Aufgabe 3: Hex-Max

Teilnahme-ID: 61099

Bearbeiter/-in dieser Aufgabe:
Malte Voos

23. April 2022

Inhaltsverzeichnis

1 Lösungsidee

Gegeben ist eine Hex-Zahl z_1 mit n Ziffern und eine Maximalzahl an Umlegungen m. Gesucht ist eine Abfolge von maximal m Umlegungen der in der Siebensegmentdarstellung von z_1 enthaltenen Stäbchen, die mit der größtmöglichen auf diese Weise erzeugbaren Hex-Zahl z_2 endet. Dabei gilt die zusätzliche Bedingung, dass bei keiner Umlegung die Siebensegmentdarstellung einer Hex-Ziffer komplett geleert werden darf.

Definition 1. Zwei Hex-Zahlen sind genau dann *verbunden*, wenn ihre Siebensegmentdarstellungen gleich viele Stäbchen beinhalten.

Definition 2. Eine Umlegungs-Abfolge zwischen zwei verbundenen Hex-Zahlen ist genau dann *optimal*, wenn keine andere Umlegungs-Abfolge mit einer geringeren Anzahl an Umlegungen zwischen diesen beiden Hex-Zahlen existiert.

Definition 3. Eine Umlegungs-Abfolge zwischen zwei verbundenen Hex-Zahlen ist genau dann *zulässig*, wenn bei keiner Umlegung die Siebensegmentdarstellung einer Hex-Ziffer komplett geleert wird.

1.1 Zerlegung in Unterprobleme

Zunächst soll gezeigt werden, dass sich das obige Problem in folgende zwei unabhängige Unterprobleme zerlegen lässt:

Unterproblem 1. *Gegeben ist eine Hex-Zahl z_1 und eine natürliche Zahl m. Welche ist die größte verbundene Hex-Zahl z_2, die sich in der Siebensegmentdarstellung in nicht mehr als $2m$ Segmenten von z_1 unterscheidet?*

Unterproblem 2. *Gegeben sind zwei verbundene Hex-Zahlen z_1 und z_2. Was ist eine optimale und zulässige Umlegungs-Abfolge von z_1 zu z_2?*

Die Formulierung von Unterproblem 2 deutet schon folgende Tatsache an:

Lemma 1. *Zwischen jeden zwei verbundenen Hex-Zahlen z_1 und z_2, deren Siebensegmentdarstellungen sich in p Segmenten unterscheiden, existiert eine optimale und zulässige Umlegungs-Abfolge aus $\frac{p}{2}$ Umlegungen.*

Beweis. Sei D_1 die Menge der Segmente, in denen z_1 ein Stäbchen hat und z_2 nicht, und D_2 die Menge der Segmente, in denen z_2 ein Stäbchen hat und z_1 nicht. Da z_1 und z_2 verbunden sind, ist $|D_1| = |D_2| = \frac{p}{2}$. Jeder Isomorphismus zwischen D_1 und D_2 entspricht einer Menge von $\frac{p}{2}$ Umlegungen, bei denen jeweils ein Stäbchen von einem Segment in D_1 zu einem Segment in D_2 umgelegt wird. Alle Umlegungs-Abfolgen, die diese $\frac{p}{2}$ Umlegungen in beliebiger Reihenfolge enthalten, sind optimal, da keine „unnötigen“ Umlegungen stattfinden, d.h., dass Stäbchen niemals „zwischengelagert“ werden und jede Umlegung die beiden beteiligten Segmente in den gewünschten Endzustand bringt.

In einigen Fällen kann es aber vorkommen, dass bei einer Ziffer der Start- und Endwert keine gemeinsamen Stäbchen haben (z.B. bei `1` und `F`) und dass alle Stäbchen der Ziffer in andere Ziffern umgelegt werden, bevor Stäbchen aus anderen Ziffern in die Ziffer zurückgelegt werden. In solch einem Fall wäre die Umlegungs-Abfolge unzulässig, da die Siebensegmentdarstellung einer Ziffer komplett geleert würde. Um dies zu vermeiden, werden zunächst die jeweils in der selben Ziffer liegenden Segmente aus D_1 und D_2 paarweise kombiniert. Übrige Segmente, die auf diese Weise nicht zugeordnet werden konnten, werden beliebig mit Segmenten aus anderen Ziffern kombiniert. Dadurch, dass Umlegungen immer priorisiert innerhalb einer Ziffer stattfinden, ist es nicht möglich, dass die Siebensegmentdarstellung einer Ziffer komplett geleert wird. Bei einem so konstruierten Isomorphismus zwischen D_1 und D_2 ist also jede zugehörige Umlegungs-Abfolge unabhängig von der Reihenfolge der Umlegungen optimal und zulässig. □

Damit kann das Hex-Max-Problem in die Unterprobleme 1 und 2 zerlegt werden: Die Lösung für Unterproblem 1 liefert die größte verbundene Hex-Zahl z_2, die sich in der Siebensegmentdarstellung in nicht mehr als $2m$ Segmenten von z_1 unterscheidet. Dies ist auch die größte Hex-Zahl, die mit maximal m Umlegungen in der Siebensegmentdarstellung von z_1 erzeugt werden kann, da eine möglicherweise größere Hex-Zahl mit mehr als $2m$ unterschiedlichen Segmenten gemäß Lemma 1 zwangsläufig nur mit mehr als $\frac{2m}{2} = m$ Umlegungen erzeugt werden könnte. Die Lösung für Unterproblem 2 liefert dann eine zulässige Umlegungs-Abfolge von z_1 zu z_2, die nach Lemma 1 nicht mehr als $\frac{2m}{2} = m$ Umlegungen enthält.

1.2 Algorithmus für Unterproblem 1

Der Algorithmus für Unterproblem 1 arbeitet im Großen und Ganzen nach dem Backtracking-Verfahren. Ausgehend von z_1 wird rekursiv versucht, die wichtigen (vorderen) Ziffern zu maximieren, indem für für die erste Ziffer absteigend die Werte `F` bis `0` gewählt werden und jeweils versucht wird, die restlichen Ziffern rekursiv auf die gleiche Weise zu maximieren. Dabei gibt es mehrere Backtracking-Kriterien, die früh signalisieren, dass der gewählte Pfad im „Backtracking-Baum“ zu keiner Lösung führt und daher verworfen werden muss.

Diese Backtracking-Kriterien stützen sich auf einige Variablen, welche im Laufe der Rekursion ständig aktualisiert werden:

- $b \in \mathbb{Z}$ ist der „Stäbchen-Kontostand“, welcher bei negativen Werten den Mangel und bei positiven Werten den Überschuss an Stäbchen angibt. Zu Anfang der Rekursion ist $b = 0$. Wenn beispielsweise die erste Ziffer von `7` auf `F` erhöht wird, wird b um 1 herabgesetzt, da die Siebensegmentdarstellung von `F` ein Stäbchen mehr enthält als die von `7` und somit ein Stäbchen „verbraucht“ wurde.

- $l \in \mathbb{N}$ gibt an, wie viele Segmente noch „geflippt“ (d.h. gefüllt oder geleert) werden können, ohne die Maximalzahl an Umlegungen m zu überschreiten. Gemäß Lemma 1 ist l zu Anfang der Rekursion gleich $2m$. Wenn beispielsweise die erste Ziffer von `7` auf `F` erhöht wird, wird l um 5 herabgesetzt, da sich die Siebensegmentdarstellungen von `7` und `F` in 5 Segmenten unterscheiden.

- $v \in \mathbb{N}$ gibt an, wie viele überschüssige Stäbchen theoretisch noch in den hinteren Ziffern untergebracht werden könnten, wenn diese zu `8` (Ziffer bestehend aus den meisten Stäbchen) aufgefüllt würden. Zu Anfang der Rekursion ist v gleich der Gesamtanzahl der leeren Segmente in der Siebensegmentdarstellung der Hex-Zahl. Bei jedem rekursiven Aufruf wird v um die Anzahl der leeren Segmente in der zuvor betrachteten Ziffer herabgesetzt.

- $o \in \mathbb{N}$ ist gewissermaßen das Gegenstück zu v und gibt an, wie viele fehlende Stäbchen theoretisch noch aus den hinteren Ziffern entnommen werden könnten, wenn diese zu `1` (Ziffer bestehend aus den

wenigsten Stäbchen) reduziert würden. Zu Anfang der Rekursion ist o gleich der Gesamtanzahl der Stäbchen in der Siebensegmentdarstellung der Hex-Zahl minus $2n$ (die Ziffer `1` besteht schließlich immer noch aus 2 Stäbchen). Bei jedem rekursiven Aufruf wird v um [die Anzahl der Stäbchen in der zuvor betrachteten Ziffer minus 2] herabgesetzt.

Mit diesen Variablen lassen sich nun folgende Backtracking-Bedingungen aufstellen, welche signalisieren, dass die gewählten ersten Ziffern zu keiner Lösung führen:

- $|b| > l$: Der Überschuss bzw. Mangel an Stäbchen ist so groß, dass er nicht mehr ausgeglichen werden kann, ohne die Maximalzahl an Umlegungen zu überschreiten.
- $b > v$: Der Überschuss an Stäbchen ist so groß, dass er nicht mehr ausgeglichen werden kann, selbst wenn alle übrigen Ziffern zu `8` aufgefüllt werden.
- $-b > o$: Der Mangel an Stäbchen ist so groß, dass er nicht mehr ausgeglichen werden kann, selbst wenn alle übrigen Ziffern zu `1` reduziert werden.

Wenn keine dieser Backtracking-Bedingungen erfüllt ist, wird die erste Ziffer der gegebenen Hex-Zahl abgespalten. Falls dies nicht möglich ist, da bereits alle Ziffern betrachtet wurden, handelt es sich nur um eine gültige Lösung, wenn es keinen Mangel oder Überschuss an Stäbchen gibt ($b = 0$).

Falls noch nicht alle Ziffern betrachtet wurden, werden für die erste Ziffer absteigend die Werte `F` bis `0` angenommen und jeweils die neuen Werte für l und b berechnet. Wenn bei der Berechnung des neuen l ein negativer Wert entstehen würde, d.h. dass schon die Änderung der ersten Ziffer nicht mehr mit den übrigen „Stäbchen-Flips“ zu bewerkstelligen wäre, kann sofort zum nächstniedrigeren Wert für die erste Ziffer übergegangen werden.

Ansonsten hängt das weitere Vorgehen von den neuen Werten für l und b ab:

- $l = 0$; $b = 0$: In diesem Fall sind keine weiteren Umlegungen mehr möglich und es gibt auch keinen Mangel oder Überschuss an Stäbchen. Damit ist die Lösung gefunden; die restlichen Ziffern bleiben unverändert.
- $l = 0$; $b \neq 0$: Es sind keine weiteren Umlegungen mehr möglich, aber es besteht ein Mangel oder Überschuss an Stäbchen. Daher muss zum nächstniedrigeren Wert für die erste Ziffer übergegangen werden.
- $l \neq 0$: Es sind weitere Umlegungen möglich. Mit diesen weiteren Umlegungen wird versucht, die restlichen Ziffern rekursiv zu maximieren. Falls dies gelingt, ist die Lösung gefunden. Anderenfalls wird der nächstniedrigere Wert für die erste Ziffer probiert.

Falls für keinen Wert der ersten Ziffer eine gültige neue Ziffernfolge gefunden werden konnte, wird der Pfad verworfen.

Dieser Algorithmus gibt immer die größtmögliche mit der gegebenen Maximalzahl an Umlegungen erreichbare Hex-Zahl zurück, da stets höhere Ziffernwerte zuerst probiert werden und Pfade nur dann verworfen werden, wenn sie sicher zu keiner Lösung führen.

1.3 Algorithmus für Unterproblem 2

Der Algorithmus für Unterproblem 2 ist im Wesentlichen schon im Beweis für Lemma 1 enthalten. Der Vollständigkeit halber wird er hier nochmal in expliziter Form niedergeschrieben.

1. Die beiden gegebenen verbundenen Hex-Zahlen z_1 und z_2 werden zu ihren jeweiligen Siebensegmentdarstellungen konvertiert.
2. Für jede Ziffer-Siebensegmentanzeige der beiden Siebensegmentdarstellungen:
   a) Die Positionen der Segmente, in denen die Ziffer von z_1 ein Stäbchen hat und die Ziffer von z_2 nicht, und die Positionen derer, in denen die Ziffer von z_2 ein Stäbchen hat und die Ziffer von z_1 nicht, werden in zwei Listen L_1 bzw. L_2 gespeichert.
   b) Es wird, soweit möglich, je ein Segment aus L_1 mit einem aus L_2 kombiniert, die entsprechende Umlegung ausgeführt und der Gesamt-Zwischenstand gespeichert.

   c) Die Positionen übriger Segmente, welche innerhalb der Ziffer keinen „Umlegungs-Partner“ gefunden haben, werden für die spätere Verarbeitung analog zu Schritt a) zu zwei Listen U_1 bzw. U_2 hinzugefügt. Bei U_1 und U_2 handelt es sich für jede Ziffer um die selbe Liste.

3. Analog zu Schritt 2b) wird nun je ein Segment aus U_1 mit einem aus U_2 kombiniert, die entsprechende Umlegung ausgeführt, und der Gesamt-Zwischenstand gespeichert; dieses Mal finden die Umlegungen aber jeweils zwischen verschiedenen Ziffern statt. Da z_1 und z_2 verbunden sind, enthalten U_1 und U_2 hier stets gleich viele Segmente.

4. Die in den vorherigen Schritten gesammelten Zwischenstände definieren nun eine optimale und zulässige Umlegungs-Abfolge von z_1 zu z_2. (siehe Lemma 1)

1.4 Laufzeitanalyse

Der Algorithmus für Unterproblem 1 führt Backtracking über n Variablen mit je 16 möglichen Werten durch. Seine Laufzeit ist daher asymptotisch von oben durch $\mathcal{O}(16^n)$ begrenzt. Hier ist anzumerken, dass diese Grenze in der Regel nicht ausgereizt wird, da viele Pfade durch die in Abschnitt 1.2 beschriebenen Backtracking-Kriterien frühzeitig ausgeschlossen werden können.

Der Algorithmus für Unterproblem 2 iteriert über alle n Ziffern, findet dabei die veränderten Segmente und führt dann k Umlegungen aus, mit $k \leq m$. Die Anzahl der ausgeführten Umlegungen k ist aber auch durch die Anzahl der Ziffern begrenzt: Selbst bei beliebig großem m werden nur so viele Umlegungen ausgeführt, wie erforderlich sind, um die größte verbundene Hex-Zahl zu erreichen. Weitere Umlegungen wären dann nur mit einer höheren Anzahl an Ziffern möglich; es gilt offensichtlich $k \in \mathcal{O}(n)$. Die Laufzeit des Algorithmus für Unterproblem 2 ist also insgesamt in $\mathcal{O}(n + n) = \mathcal{O}(n)$.

Damit ist die Laufzeit des Gesamt-Algorithmus für das Hex-Max-Problem in $\mathcal{O}(16^n + n) = \mathcal{O}(16^n)$.

2 Umsetzung

Das Programm wurde in der Programmiersprache Rust geschrieben. Die Funktion `solve_pt1` implementiert den Algorithmus für Unterproblem 1, die Funktion `solve_pt2` den für Unterproblem 2. Die Funktion `solve_task` entspricht dem Gesamt-Algorithmus für das Hex-Max-Problem: Sie erhält die eingelesene Aufgabe (Datentyp `Task`), ruft `solve_pt1` und `solve_pt2` nacheinander auf, und gibt schließlich die Lösung (Datentyp `Solution`) zurück.

Um Speicherplatz zu sparen, wird eine Siebensegmentanzeige (Datentyp `SevenSegmentDisplay`) als 8-Bit-Integer repräsentiert:

```
#[derive(Clone, Copy)]
struct SevenSegmentDisplay(u8);
```

Die ersten sieben Bits (von rechts) geben jeweils an, ob das entsprechende Segment der Siebensegmentanzeige „an“ ist bzw. ein Stäbchen enthält: wenn ja, ist das Bit eine 1; anderenfalls eine 0. Das letzte Bit ist immer eine 0.

Die erforderlichen Operationen für Siebensegmentanzeigen wurden mithilfe von bitweisen Operationen realisiert:

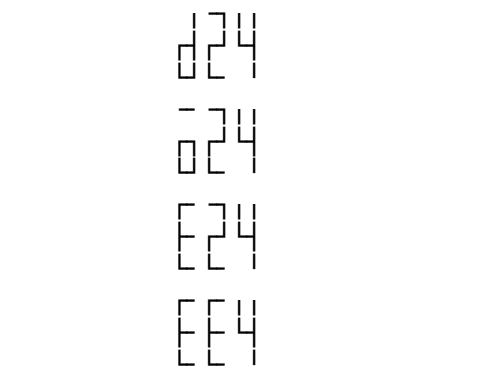
```
impl SevenSegmentDisplay {
    // Gibt das `idx`-te Segment dieser Siebensegmentanzeige zurück.
    fn get(self, idx: u8) -> bool {
        self.0 & (1 << idx) != 0
    }

    // Schaltet das `idx`-te Segmente dieser Siebensegmentanzeige um.
    fn toggle(&mut self, idx: u8) {
        self.0 ^= 1 << idx;
    }
}
```

Abgesehen von dieser implementierungsspezifischen Besonderheit entspricht das Programm ziemlich direkt der in Abschnitt 1 beschriebenen Lösungsidee. Für weitere Details zur Implementierung wird auf den angehängten, ausgiebig kommentierten Quellcode verwiesen.

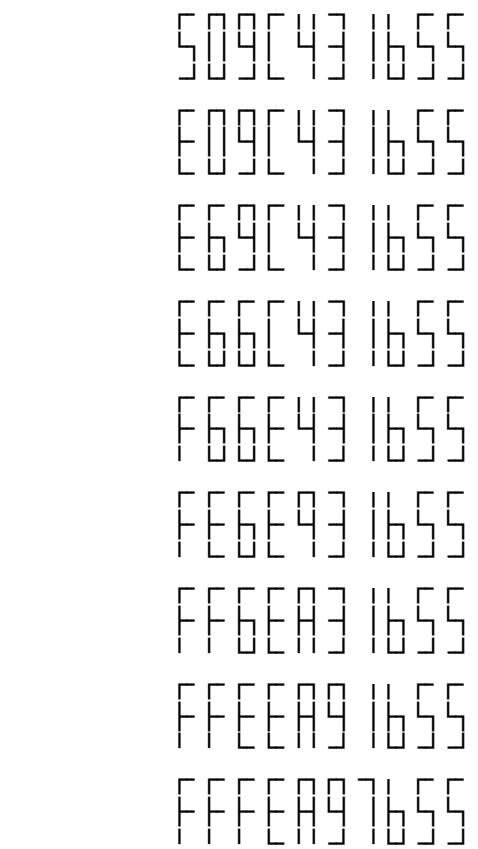
3 Beispiele

Zunächst die Beispiele von der BwInf-Webseite:

```
$ AusführbaresProgramm/aufgabe3-x86_64-linux-static Beispieleingaben/hexmax0.txt
Gegebene Hex-Zahl: D24
```



```
Größtmögliche Hex-Zahl: EE4
Gegebene Maximalzahl an Umlegungen: 3
Anzahl genutzter Umlegungen: 3

Laufzeit ohne I/O: 5.891µs

$ AusführbaresProgramm/aufgabe3-x86_64-linux-static Beispieleingaben/hexmax1.txt
Gegebene Hex-Zahl: 509C431B55
```



```
Größtmögliche Hex-Zahl: FFFEA97B55
Gegebene Maximalzahl an Umlegungen: 8
Anzahl genutzter Umlegungen: 8

Laufzeit ohne I/O: 13.251µs
```

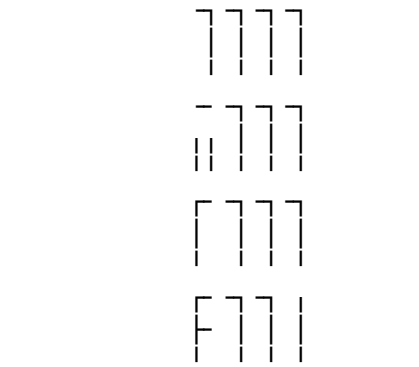
```
$ AusführbaresProgramm/aufgabe3-x86_64-linux-static Beispieleingaben/hexmax2.txt
Gegebene Hex-Zahl: 632B29B38F11849015A3BCAEE2CDA0BD496919F8
```

```
Größtmögliche Hex-Zahl: FFFFFFFFFFFFFFFFD9A9BEAEE8EDA8BDA989D9F8
Gegebene Maximalzahl an Umlegungen: 37
Anzahl genutzter Umlegungen: 37

Laufzeit ohne I/O: 42.484µs

$ AusführbaresProgramm/aufgabe3-x86_64-linux-static Beispieleingaben/hexmax3.txt
Gegebene Hex-Zahl: 0E9F1DB46B1E2C081B059EAF198FD491F477CE1CD37EBFB65F8D765055757C
6F4796BB8B3DF7FCAC606DD0627D6B48C17C09
-- Ausgabe gekürzt --
Größtmögliche Hex-Zahl: FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFAA98BB8B9DFAFEAE888DD888AD8BA8EA8888
Gegebene Maximalzahl an Umlegungen: 121
Anzahl genutzter Umlegungen: 121

Laufzeit ohne I/O: 102.92µs

$ AusführbaresProgramm/aufgabe3-x86_64-linux-static Beispieleingaben/hexmax4.txt
Gegebene Hex-Zahl: 1A02B6B50D7489D7708A678593036FA265F2925B21C28B4724DD822038E3B4
804192322F230AB7AF7BDA0A61BA7D4AD8F888
-- Ausgabe gekürzt --
Größtmögliche Hex-Zahl: FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEB8DE88BAA8ADD8888
98E9BA88AD98988F898AB7AF7BDA8A61BA7D4AD8F888
Gegebene Maximalzahl an Umlegungen: 87
Anzahl genutzter Umlegungen: 87

Laufzeit ohne I/O: 119.441µs

$ AusführbaresProgramm/aufgabe3-x86_64-linux-static Beispieleingaben/hexmax5.txt
Gegebene Hex-Zahl: EF50AA77ECAD25F5E11A307B713EAAEC55215E7E640FD263FA529BBB48DC8F
AFE14D5B02EBF792B5CCBBE9FA1330B867E330A6412870DD2BA6ED0DBCAE553115C9A31FF350C5DF9
93824886DB5111A83E773F23AD7FA81A845C11E22C4C45005D192ADE68AA9AA57406EB0E7C9CA13AD
03888F6ABEDF1475FE9832C66BFDC28964B7022BDD969E5533EA4F2E4EABA75B5DC11972824896786
BD1E4A7A7748FDF1452A5079E0F9E6005F040594185EA03B5A869B109A283797AB31394941BFE4D38
392AD12186FF6D233585D8C820F197FBA9F6F063A0877A912CCBDCB14BEECBAEC0ED061CFF60BD517
B6879B72B9EFE977A9D3259632C718FBF45156A16576AA7F9A4FAD40AD8BC87EC569F9C1364A63B16
23A5AD559AAF6252052782BF9A46104E443A3932D25AAE8F8C59F10875FAD3CBD885CE68665F2C826
B1E1735EE2FDF0A1965149DF353EE0BE81F3EC133922EF43EBC09EF755FBD740C8E4D024B033F0E8F
3449C94102902E143433262CDA1925A2B7FD01BEF26CD51A1FC22EDD49623EE9DEB14C138A7A6C47B
677F033BDEB849738C3AE5935A2F54B99237912F2958FDFB82217C175448AA8230FDCB3B3869824A8
26635B538D47D847D8479A88F350E24B31787DFD60DE5E260B265829E036BE340FFC0D8C05555E750
92226E7D54DEB42E1BB2CA9661A882FB718E7AA53F1E606
-- Ausgabe gekürzt --
Größtmögliche Hex-Zahl: FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF88EFA9EBE89EFA99FBDAA8E8EAD88AB899
F8E8F9AA9E9AD88988EDA9A99888EDAD989A8BAFD8A88888888888888888888888888888888888888
888888888888888888888888888888888888888888888888888888888888888888888888888888888
888888888888888888888888888888888888888888888888888888888888888888888888888888888
8888888888888888888888888888888888888888888888888888
Gegebene Maximalzahl an Umlegungen: 1369
Anzahl genutzter Umlegungen: 1369
```

```
Laufzeit ohne I/O: 2.009263ms
```

Bei den BwInf-Beispielen wurde die Maximalzahl an Umlegungen immer voll ausgeschöpft. Die Beispieleingabe `ungenutzte_umlegungen.txt` deckt den Fall ab, dass am Ende noch ungenutzte Umlegungen verbleiben:

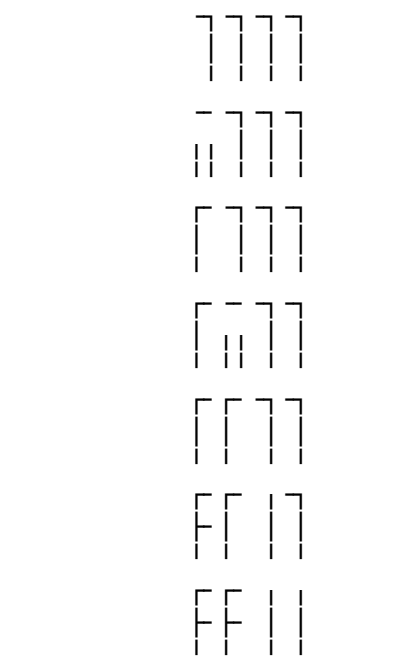
```
$ AusführbaresProgramm/aufgabe3-x86_64-linux-static Beispieleingaben/ungenutzte_umlegungen.txt
Gegebene Hex-Zahl: 7777
```



```
Größtmögliche Hex-Zahl: F771
Gegebene Maximalzahl an Umlegungen: 5
Anzahl genutzter Umlegungen: 3

Laufzeit ohne I/O: 7.52µs
```

Erst bei $m = 6$ kann eine größere Hex-Zahl erreicht werden:

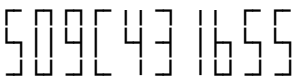
```
$ AusführbaresProgramm/aufgabe3-x86_64-linux-static Beispieleingaben/ungenutzte_umlegungen_bum
Gegebene Hex-Zahl: 7777
```



```
Größtmögliche Hex-Zahl: FF11
Gegebene Maximalzahl an Umlegungen: 6
Anzahl genutzter Umlegungen: 6

Laufzeit ohne I/O: 8.68µs
```

Bei dem Beispiel `keine_umlegungen.txt` ist die Maximalzahl an Umlegungen m gleich 0:

```
$ AusführbaresProgramm/aufgabe3-x86_64-linux-static Beispieleingaben/keine_umlegungen.txt
Gegebene Hex-Zahl: 509C431B55
```



```
Größtmögliche Hex-Zahl: 509C431B55
Gegebene Maximalzahl an Umlegungen: 0
Anzahl genutzter Umlegungen: 0

Laufzeit ohne I/O: 7.691µs
```

Das Beispiel `umlegungen_en_masse.txt` behandelt den umgekehrten Fall, dass weit mehr Umlegungen zur Verfügung stehen, als überhaupt verarbeitet werden können:

```
$ AusführbaresProgramm/aufgabe3-x86_64-linux-static Beispieleingaben/umlegungen_en_masse.txt
Gegebene Hex-Zahl: 509C431B55
```

```
Größtmögliche Hex-Zahl: FFFFFFFE88
Gegebene Maximalzahl an Umlegungen: 1000
Anzahl genutzter Umlegungen: 17

Laufzeit ohne I/O: 13.622µs
```

4 Quellcode

Teile des Quellcodes, die nur der Ein- bzw. Ausgabe dienen, werden hier nicht abgedruckt.

```rust
// Beschreibt eine einzelne Hex-Ziffer. Der enthaltene Integer nimmt nur die
// Werte 0x0 bis 0xF an.
#[derive(Clone, Copy)]
struct HexDigit(u8);

// Beschreibt eine Hex-Zahl mit beliebig vielen Ziffern. Die Ziffern sind in
// Lesereihenfolge gespeichert.
struct HexNumber {
    digits: Vec<HexDigit>,
}

// Beschreibt eine zu lösende Aufgabe.
struct Task {
    number: HexNumber,
    max_moves: usize,
}

// Beschreibt den Zustand einer Siebensegmentanzeige als 8-Bit-Zahl.
//
// Die ersten sieben Bits (von rechts) geben jeweils an, ob das entsprechende
// Segment der Siebensegmentanzeige "an" ist bzw. ein Stäbchen enthält: wenn
// ja, ist das Bit eine 1; anderenfalls eine 0. Das letzte Bit ist immer eine
// 0.
//
// Die Reihenfolge der Segmente entspricht dieser Abbildung:
// https://en.wikipedia.org/wiki/Seven-segment_display#/media/File:7_Segment_Display_with_Labeled_Segments.svg
//
// Allgemein wird der Zustand der Siebensegmentanzeige also durch die Zahl
// 0b0GFEDCBA repräsentiert.
#[derive(Clone, Copy)]
struct SevenSegmentDisplay(u8);

// Beschreibt den Zustand einer Reihe von Siebensegmentanzeigen.
struct SevenSegmentDisplayRow {
    displays: Vec<SevenSegmentDisplay>,
}

// Vom Algorithmus zurückgegebene Lösung.
// Der Typ enthält ein paar redundante Informationen, die aber die Ausgabe des
// Programms anschaulicher machen.
struct Solution {
    // Die gegebene Anfangs-Hex-Zahl.
    initial: HexNumber,
    // Alle Zwischenstände, einschließlich des Anfangs- und Endzustandes.
    states: Vec<SevenSegmentDisplayRow>,
    // Die ermittelte größtmögliche Hex-Zahl.
    r#final: HexNumber,
```

```
    // Die gegebene Maximalzahl an Umlegungen.
    max_moves: usize,
    // Die Anzahl der tatsächlich genutzten Umlegungen.
    used_moves: usize,
}

// Löst eine gegebene Aufgabe, indem Teil 1 und 2 des Algorithmus nacheinander
// aufgerufen werden.
fn solve_task(task: Task) -> Solution {
    let greatest_reachable_number = solve_pt1(&task);
    let states = solve_pt2(&task.number.digits, &greatest_reachable_number);

    Solution {
        initial: task.number,
        r#final: HexNumber {
            digits: greatest_reachable_number,
        },
        max_moves: task.max_moves,
        used_moves: states.len() - 1,
        states,
    }
}

// Implementiert den ersten Teil des Algorithmus, der aus einer Aufgabe die
// größtmögliche Hex-Zahl errechnet.
fn solve_pt1(task: &Task) -> Vec<HexDigit> {
    // `solve_pt1_internal` ist der rekursive Teil der Funktion, der versucht,
    // den gegebenen Ausschnitt der Hex-Zahl zu maximieren.
    fn solve_pt1_internal(
        // Der zu maximierende Ausschnitt der Hex-Zahl.
        digits: &[HexDigit],
        // `segment_balance` enthält im Laufe der Rekursion immer den
        // "Kontostand" der Stäbchen. Wenn es einen Überschuss bzw. Mangel an
        // Stäbchen gibt, ist `segment_balance` positiv bzw. negativ. Nur wenn
        // `segment_balance` am Ende 0 ist, kann die neue Ziffernreihe durch
        // Umlegungen realisiert werden.
        segment_balance: isize,
        // `segment_flips_left` gibt an, wie viele Segmente noch "geflippt"
        // (d.h. gefüllt oder geleert) werden können, ohne die Maximalzahl an
        // Umlegungen zu überschreiten.
        segment_flips_left: usize,
        // `vacant_segments_left` gibt an, wie viele überschüssige Stäbchen
        // theoretisch noch in den hinteren Ziffern untergebracht werden
        // könnten, wenn diese zu '8' (Ziffer bestehend aus den meisten
        // Stäbchen) aufgefüllt würden.
        vacant_segments_left: usize,
        // `occupied_segments_left` gibt an, wie viele fehlende Stäbchen
        // theoretisch noch aus den hinteren Ziffern entnommen werden könnten,
        // wenn diese zu '1' (Ziffer bestehend aus den wenigsten Stäbchen)
        // reduziert werden.
        occupied_segments_left: usize,
    ) -> Option<Vec<HexDigit>> {
        // Es gibt mehrere Backtracking-Bedingungen, die signalisieren, dass
        // dieser Pfad zu keiner gültigen Lösung führen kann.
        if
        // 1. Der Überschuss bzw. Mangel an Stäbchen ist so groß, dass er nicht
        // mehr ausgeglichen werden kann, ohne die Maximalzahl an Umlegungen zu
        // überschreiten.
        segment_balance.abs() as usize > segment_flips_left
            // 2. Der Überschuss an Stäbchen ist so groß, dass er nicht mehr
            // ausgeglichen kann, selbst wenn alle übrigen Ziffern zu '8'
            // aufgefüllt werden.
            || segment_balance > vacant_segments_left as isize
            // 3. Der Mangel an Stäbchen ist so groß, dass er nicht mehr
            // ausgeglichen werden kann, selbst wenn alle übrigen Ziffern zu
            // '1' reduziert werden.
            || -segment_balance > occupied_segments_left as isize
        {
            return None;
        }
        // Es wird immer nur die erste Ziffer betrachtet, und die restlichen
        // Ziffern werden rekursiv ergänzt.
        match digits.split_first() {
```

```
            // Falls schon alle Ziffern betrachtet wurden, handelt es sich nur
            // um eine gültige Lösung, wenn der Stäbchen-Kontostand
            // `segment_balance` gleich 0 ist (s.o.).
            None => match segment_balance {
                0 => Some(Vec::new()),
                _ => None,
            },
            Some((digit, rest)) => {
                // Die Anzahl der Stäbchen, aus der die erste Ziffer besteht.
                let digit_num_sticks = digit.num_sticks();
                // `vacant_segments_left` und `occupied_segments_left` müssen
                // bezogen auf die restlichen Stäbchen angepasst werden.
                // '8', die Ziffer bestehend aus den meisten Stäbchen, enthält
                // 7 Stäbchen.
                let new_vacant_segments_left = vacant_segments_left - (7 - digit_num_sticks);
                // '1', die Ziffer bestehend aus den wenigsten Stäbchen,
                // enthält 2 Stäbchen.
                let new_occupied_segments_left = occupied_segments_left - (digit_num_sticks - 2);

                // Es wird versucht, die erste Ziffer zu maximieren. Dafür
                // werden alle möglichen Hex-Ziffern absteigend durchgegangen
                // und jeweils versucht, die erste Ziffer auf die gewählte
                // Ziffer zu setzen und davon ausgehend die restlichen Ziffern
                // anzupassen, sodass am Ende die gefundene maximale Hex-Zahl
                // durch eine nicht zu hohe Anzahl an Umlegungen erreicht
                // werden kann.
                for candidate_digit in (0x0..=0xF).rev().map(HexDigit) {
                    // `segment_num_difference` ist die Differenz zwischen der
                    // alten und der neuen Anzahl an Stäbchen für die erste
                    // Ziffer.
                    let segment_num_difference =
                        digit_num_sticks as isize - candidate_digit.num_sticks() as isize;
                    // `num_required_segment_flips` ist die erforderliche
                    // Anzahl an "Stäbchen-Flips" (s.o.), um von der alten
                    // ersten Ziffer zur neuen zu kommen.
                    let num_required_segment_flips =
                        digit.num_required_segment_flips(candidate_digit);

                    // Der "Stäbchen-Kontostand" sowie die noch verfügbare
                    // Anzahl an "Stäbchen-Flips" werden bezogen auf die
                    // restlichen Ziffern angepasst.
                    let new_segment_balance = segment_balance + segment_num_difference;
                    let new_segment_flips_left =
                        match segment_flips_left.checked_sub(num_required_segment_flips) {
                            // Wenn schon die Änderung der ersten Ziffer nicht
                            // mehr mit den übrigen "Stäbchen-Flips" zu
                            // bewerkstelligen ist, kann sofort zum
                            // nächstniedrigeren Wert für die erste Ziffer
                            // übergegangen werden.
                            None => continue,
                            Some(x) => x,
                        };

                    match (new_segment_flips_left, new_segment_balance) {
                        // Wenn keine weiteren Umlegungen mehr möglich sind und
                        // es keinen Mangel oder Überschuss an Stäbchen gibt,
                        // ist die Lösung gefunden. Die restlichen Ziffern
                        // bleiben unverändert.
                        (0, 0) => {
                            let mut ret = vec![candidate_digit];
                            ret.append(&mut rest.to_vec());
                            return Some(ret);
                        }
                        // Wenn keine weiteren Umlegungen mehr möglich sind,
                        // aber es einen Mangel oder Überschuss an Stäbchen
                        // gibt, wird der nächstniedrigere Wert für die erste
                        // Ziffer probiert.
                        (0, _) => continue,
                        // Ansonsten wird versucht, die restlichen Ziffern
                        // rekursiv zu maximieren.
                        (_, _) => {
                            let mut new_rest = match solve_pt1_internal(
                                rest,
```

```
                                new_segment_balance,
                                new_segment_flips_left,
                                new_vacant_segments_left,
                                new_occupied_segments_left,
                            ) {
                                // Falls dies nicht gelingt, wird der
                                // nächstniedrigere Wert für die erste Ziffer
                                // probiert.
                                None => continue,
                                Some(x) => x,
                            };
                            let mut ret = vec![candidate_digit];
                            ret.append(&mut new_rest);
                            return Some(ret);
                        }
                    }
                }
                // Der Pfad wird verworfen, da unter den gegebenen Bedingungen
                // für keinen Wert der ersten Ziffer eine gültige neue
                // Ziffernfolge gefunden werden konnte.
                None
            }
        }
    }

    // Die Maximalzahl an "Stäbchen-Flips" ist zweimal die Maximalzahl an
    // Umlegungen, da eine Umlegung aus zwei "Stäbchen-Flips" besteht: an einer
    // Position wird ein Stäbchen entfernt und an einer zweiten ein Stäbchen
    // abgelegt.
    let max_segment_flips = 2 * task.max_moves;
    // Die Startwerte für `vacant_segments_left` und `occupied_segments_left`
    // werden ermittelt, indem einmal über alle Ziffern iteriert wird.
    let (initial_vacant_segments, initial_occupied_segments) = task.number.digits.iter().fold(
        (0, 0),
        |(acc_vacant_segments, acc_occupied_segments), digit| {
            let num_sticks = digit.num_sticks();
            (
                acc_vacant_segments + (7 - num_sticks),
                acc_occupied_segments + (num_sticks - 2),
            )
        },
    );

    solve_pt1_internal(
        &task.number.digits,
        0,
        max_segment_flips,
        initial_vacant_segments,
        initial_occupied_segments,
    )
    .unwrap()
}

// Implementiert den zweiten Teil des Algorithmus, der aus der gegebenen
// Hex-Zahl und der errechneten größtmöglichen Hex-Zahl eine Abfolge von
// Umlegungen erzeugt.
fn solve_pt2(start: &[HexDigit], end: &[HexDigit]) -> Vec<SevenSegmentDisplayRow> {
    // Zunächst werden die beiden Hex-Zahlen jeweils zu einem Array von
    // Siebensegmentanzeigen konvertiert.
    let start_state = start
        .into_iter()
        .map(|digit| digit.to_seven_segments())
        .collect::<Vec<SevenSegmentDisplay>>();
    let end_state = end
        .into_iter()
        .map(|digit| digit.to_seven_segments())
        .collect::<Vec<SevenSegmentDisplay>>();

    // In diesem Array werden die Zwischenstände zwischen den Umlegungen
    // gesammelt. Diese werden am Ende auch zurückgegeben.
    let mut states = vec![SevenSegmentDisplayRow {
        displays: start_state.clone(),
    }];
```

```rust
// Beschreibt die Position eines Segments in einer Reihe von
// Siebensegmentanzeigen.
struct Coordinates {
    display_idx: usize,
    segment_idx: u8,
}

let mut current_state = start_state.clone();

// In diesen beiden Arrays werden die Positionen der Segmente gespeichert,
// die "ein-" bzw. "ausgeschaltet" werden müssen, um von der urspünglichen
// Hex-Zahl zur größtmöglichen zu kommen, und die nicht schon durch eine
// Umlegung innerhalb einer Siebensegmentanzeige richtiggestellt werden
// konnten.
let mut unmatched_on_flips: Vec<Coordinates> = Vec::new();
let mut unmatched_off_flips: Vec<Coordinates> = Vec::new();

for (display_idx, (segments_start, segments_end)) in
    start_state.iter().zip(end_state.iter()).enumerate()
{
    // In diesen beiden Arrays werden die Indizes der Segmente gespeichert,
    // die in dieser Siebensegmentanzeige "ein-" bzw. "ausgeschaltet"
    // werden müssen.
    let mut on_flips = VecDeque::<u8>::new();
    let mut off_flips = VecDeque::<u8>::new();

    for (segment_idx, (segment_start, segment_end)) in segments_start
        .into_iter()
        .zip(segments_end.into_iter())
        .enumerate()
    {
        match (segment_start, segment_end) {
            (false, true) => on_flips.push_back(segment_idx as u8),
            (true, false) => off_flips.push_back(segment_idx as u8),
            _ => {}
        }
    }

    // `num_intra_display_moves` ist die Anzahl der Umlegungen, die
    // innerhalb dieser Siebensegmentanzeige erfolgen können.
    let num_intra_display_moves = on_flips.len().min(off_flips.len());
    // Es werden vorab schon innerhalb dieser Siebensegmentanzeige
    // `num_intra_display_moves` Umlegungen ausgeführt, indem gleichzeitig
    // über die ersten `num_intra_display_moves` "On-Flips" und "Off-Flips"
    // iteriert wird. Nach jeder Umlegung wird eine Kopie des
    // Gesamt-Zwischenstandes gespeichert.
    for (on_flip, off_flip) in on_flips
        .drain(..num_intra_display_moves)
        .zip(off_flips.drain(..num_intra_display_moves))
    {
        current_state[display_idx].toggle(on_flip);
        current_state[display_idx].toggle(off_flip);
        states.push(SevenSegmentDisplayRow {
            displays: current_state.clone(),
        })
    }

    // Übrig gebliebene "Stäbchen-Flips", die nicht durch eine Umlegung
    // innerhalb dieser Siebensegmentanzeige realisiert werden konnten,
    // werden für die spätere Verarbeitung gespeichert.
    let complete_coords = |segment_idx| Coordinates {
        display_idx,
        segment_idx,
    };
    unmatched_on_flips.extend(on_flips.drain(..).map(complete_coords));
    unmatched_off_flips.extend(off_flips.drain(..).map(complete_coords));
}

// Sanity Check
assert_eq!(unmatched_on_flips.len(), unmatched_off_flips.len());

// Zuletzt wird gleichzeiting über die noch nicht realisierten
```

```
    // "Stäbchen-Flips" iteriert. Es wird jeweils die beschriebene Umlegung
    // ausgeführt, und nach jeder Umlegung wird eine Kopie des Zwischenstandes
    // gespeichert, sodass am Ende die genaue Umlegungs-Abfolge einsehbar ist.
    for (on_flip, off_flip) in unmatched_on_flips
        .into_iter()
        .zip(unmatched_off_flips.into_iter())
    {
        current_state[on_flip.display_idx].toggle(on_flip.segment_idx);
        current_state[off_flip.display_idx].toggle(off_flip.segment_idx);
        states.push(SevenSegmentDisplayRow {
            displays: current_state.clone(),
        })
    }

    states
}

impl HexDigit {
    // Stellt eine Hex-Ziffer auf einer Siebensegmentanzeige dar.
    fn to_seven_segments(self) -> SevenSegmentDisplay {
        let segments = match self.0 {
            0x0 => 0b0111111,
            0x1 => 0b0000110,
            0x2 => 0b1011011,
            0x3 => 0b1001111,
            0x4 => 0b1100110,
            0x5 => 0b1101101,
            0x6 => 0b1111101,
            0x7 => 0b0000111,
            0x8 => 0b1111111,
            0x9 => 0b1101111,
            0xA => 0b1110111,
            0xB => 0b1111100,
            0xC => 0b0111001,
            0xD => 0b1011110,
            0xE => 0b1111001,
            0xF => 0b1110001,
            _ => unreachable!(),
        };
        SevenSegmentDisplay(segments)
    }

    // Gibt die Anzahl der Stäbchen zurück, die für die Darstellung dieser
    // Hex-Ziffer auf einer Siebensegmentanzeige benötigt werden.
    fn num_sticks(self) -> usize {
        self.to_seven_segments().0.count_ones() as usize
    }

    // Gibt die Anzahl der "Stäbchen-Flips" zurück, die benötigt werden, um
    // von der Siebensegmentdarstellung dieser Hex-Ziffer zu der einer anderen
    // zu gelangen.
    fn num_required_segment_flips(self, other: Self) -> usize {
        (self.to_seven_segments().0 ^ other.to_seven_segments().0).count_ones() as usize
    }
}

impl SevenSegmentDisplay {
    // Gibt das `idx`-te Segment dieser Siebensegmentanzeige zurück.
    fn get(self, idx: u8) -> bool {
        self.0 & (1 << idx) != 0
    }

    // Schaltet das `idx`-te Segmente dieser Siebensegmentanzeige um.
    fn toggle(&mut self, idx: u8) {
        self.0 ^= 1 << idx;
    }
}

// Iterator über die Segmente einer Siebensegmentanzeige.
struct SevenSegmentDisplayIterator {
    idx: u8,
    display: SevenSegmentDisplay,
}
```

```rust
impl Iterator for SevenSegmentDisplayIterator {
    type Item = bool;

    fn next(&mut self) -> Option<Self::Item> {
        if self.idx <= 7 {
            let bit = self.display.get(self.idx);
            self.idx += 1;
            Some(bit)
        } else {
            None
        }
    }
}

impl IntoIterator for SevenSegmentDisplay {
    type Item = bool;
    type IntoIter = SevenSegmentDisplayIterator;

    fn into_iter(self) -> Self::IntoIter {
        SevenSegmentDisplayIterator {
            idx: 0,
            display: self,
        }
    }
}

// Einstiegspunkt für das Programm.
fn main() {
    let task_file_name = match env::args().nth(1) {
        Some(x) => x,
        None => {
            eprintln!("Nutzung: aufgabe3 <dateiname>");
            process::exit(1);
        }
    };
    let task_str = fs::read_to_string(task_file_name).expect("Datei kann nicht gelesen werden");
    let task = Task::try_from(task_str.as_str()).expect("Datei enthält keine gültige Aufgabe");

    let start = time::Instant::now();
    let solution = solve_task(task);
    let elapsed = start.elapsed();

    println!("{}", solution);
    println!("Laufzeit ohne I/O: {:?}", elapsed);
}
```