452A5079E0F9E6005F040594185EA03B5A869B109A283797AB3139
4941BFE4D38392AD12186FF6D233585D8C820F197FBA9F6F063A0877A912CCBDCB14BEECBAE
C0ED061CFF60BD517B6879B72B9EFE977A9D3259632C718FBF45156A16576AA7F9A4FAD40AD
8BC87EC569F9C1364A63B1623A5AD559AAF6252052782BF9A46104E443A3932D25AAE8F8C59
F10875FAD3CBD885CE68665F2C826B1E1735EE2FDF0A1965149DF353EE0BE81F3EC133922EF
43EBC09EF755FBD740C8E4D024B033F0E8F3449C94102902E143433262CDA1925A2B7FD01BE
F26CD51A1FC22EDD49623EE9DEB14C138A7A6C47B677F033BDEB849738C3AE5935A2F54B992
37912F2958FDFB82217C175448AA8230FDCB3B3869824A826635B538D47D847D8479A88F350
E24B31787DFD60DE5E260B265829E036BE340FFC0D8C05555E75092226E7D54DEB42E1BB2CA
9661A882FB718E7AA53F1E606
Höchstmögliche Zahl:
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF88EF
A9EBE89EFA99FBDAA8888888888888888888888888888888888888888888888888888888888
888888888888888888888888888888888888888888888888888888888888888888888888888
888888888888888888888888888888888888888888888888888888888888888888888888888
888888888888888888888888888888888888888888888888888888888888888888888888888
8888888888888888888888888
1369 Umlegungen
```

5 Quellcode

```python
import sys
from functools import lru_cache

class HashableArray:
    def __init__(self, vals):
        self.vals = [bool(val) for val in vals]
        self.sum = sum(vals)
        self.inverted = None
        self.hash = None

    def __invert__(self):
        if self.inverted is None:
            self.inverted = HashableArray([not val for val in self.vals])
        return self.inverted
```

```
    @lru_cache(64)
    def __and__(self, other):
        return HashableArray(
            [
                val and other_val
                for val, other_val in zip(self.vals, other.vals)
            ]
        )

    def __hash__(self):
        if self.hash is None:
            self.hash = hash(tuple(self.vals))
        return self.hash

    def __repr__(self):
        return str(self)

    def __str__(self):
        return f"{self.vals}"

    def __eq__(self, other):
        if isinstance(other, HashableArray):
            return all(
                [
                    self_val == other_val
                    for self_val, other_val in zip(self.vals, other.vals)
                ]
            )
        return False

    def __getitem__(self, key):
        return self.vals[key]


# Repräsentationen der 16 Ziffer in aufsteigender Reihenfolge
number_mappings = [
    HashableArray(
        [1, 1, 1, 0, 1, 1, 1],
    ),
    HashableArray(
        [0, 0, 1, 0, 0, 1, 0],
    ),
    HashableArray(
        [1, 0, 1, 1, 1, 0, 1],
    ),
    HashableArray(
        [1, 0, 1, 1, 0, 1, 1],
    ),
    HashableArray(
        [0, 1, 1, 1, 0, 1, 0],
    ),
    HashableArray(
        [1, 1, 0, 1, 0, 1, 1],
    ),
    HashableArray(
        [1, 1, 0, 1, 1, 1, 1],
    ),
    HashableArray(
        [1, 0, 1, 1, 0, 1, 0],
    ),
    HashableArray(
        [1, 1, 1, 1, 1, 1, 1],
    ),
    HashableArray(
        [1, 1, 1, 1, 0, 1, 1],
    ),
    HashableArray(
        [1, 1, 1, 1, 1, 1, 0],
    ),
    HashableArray(
        [0, 1, 0, 1, 1, 1, 1],
    ),
    HashableArray(
```

```
            [1, 1, 0, 0, 1, 0, 1],
        ),
        HashableArray(
            [0, 0, 1, 1, 1, 1, 1],
        ),
        HashableArray(
            [1, 1, 0, 1, 1, 0, 1],
        ),
        HashableArray(
            [1, 1, 0, 1, 1, 0, 0],
        ),
    ]

    # Hexziffern als String in aufsteigender Reihenfolge
    hex_numbers = [
        "0",
        "1",
        "2",
        "3",
        "4",
        "5",
        "6",
        "7",
        "8",
        "9",
        "A",
        "B",
        "C",
        "D",
        "E",
        "F",
    ]

    def sevseg_repr(number):
        result = []
        result.append(
            (u"\u2588" if number[0] or number[1] else "␣")
            + (u"\u2588" if number[0] else "␣")
            + (u"\u2588" if number[0] or number[2] else "␣")
        )
        result.append(
            (u"\u2588" if number[1] else "␣")
            + ("␣")
            + (u"\u2588" if number[2] else "␣")
        )
        result.append(
            (u"\u2588" if number[3] or number[1] or number[4] else "␣")
            + (u"\u2588" if number[3] else "␣")
            + (u"\u2588" if number[3] or number[2] or number[5] else "␣")
        )
        result.append(
            (u"\u2588" if number[4] else "␣")
            + ("␣")
            + (u"\u2588" if number[5] else "␣")
        )
        result.append(
            (u"\u2588" if number[6] or number[4] else "␣")
            + (u"\u2588" if number[6] else "␣")
            + (u"\u2588" if number[6] or number[5] else "␣")
        )
        return result

    def print_sevseg_numbers(hex_numbers):
        digits = [sevseg_repr(number) for number in hex_numbers]
        for i in range(5):
            print("␣".join(digit[i] for digit in digits))

    # verbindet eine liste von digits zu einer einzelnen
    @lru_cache(1024)
```

```
def concat(digits):
    val_list = []
    for digit in digits:
        val_list += digit.vals
    return HashableArray(val_list)


# berechnet die anzahl an umlegungen für eine transformation
@lru_cache(1024)
def n_moves(number_old, number_new):
    return (number_old & ~number_new).sum


# berechnet die höchstmögliche anzahl an stäbchen, die nach einer transformation übrig bleiben
@lru_cache(1024)
def max_difference(digits):
    result = 0
    for digit in digits:
        result += digit.sum - 2
    return result


# berechnet die höchstmögliche anzahl an stäbchen, die für eine transformation zusätzlich benötigt werden
@lru_cache(1024)
def min_difference(digits):
    result = 0
    for digit in digits:
        result += digit.sum - 7
    return result


# difference_sum = anzahl von übrig bleibenden stäbchen
# max_moves = anzahl von stäbchen, die noch wegenommen werden dürfen
# max_moves_rev = anzahl von stäbchen, die noch hingelegt werden dürfen
@lru_cache(200_000)
def find_best(number_old, max_moves, max_moves_rev, difference_sum):
    # Wenn keine Zahl, prüfe ob vorherige Ziffern gültig sind
    if not number_old:
        if difference_sum == 0:
            return tuple(), True
        else:
            return None, False

    # Prüfung, ob Differenz mit maximierung/minimierung der differenz bei rest kompensiert werden kann
    if difference_sum < -max_difference(number_old):
        return None, False
    if difference_sum > -min_difference(number_old):
        return None, False

    # Prüfung, ob differenz mit übrigen umlegungen kompensiert werden kann
    if -difference_sum > max_moves:
        return None, False
    if difference_sum > max_moves_rev:
        return None, False

    # setze erste ziffer auf F, dann E, dann D usw.
    for i in range(16)[::-1]:
        digit_new = number_mappings[i]
        # Gib Zahl zurück, wenn umlegungslimit erreicht ist und differenz null
        if max_moves == n_moves(number_old[0], digit_new):
            if (number_old[0].sum - digit_new.sum) == -difference_sum:
                return (digit_new,) + number_old[1:], True

        # Suche weiter, wenn umlegungslimit nicht überschritten ist (beide richtungen)
        if (max_moves >= n_moves(number_old[0], digit_new)) and (
            max_moves_rev >= n_moves(digit_new, number_old[0])
        ):
            next_best, success = find_best(
                number_old[1:],
                max_moves - n_moves(number_old[0], digit_new),
```

```
                max_moves_rev - n_moves(digit_new, number_old[0]),
                difference_sum + (number_old[0].sum - digit_new.sum),
            )
            # Gib ergebnis zurück, falls vertiefung erfolgreich war
            if success:
                return (digit_new,) + next_best, True
    # Gib miserfolg zurück, wenn keine passende zahl gefunden
    return None, False


# leert alle caches
def clear_caches():
    find_best.cache_clear()
    concat.cache_clear()
    n_moves.cache_clear()
    max_difference.cache_clear()
    min_difference.cache_clear()


# findet einzelne umlegungen, um von einer position zu einer anderen zu kommen.
def single_moves(number_old, number_new):
    number_curr = list(number_old)

    print_sevseg_numbers(number_curr)
    moves = []
    number_new = list(number_new)
    # gehe alle segmente von number_curr durch, bis es gleich number_new ist
    while number_curr != number_new:
        for i in range(len(number_old)):
            for k in range(7):
                # falls bei dem segment ein stäbchen liegt, was im ziel dort nicht liegen soll,
                # suche einen platz für es, außer wenn es das eizige in der ziffer ist
                if (
                    number_curr[i].vals[k]
                    and not number_new[i].vals[k]
                    and number_curr[i].sum != 1
                ):
                    for l in range(len(number_old)):
                        for m in range(7):
                            if (
                                number_new[l].vals[m]
                                and not number_curr[l].vals[m]
                            ):
                                vals = number_curr[i].vals.copy()
                                vals[k] = False
                                number_curr[i] = HashableArray(vals)
                                vals = number_curr[l].vals.copy()
                                vals[m] = True
                                number_curr[l] = HashableArray(vals)
                                moves.append(((i, k), (l, m)))
                                print("->")
                                print_sevseg_numbers(number_curr)
                                break
                        else:
                            continue
                        break
    return moves


def main():
    # Datei einlesen
    with open(input("Pfad: ")) as f:
        number_str = f.readline().strip()
        max_moves = int(f.readline())

    print_steps = input("Schritte ausgeben? (J für ja): ") == "J"
    sys.setrecursionlimit(3000)

    # Eingegebenen String in HashableArray umwandeln
    number_parsed = tuple(
        [number_mappings[int(digit, base=16)] for digit in number_str]
    )
```

```
305
        # höchste umlegungszahl finden
307     best, _ = find_best(number_parsed, max_moves, max_moves, 0)
        number_mappings_list = number_mappings
309     best_remapped = [number_mappings_list.index(digit) for digit in best]
        print()
311     print("Ursprüngliche Zahl:")
        print(number_str)
313     print("Höchstmögliche Zahl:")
        print("".join([hex_numbers[number] for number in best_remapped]))
315     print(f"{n_moves(concat(number_parsed), concat(best))} Umlegungen")

317     if print_steps:
            print("Zwischenstände:")
319         single_moves(number_parsed, best)


321
    if __name__ == "__main__":
323     main()
```

hexmax.py