Aufgabe 3: Hex-Max

Teilnahme-ID: 60809

Bearbeiter dieser Aufgabe:
Tobias Steinbrecher

24. April 2022

Inhaltsverzeichnis

1 Lösungsidee **2**
1.1 Hexadezimalsystem . . . 2
1.2 Umlegungen . . . 2
1.3 Optionen des Umlegens . . . 3
1.4 Rekursives Vorgehen . . . 4
1.5 Vermeiden von „leeren“ Ziffern . . . 5
1.6 Optimierungen . . . 6
1.7 Erweiterungen . . . 7
1.7.1 Minimierung der Hexadezimalzahl (*HexMin*) . . . 7
1.7.2 Weitere Zahlensystem (z.B. *BinMax*, *DezMax*) . . . 7

2 Umsetzung **7**
2.1 hexInSSD . . . 7
2.2 main() und parseInput() . . . 8
2.3 maximieren() . . . 8
2.4 maxZiffer() . . . 8
2.4.1 Abbruchkriterien . . . 9
2.4.2 Iteration über alle Optionen . . . 9
2.4.3 Umlegen einer Ziffer . . . 9
2.5 Erweiterungen . . . 10
2.5.1 Minimierung der Hexadezimalzahl (*HexMin*) . . . 10
2.5.2 Weitere Zahlensystem (z.B. *BinMax*, *DezMax*) . . . 10

3 Zeitkomplexität **10**
3.1 Formulierung als Entscheidungsproblem . . . 11
3.2 *NP-Zertifikat* . . . 11
3.3 Fomalisierung des Entscheidungsproblems . . . 11
3.4 Ähnlichkeit zum *Mehrfachauswahl-Rucksack-Problem* . . . 12
3.5 Polynomialzeit-Reduktion . . . 12
3.6 Entwicklung der Reallaufzeit . . . 13

4 Beispiele **13**
4.1 Randinformationen . . . 13
4.2 BWINF-Beispiele . . . 14
4.3 Eigene Beispiele . . . 15

5 Quellcode **17**

1 Lösungsidee

1.1 Hexadezimalsystem

Zunächst sollte sich das Hexadezimalsystem genauer angeschaut werden. Die einzelnen Ziffern einer Hexadezimalzahl beschreiben wie in jedem anderen polyadisches Zahlensystem, Potenzen einer bestimmten Basis (hier: $b = 16$). Man weise den n Ziffern von Rechts nach Links Indizes $(0, 1, ..., n-1)$ zu. An der Stelle mit Index i befindet sich nach der Definition, der Wert $a_i \cdot 16^i$, wobei der Koeffizient $a_i \in \mathbb{Z}_{16}$ ($\mathbb{Z}_{16} = \{0, 1, ..., 15\}$:= Ziffernvorrat des Hexadezimalsystems in Dezimalschreibweise). Daraus folgt, dass sich die gesamte Hexadezimalzahl als

$$H = \sum_{i=0}^{n-1} a_i \cdot 16^i \tag{1}$$

ausdrücken lässt.
Es lässt sich beweisen, dass das Erhöhen von $\sum_{i=0}^{j} a_i \cdot 16^i$ immer eine geringere Vergrößerung von H bewirkt, als das Erhöhen von $a_{j+1} \cdot 16^{j+1}$, wenn $a_i, a_{j+1} \in \mathbb{Z}_{16}$.
Der maximale Wert für a_i ist 15, $\forall y \in \mathbb{Z}_{16} : y \leq 15$.
Folgend setze man 15 für a_i ein, um den maximalen Wert von $\sum_{i=0}^{j} a_i \cdot 16^i$ zu erzeugen.
Der minimale Wert für den Term $a_{j+1} \cdot 16^{j+1}$ ist 16^{j+1}, da bewiesen werden soll, welche Auswirkung das Erhöhen von a_{j+1} hat. Somit beträgt a_{j+1} mindestens 1.
Man beweise somit $15 \cdot \sum_{i=0}^{j} 16^i < 16^{j+1}$.

$$\begin{aligned}
15 \cdot \sum_{i=0}^{j} 16^i &< 16^{j+1} \qquad | \quad : 16^j \\
15 \cdot \sum_{i=0}^{j} 16^{i-j} &< 16 \\
(16-1) \cdot (16^{-j} + 16^{1-j} + ... + 16^0) &< 16 \\
(16^{1-j} + 16^{2-j} + ... + 16) - (16^{-j} + 16^{1-j} + ... + 16^0) &< 16 \\
16 - 16^{-j} &< 16 \qquad | \quad -16 \\
-16^{-j} &< 0
\end{aligned} \tag{2}$$

$-16^{-j} < 0$ mit $j \in \mathbb{N}_0$ liefert immer eine wahre Aussage. Somit ist die gewünschte Aussage bewiesen. ■
Da H maximiert werden muss, ist die Idee folglich beim Koeffizienten a_{n-1} anzufangen und diesen mit $\mathbb{Z}_{16}$ zu maximieren, da dieser, wie herausgestellt, die stärkste Vergrößerung von H bewirkt.
Folgend maximiere man $a_{n-2}, a_{n-3}, ..., a_0$. Daraus ergibt sich durch die beschriebene Definition immer die größtmögliche Hexadezimalzahl mit n Ziffern.
Verschiedene Aspekte der Problemstellung erschweren diesen Prozess der Maximierung. Das Prinzip der Umlegungen und die Maximalzahl der Umlegungen m muss beachtet werden. Im folgenden Unterabschnitt 1.2, wird das Prinzip des Umlegens konkretisiert.

1.2 Umlegungen

Die Aufgabenstellung erfordert, dass nicht nur die maximale Hexadezimalzahl, welche mit m Umlegungen erzeugt werden kann gefunden wird, sondern auch, dass die Umlegungen angegeben werden, welche nötig sind um diese zu erreichen.
Sei $\mathbb{U}$ die Menge der Umlegungen, die getätigt werden, um H mit maximal m Umlegungen ($|\mathbb{U}| \leq m$) zu maximieren und somit die gesuchte Lösung des *HexMax*-Problems.
Eine Umlegung $u \in \mathbb{U}$ ist darüber definiert, dass ein Segment von seiner ursprünglichen Position p_u, an eine andere Position p'_u bewegt wird, welche nicht belegt ist.
Die Idee, um $\mathbb{U}$ zu ermitteln und somit das Problem zu lösen ist, rekursiv vorzugehen (siehe Unterabschnitt 1.4).
Dadurch, dass die Ziffern somit erst nach und nach angeschaut werden und für manche Umlegungen p_u und p'_u bei zwei verschiedenen Ziffern liegen (Umlegungen sind ziffernübergreifend möglich), ist der Begriff der *temporären Unbestimmtheit* von p_u bzw. p'_u einzuführen.
Im Folgenden werden p_u oder p'_u *temporär unbestimmt* genannt, wenn diese noch nicht feststehen und somit noch im Prozess der Rekursion ermittelt werden müssen.

Bei der Änderung eines Koeffizienten a_i in eine andere Ziffer b_i, wobei $b_i \in \mathbb{Z}_{16}$, werden jeweils Umlegungen benötigt.
Die Umlegungen dabei sind abhängig von der Darstellung von a_i und b_i in der Sieben-Segment-Anzeige (SSA). Da die Darstellung in der SSA rein optisch definiert ist, wird bei der Implementierung eine eindeutige Übersetzung benötigt (siehe Unterabschnitt 2.1). Im Folgenden wird ein angeschaltetes Segment in der SSA als „vorhanden" und ein ausgeschaltetes Segment als „nicht vorhanden" betitelt.
Die Idee ist, alle Segmente von a_i und b_i zu vergleichen. Beim Vergleichen eines Segmentes s in den Darstellungen von a_i und b_i, lässt sich eine Fallunterscheidung treffen:

1. s ist in der Darstellung von a_i und von b_i vorhanden
   oder s ist in der Darstellung von a_i und von b_i nicht vorhanden.
   ⇒ Es wird keine Umlegung für dieses Segment s gebraucht
2. s ist in der Darstellung von a_i vorhanden und in der Darstellung von b_i nicht.
   ⇒ Das Segment s muss weggelegt werden (Es wird nicht mehr benötigt).
   a) Es wurde bereits eine andere Umlegung u gefunden, für die gilt, dass p_u *temporär unbestimmt* ist
      ⇒ p_u kann auf s festgelegt werden und es wird keine vollständig neue Umlegung hinzugefügt.
   b) Es wurde keine andere Umlegung u gefunden, für die gilt, dass p_u *temporär unbestimmt* ist
      ⇒ Es muss eine vollständig neue Umlegung u hinzugefügt werden, für die gilt, dass p'_u *temporär unbestimmt* ist.
3. s ist in der Darstellung von b_i vorhanden und in der Darstellung von a_i nicht.
   ⇒ Ein Segment s muss an den aktuellen Platz gelegt werden (Es wird benötigt).
   a) Es wurde bereits eine andere Umlegung u gefunden, für die gilt, dass p'_u *temporär unbestimmt* ist
      ⇒ p'_u kann auf s festgelegt werden und es wird keine vollständig neue Umlegung hinzugefügt.
   b) Es wurde noch keine andere Umlegung u gefunden, für die gilt, dass p'_u *temporär unbestimmt* ist
      ⇒ Es muss eine vollständig neue Umlegung u hinzugefügt werden, für die gilt, dass p_u *temporär unbestimmt* ist.

Mit diesem Vorgehen ist es möglich die **minimalen** Umlegungen beim Umlegen einer Ziffer a_i in eine anderen Ziffer b_i zu generieren. Die Idee ist, dass die bereits getätigten Umlegungen ziffernübergreifend zugänglich sind und somit ermöglicht wird, dass für eine Umlegung $u \in \mathbb{U}$, p_u und p'_u auch bei zwei verschiedenen Ziffern liegen können, dadurch, dass beipielsweise Fall 2b) und Fall 3a) beim Umlegen zweier verschiedenen Ziffern eintreten.
Im Folgenden wird die Anzahl an *temporär unbestimmten* p_u bzw. p'_u als Segmentumsatz ΔS bezeichnet. Dieser lässt sich als ein einziger Skalar definieren, da aufgrund der beschriebnen Umlegelogik, niemals *temporär unbestimmte* p_u und p'_u gleichzeitig vorhanden sein können (siehe Unterabschnitt 1.2). Sei $\Delta S > 0$ wenn *temporär unbestimmte* p'_u vorhanden sind und $\Delta S < 0$ wenn *temporär unbestimmte* p_u vorhanden sind. Dabei gilt $|\Delta S| =$ Anzahl der *temporär unbestimmten* Positionen.
Somit agiert ΔS sozusagen als die Anzahl an übrigen Segmenten.
Eine wichtige Eigenschaft der Menge $\mathbb{U}$ ist,

$$\forall u \in \mathbb{U}: \quad p_u \neq \textit{unbestimmt} \land p'_u \neq \textit{unbestimmt} \qquad \text{bzw.} \qquad \Delta S \stackrel{!}{=} 0 \tag{3}$$

da alle Segmente letztendlich untergebracht werden müssen und Segmente nicht erschaffen werden dürfen. Während der Rekursion ist die Eigenschaft der *temporären Unbestimmtheit* von p_u oder p'_u jedoch relevant, solange aber mindestens noch eine Position *unbestimmt* ist, wurde $\mathbb{U}$ noch nicht gefunden.
In Kombination mit $|\mathbb{U}| \leq m$, wurden somit neben der Maximierung, die einzigen Eigenschaften von $\mathbb{U}$ beschrieben. Außerdem wird in der Problemstellung die Einschränkung getroffen, dass die Darstellung von Ziffern niemals komplett „geleert" werden darf (siehe Unterabschnitt 1.5).

1.3 Optionen des Umlegens

Die Optionen $\mathbb{O}_i$, in die ein Koeffizient a_i umgelegt werden kann, entsprechen zunächst dem Ziffernvorrat des Hexadezimalsystems $\mathbb{Z}_{16}$.

Wie in Unterabschnitt 1.1 beschrieben, maximiere man die Hexadezimalzahl H.
Für das Vergrößern von H muss mindestens ein beliebiger Koeffizient a_k vergrößert werden, während alle a_i mit $k < i < n$ unverändert bleiben. Für eine stärkere Vergrößerung werde dabei der Index k möglichst groß. Diese Annahmen lassen sich aus Gleichung 2 ableiten $\Rightarrow \sum_{i=0}^{k-1} a_i \cdot 16^i < a_k \cdot 16^k$.
Als Optionen für a_k kommen also nur die maximierenden Optionen $\mathbb{M}_k \subset \mathbb{O}_k$ infrage, wobei

$$\mathbb{M}_k = \{x \mid x \in \mathbb{O}_k, x > a_k\}$$

und somit alle Optionen zur Vergrößerung des Koeffizienten an der Stelle k bietet. Falls $|\mathbb{M}_k| = 0$ und somit keine Optionen für die Vergrößerung des Koeffizienten a_k vorhanden sind (Beispiel: $a_k = 15$), folgt keine Umlegung von a_k und $k = k - 1$. Somit wird garantiert, dass der erste Koeffizient von links, der vergrößert werden kann, vergrößert wird. Es kann jedoch auch der Fall eintreten, dass kein Koeffizient vergrößert werden kann (Beispiel: $\forall a : a = 15$), in diesem Fall kann dementsprechend auch keine Vergrößerung von H stattfinden ($H = \sum_{i=0}^{n-1} a_i \cdot 16^i$).
Für den Rest der Koeffizienten (a_i mit $0 \leq i < k$), sind dennoch alle Optionen $\mathbb{O}_i$ relevant, da es durch das Prinzip des Umlegens (Unterabschnitt 1.2), Szenarien gibt, in denen a_i verringert werden kann, um a_k zu maximieren, was letztendlich gemäß Gleichung 2 trotzdem für eine Vergrößerung von H sorgt, z.B.: $a_0 = 15(F_{16}) \wedge a_1 = 1(1_{16}) \Rightarrow$ Unabhängig von der Anzahl an Umlegungen m, muss hierbei für eine Vergrößerung von H, zwangsweise a_1 vergrößert werden und a_0 verkleinert werden.
Mit den Mengen $(\mathbb{M}_k \wedge \mathbb{O}_{i<k})$ lassen sich alle Wege für die Vergrößerung von H darstellen. Wenn man die einzelnen Optionen dabei als Umlegungen von a_i in den Wert $b_i \in \mathbb{O}_i$ betrachtet, entspricht ein bestimmte Wahl der Optionen, den optimalen Umlegungen $\mathbb{U}$ und somit der Lösung des Problems.

1.4 Rekursives Vorgehen

Wie bereits angedeutet, ist die Idee, das Problem mittels dynamischer Programmierung, spezifischer Rekursion zu lösen. Das grobe Konzept ähnelt einem Brute-Force-Algorithmus, mit welchem alle Wege um H zu maximieren, von der stärksten bis zur schwächsten Vergrößerung ausprobiert werden und mithilfe der Bedingungen aus Unterabschnitt 1.2:

1. $\Delta S = 0$
2. $|\mathbb{U}| \leq m$

die Validität der verschiedenen rekursiv aufgezählten Mengen überprüft wird. Wenn die Bedingungen das erste Mal zutreffen, wurde die Lösung der optimalen Umlegungen $\mathbb{U}$ gefunden. Dabei ist jedoch wichtig, dass Bedingung 1 erst überprüft werden kann, wenn jede Ziffer vermeintlich maximiert wurde, da wie in Unterabschnitt 1.2 beschrieben, mithilfe der *Unbestimmtheit* von $p_u \wedge p'_u$ zunächst überhaupt Umlegungen und ziffernübergreifende Umlegungen ermöglicht werden.
Das Verfahren lässt sich durch einige Optimierungen (siehe Unterabschnitt 1.6) sehr effizient gestalten.
Die Idee hinter dynamischer Programmierung, ist die gesamte Lösung des Problems (Maximierung von H), auf Basis von kleineren Teilproblemen und Speicherungen zu berechnen. Im Sachzusammenhang könnte man das Subproblem als: „Wie kann ein bestimmter Koeffizient a_i mit maximal m' Umlegungen und mit einem bereitgesetllten Segmentumsatz ΔS, maximiert werden?“ formulieren.
Wie in Unterabschnitt 1.3 beschrieben, muss es mindestens einen Koeffizienten a_k geben, welcher vergrößert wird, damit H vergrößert wird. Ansonsten bleibt H unverändert. Das höchstmögliche k wird wie in Unterabschnitt 1.3 beschrieben ermittelt und dient als Startpunkt der Rekursion/Maximierung.
Da folglich die stärkste Maximierung ausprobiert werden soll, werden zuerst die Umlegungen für a_k in $b_k \in \mathbb{M}_k$, $\forall y \in \mathbb{M}_k : y \leq b_k$ gemäß Unterabschnitt 1.2 generiert. Anschließend wird die maximale Option für die nächsten Koeffizienten a_i $(i < k)$ gewählt.
Sobald Bedingung 2 verletzt wird und somit mehr als m Umlegungen durchgeführt wurden oder Bedingung 1 verletzt ist, wird die nächstgeringere Option des zuletzt umgelegten Koeffizienten a_i gewählt. Dieses Schema wird wiederholt angewandt. Sollte es keine nächstgeringere Option für a_i geben, wählt man die nächstgeringere Option für a_{i+1} bis hin zu a_k. Sollte der Fall eintreten, dass es keine geringere Option für a_k gibt, folgt wie in Unterabschnitt 1.3, $k = k - 1$ und das Verfahren beginnt erneut.
Dabei sorgt, das kontinuierliche Umlegen in die maximalmögliche Ziffer, für das rekursive Aufzählen/Ermitteln der Lösung $\mathbb{U}$.
Um das Vorgehen zu visualisieren, eignet sich folgendes Schaubild:

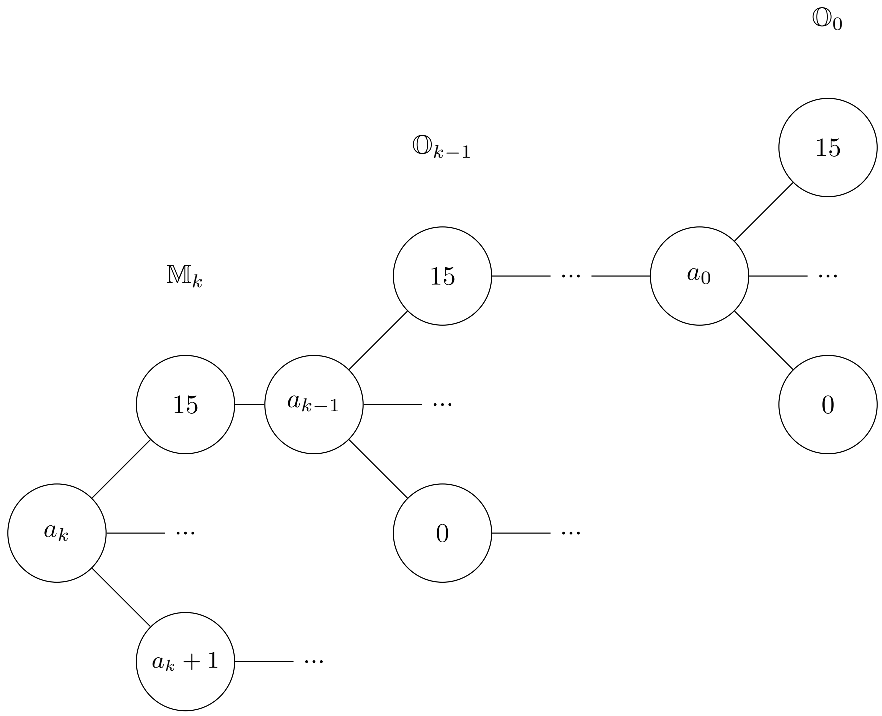

Abbildung 1: Rekursives Verfahren

Abb. 1 zeigt die mögliche Aufrufe des rekursiven Vorgehens. Dabei wird zunächst die maximale Option für das Umlegen von a_k betrachtet (in Abb. 1: 15). Folgend wird die maximale Option für das Umlegen von a_{k-1} gewählt. Sollte dabei zu einem beliebigen Zeitpunkt Bedingung 2 verletzt werden, muss wie bereits erläutert, die nächstniedrigere Option der entsprechenden Koeffizienten gewählt werden.
Der dargestellte Baum verkörpert eine Struktur an allen möglichen Kombinationen für die Vergrößerung von H. Diese werden nach dem beschriebenen Vorgehen, systematisch ausprobiert.

1.5 Vermeiden von „leeren“ Ziffern

Da in der Aufgabenstellung vorausgesetzt ist, dass die Darstellung einer Ziffer beim Umlegen, niemals komplett „geleert“ wird, muss auch diese Einschränkung betrachtet werden. Die beschriebene Vorgehensweise wird dieser Einschränkung jedoch bereits gerecht.
Die Darstellung einer Ziffer kann durch das beschriebene Verfahren, niemals komplett geleert werden, da eine Segmentposition mit dem beschriebenen Verfahren, in einem Zweig, niemals zweimal betrachtet bzw. niemals zwei Aktionen (Wegnehmen und Hinlegen) an dieser Segmentposition getätigt werden.
Das lässt sich vor allem an dem in Unterabschnitt 1.2 beschriebenen Verfahren ableiten. Für jede Ziffer a_i, wird jedes Segment mit der „Zielziffer“ b_i verglichen, wenn ein Segment nicht passt, tritt der entsprechende Fall ein.
Dadurch, dass maximal ein Fall pro Segmentposition eintreten kann, muss dieser, direkt die Eigenschaft (angeschaltet $\vee$ ausgeschaltet) von b_i übernehmen. An den hart-codierten Darstellungen der verschiedenen Ziffern lässt sich ableiten, dass es es insgesamt nur 2 Kombinationen von Ziffern a_i, b_i gibt, welche kein übereinstimmendes vorhandenes Segment besitzen. Für alle anderen Kombinationen tritt also mindestens einmal der Fall aus Unterabschnitt 1.2 ein, bei welchem ein Segment unverändert liegen bleibt. Dadurch, wird bereits automatisch für die Umlegung von fast jeder Ziffer a_i in eine andere Ziffer b_i verhindert, dass die Darstellung „geleert“ wird. Dies ist mit der beschriebenen Logik des einmaligen Ansehens bzw. der einmaligen Aktion zu begründen: Ein Segment bleibt liegen, somit kann die Darstellung nicht geleert werden.
Die beiden hervorzuhebenen Grenzfälle bestehen in den Kombinationen von $1, F$ und $1, E$.
Um diese Problematik zu beheben, ist es wichtig sich das, in Unterabschnitt 1.2 beschriebene, Setzen von *temporär unbestimmten* p_u sowie die Reihenfolge, in welcher die Segmente von zwei Ziffern verglichen werden anzuschauen.
In der Implementierung (Abschnitt 2) wird beim obersten Segment angefangen und im Uhrzeigersinn fortgefahren (siehe Unterabschnitt 2.1).

Wenn somit 1 in F oder E umgelegt werden soll, wird sich das oberste Segment als Erstes angeschaut. Bei diesem ersten Segmentvergleich tritt der Fall ein, dass...

1. ...entweder eine *temporär unbestimmte* Zielposition p'_u, in den bereits vorhandenen Umlegungen, ergänzt wird (somit woanders ein Segment weggenommen wird), wodurch die Darstellung der 1 in diesem Zweig nicht mehr „geleert“ werden kann, da mindestens das oberste Segment vorhanden ist.

2. ...oder keine vorherige Umlegung mit einem *temporär unbestimmten* p'_u vorhanden ist und somit gemäß Unterabschnitt 1.2 eine neue Umlegung mit *temporär unbestimmter* p_u hinzugefügt wird. Der Segmentumsatz ΔS ist nun sicher negativ. Als Nächstes tritt sicher der Fall ein, dass ein Segement weggelegt werden muss um 1 in F bzw. E umzulegen (Segment oben rechts).
   Nun ist die Reihenfolge relevant, in welcher die Umlegungen begutachtet werden, um eine *temporär unbesimmte* Position p_u auszugleichen, falls es mehrere *unbestimmte* p_u gibt ($\Delta S < -1$). Die letzte Umlegung für welche p_u unbestimmt ist, ist die Umlegung des oberen (vorherigen) Segmentes. Wenn somit zuerst für die letzte Umlegung das *temporär unbestimmte* p_u ausgeglichen wird, besteht die Umlegung in der Bewegung des Segmentes von oben rechts (Index 1) an die obere Position (Index 0). Da dieses Segment in diesem Zweig nicht noch einmal beguachtet wird, kann auch hier die Darstellung keines Falles vollständig „geleert“ werden.

Diese Mechanik beeinflusst auf keinem Weg die maximierende Funktion des Algorithmus, da lediglich die Reihenfolge, in welcher die *temporär unbestimmten* p_u aufgefüllt werden festgelegt wird. Falls $F \to 1$ oder $E \to 1$ umgelegt wird, greift die gleiche Mechanik bloß „später“. Bei F tritt die beschriebene Fallunterscheidung bei Segment mit Index 4 (unten links) ein $\Rightarrow$ Bei E bereits Index 3 (unten). Durch diese erzwungene Umlegung innerhalb der Ziffer, wird garantiert, dass die Darstellung der Ziffer nicht „geleert“ werden kann.

1.6 Optimierungen

Zwei wichtige Optimierungen die beim rekursiven Verfahren getroffen werden können, betreffen die in Unterabschnitt 1.4 beschriebene Bedingung 1. Es lassen sich zwei weitere Bedingungen ableiten, welche bereits während der rekursiven Aufzählung, wie auch Bedingung 2 in Unterabschnitt 1.4 eingesetzt werden können:

1. $\Delta S \leq 7 \cdot (i-1) - [\text{Anzahl der Segmente von } a_0 \text{ bis } a_{i-1}]$

   wobei i dem aktuelle Index des Koeffizienten in der Rekursion entspricht.
   Die Anzahl der *unbestimmten* p'_u (ΔS) entspricht der Anzahl der Umlegungen, für welche noch keine Position zum Ablegen des Segmentes festgelegt wurde. Diese Anzahl, darf die Anzahl der freien Plätze in den verbleibenden Darstellungen der Koeffizienten a_j mit $j < i$ nicht überschreiten. Die Anzahl der verbleibenden freien Plätze (ausgeschaltet), leitet sich dabei über $7 \cdot (i-1) -$ [Anzahl der Segmente von a_0 bis a_{i-1}] her.

2. $-\Delta S \leq [\text{Anzahl der Segmente von } a_0 \text{ bis } a_{i-1}] - 2 \cdot (i-1)$

   wobei i dem aktuelle Index des Koeffizienten in der Rekursion entspricht.
   Die Anzahl der *unbestimmten* p_u ($-\Delta S$) entspricht der Anzahl der Umlegungen, für welche noch keine Position zum Wegnehmen des Segmentes festgelegt wurde. Diese Anzahl darf die Anzahl der belegten Plätze in den verbleibenden Darstellungen der Koeffizienten a_j mit $j < i$ nicht überschreiten. Da nachdem alle Umlegungen getätigt wurden, valide Ziffern vorhanden sein müssen, müssen in jeder Darstellung noch mindestens 2 Segmente übrig bleiben (Darstellung einer 1 $\Rightarrow$ benötigt am wenigsten Segmente).

Mit diesen Optimierungen vermeidet man in vielen Fällen unnötige Zweige und kann früher, eine rekursive Aufzählung vermeiden, welche auf einer falschen Wahl einer Option eines Koeffizienten basiert.

Weiterführend lässt sich die beschriebene Logik der maximierenden Optionen $\mathbb{M}$ ausweiten. Nicht nur der erste Koeffizient a_k, welcher verändert wird, muss vergrößert werden. Wenn $\Delta S = 0$, müssen keine Segmente frei oder benutzt werden. Aus diesem Grund kommen für den nächsten Koeffizienten lediglich die maximierenden Optionen $\mathbb{M}_i$ infrage, wobei der Koeffizient womöglich auch unverändert bleiben kann. Da keine Veränderung keine Umlegungen verbraucht und den Segmentumsatz ΔS nicht verändert.

Außerdem lässt sich eine Optimierung im Rahmen des Ausprobierens der Optionen treffen. Beim wiederholten Auftreten eines Zifferntypes, ist es ineffizient, die gleichen Optionen nochmals auszuprobieren, während in der Zwischenzeit an der Gesamtzahl nichts verändert wurde und somit die gleichen Umstände (ΔS und m') bestehen. Wenn somit ein Zweig beispielsweise aufgrund von Bedingung 1 fehlschlägt, kann die gewählte Option zwischengespeichert werden. Somit wird verhindert, das wiederholt die gleiche Fehloption gewählt wird.

1.7 Erweiterungen

1.7.1 Minimierung der Hexadezimalzahl (*HexMin*)

Eine offensichtliche Erweiterung des *HexMax*-Problems ist das *HexMin*-Problem. Es besteht darin, eine gegebene Hexadezimalzahl mit maximal m Umlegungen zu minimieren. Dabei gelten die gleichen Umstände wie beim *HexMax*-Problem (Vermeidung von „leeren Ziffern" usw.). Intressanterweise lässt sich das *HexMin* über eine minimale Abwandlung der beschriebenen Idee lösen. Die einzige Veränderung muss bei der Reihenfolge des Ausprobierens der Optionen durchgeführt werden. Die Optionen $\mathbb{O}_i$ dürfen beim rekursiven Vorgehen nicht mehr absteigend ausprobiert werden, sondern aufsteigend. Zudem müssen die Optionen $\mathbb{M}$ verkleinernd anstatt vergrößernd agieren:

$$\mathbb{M}_i = \{x \mid x \in \mathbb{O}_i, x < a_i\}$$

da um die gesamte Hexadezimalzahl zu verkleinern, mindestens ein Koeffizient a_i verkleinert werden muss. Alle anderen Punkte des Algorithmus bleiben unverändert. Details zur Implementierung dieser Erweiterung sind in Unterunterabschnitt 2.5.1 zu finden.

1.7.2 Weitere Zahlensystem (z.B. *BinMax*, *DezMax*)

Eine weitere Erweiterung betrifft das Hexadezimalsystem. Da die beschriebene Logik aus Gleichung 2 auch für alle anderen polyadischen Zahlensysteme mit beliebiger Basis b zutrifft:

$$\begin{aligned}(b-1) \cdot \sum_{i=0}^{j} b^i &< b^{j+1} \qquad \mid \quad : b^j \\ (b-1) \cdot (b^{-j} + b^{1-j} + ... + b^0) &< b \\ (b^{1-j} + b^{2-j} + ... + b) - (b^{-j} + b^{1-j} + ... + b^0) &< b \\ b - b^{-j} &< b \qquad \mid \quad -b \\ -b^{-j} &< 0\end{aligned} \tag{4}$$

kann die Lösungsidee auch auf eine beliebige Basis ausgeweitet werden. Dafür muss lediglich der Ziffernvorrat $\mathbb{Z}_b$ entsprechend angepasst werden. Weitere Details sind in Unterunterabschnitt 2.5.2 zu finden. Außerdem kann diese Erweiterung ohne Probleme mit Unterunterabschnitt 1.7.2 kombiniert werden.
Durch diese Erweiterungen können Zahlen aus beliebigen Zahlensystemen mit maximal m Umlegungen in der SSA maximiert und minimiert werden.

2 Umsetzung

2.1 hexInSSD

Zunächst muss eine Übersetzung eines Wertes $a_i \in \mathbb{Z}_{16}$ in die Darstellung der SSA implementiert werden. Die Darstellung muss dabei hart codiert werden. Dafür wurde die Funktionalität von Schlüssel-Werte-Paaren einer *Dictionary* genutzt. Der String der möglichen Ziffern wird in eine Liste von 7 verschiedenen $1 \vee 0$ übersetzt (1: Segment ist vorhanden, 0: Segment ist nicht vorhanden). Die verschiedenen Segmente werden dabei in den Indizes wie folgt abgebildet:

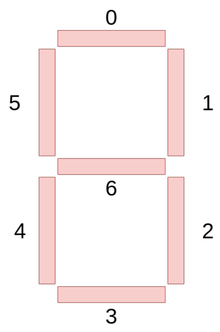

Abbildung 2: Deklaration der verschiedenen Segment-Indizes

Mit diesem Vorgehen kann somit zu jedem Zeitpunkt eine beliebige Ziffer in ihre Darstellung in der SSA übersetzt werden.

2.2 main() und parseInput()

Zunächst muss eine Eingabedatei übergeben werden. Dies kann entweder über Argumentübergabe (z.B. python A3.py hexmax0.txt) oder manuelle Eingabe nach Starten des Programmes vollzogen werden. Die Werte aus der Eingabedatei müssen gelesen werden. Dafür eignet sich die *readlines()* Funktion von *Python*, mit welcher hierbei die Zeilen der Eingabedatei als Liste von Strings zurückgegeben werden. Somit kann m, die Maximalzahl an Umlegungen und H, die Hexadezimalzahl ausgelsen werden. In der *main()*-Funktion werden folgend einige Flaggen, welche ebenfalls als Argumente übergeben werden, eingelesen. Diese dienen vor allem zum Anwenden von Erweiterungen (Unterabschnitt 1.7). Folgend wird das *HexMax*-Problem für die eingelesenen Werte mit *maximieren()* gelöst. Abschließend wird das Ergebnis in der Konsole sowie in einer Datei mit dem Präfix *ergebnis_* ausgegeben.

2.3 maximieren()

Die *maximieren()*-Funktion soll den Hauptlösungsprozess initialisieren, sowie das Ergebnis der *umwandeln()*-Funktion anwenden (die ermittelten Umlegungen sollen „durchgeführt“ werden). Als Parameter werden die *hexZahl* (String) und die *maxUmlegungen* (int) übergeben. Zudem kann eingestellt werden, inwiefern der „Zwischenstand“ nach jeder Umlegung angezeigt werde soll. Zunächst wird die *umwandeln()*-Funktion ausgeführt, welche die Liste der Umlegungen (s.o. $\mathbb{U}$) und somit die grundlegende Lösung zurückgibt. (Unterabschnitt 2.4). Anschließend wird überprüft, ob Umlegungen gefunden wurden (die Liste der Umlegungen nicht leer ist). In diesem Fall wird die *hexZahl* der Eingabe zurückgegeben. Andernfalls, wird eine Liste für die SSA initialisiert, in welcher danach, in einer Iteration über alle herausgefundenen Umlegungen, die dementsprechenden Umlegungen getätigt werden. Zuletzt, wird mithilfe der *hexInSSD-Dictionary*, die Liste der SSA in einen *ergebniseString* (String) umgewandelt. In der Theorie wäre es auch möglich direkt in der Funktion *maxZiffer()*, beim Ermitteln der Umlegungen, die einzelnen Umlegungen direkt durchzuführen, jedoch ist es gefordert, die Zwischenstände nach jeder Umlegung zu ermitteln. Da im rekursiven Prozess die Umlegungen noch nicht direkt feststehen (Unterabschnitt 1.2), muss noch mindestens einmal über die abgeschlossene Liste (entsprechend $\mathbb{U}$) iteriert werden.

2.4 maxZiffer()

Diese Funktion beinhaltet den Hauptteil des beschriebenen Algorithmus und die entscheidende Funktionalität zum Lösen des *HexMax*-Problems (Rekursive Funktion). Als Parameter ist m, die Zahl der maximalen Umlegungen, sowie die *hexZahl* (String) zu übergeben, während des Rekursionsprozesses, werden zusätzlich

index Index der aktuellen Ziffer von links nach rechts (nicht wie in Unterabschnitt 1.1)

schritte Liste an bereits getätigten Umlegungen im aktuellen Zweig der Rekursion. Eine Umlegung muss dabei die beiden Positionen $p_u \wedge p'_u$ des Segmentes beinhalten. Somit besteht ein Element der *schritte*-Liste hierbei in einer weiteren Liste mit vier Elementen: [Index alt, Segment Index alt, Index neu, Segment Index neu].

übrigerUmsatz Der Segmentumsatz ΔS. Die Variable *übrigerUmsatz* ließe sich auch mit den Elemente der schritte Liste reproduzieren. Um diese Iteration zu vermeiden, sollen mit *übrigerUmsatz* jedoch direkt die Umlegungen in *schritte* ausgemacht werden, welche über *temporär unbestimmte* Positionen verfügen.

tempOptionen Im Rahmen der Optimierungen (Unterabschnitt 1.6), muss eine Liste angelegt werden, in welcher die bereits gescheiterten Optionen b_i übergeben werden.

übergeben.

2.4.1 Abbruchkriterien

Zunächst muss ein Abbruchkriterium für die Rekursion hinterlegt werden: Wenn der *index* größer als die Anzahl an Ziffern ist, wurde jede Ziffer betrachtet und maximiert. Weitergehend muss überprüft werden, ob Bedingung 1 (Unterabschnitt 1.4) erfüllt ist, und somit keine übrigen Segmente vorhanden sind. Dafür muss lediglich *übrigerUmsatz* (ΔS) überprüft werden. Sollte diese Bedingung erfüllt sein, werden die *schritte* zurückgegeben, welche hierbei der Lösung entsprechen. Andernfalls wird eine leere Liste, als Signal für einen gescheiterten Versuch in diesem Zweig (keine validen Umlegungen in diesem Zweig der Rekursion möglich), zurückgegeben.
Weitere Abbruchkritieren sind durch die Optimierungen (Unterabschnitt 1.6) gegeben. Für die Implementierung eignet sich ebenfalls der *übrigerUmsatz* Wert. Wenn *übrigerUmsatz* die Anzahl an freien Segmentpositionen in den Darstellungen der restlichen Ziffern überschreitet, oder die Anzahl an bewegbaren Segmenten (jede Ziffer benötigt mindestens 2 Segmente) unterschreitet, wird ebenfalls eine leere Liste zurückgegeben.

2.4.2 Iteration über alle Optionen

Zunächst wird eine Kopie der übergebenen Liste der *tempOptionen* angelegt. Diese beinhaltet die bereits gescheiterten Optionen, welche somit nicht mehr ausprobiert werden müssen. Für die Iteration muss die aktuelle *ziffer*, welche hierbei über den *index* bestimmt wird, systematisch in die höchstmögliche Option $b_i \in \mathbb{O}_i$ umgelegt werden, um den Zweig der Rekursion zu erweitern. Dafür wird über alle in *hexInSSD* implementierten Schlüssel iteriert. Wichtig dabei ist die Reihenfolge dieser Schlüssel. Durch die Sortierung von $F \rightarrow 0$, wird garantiert, dass bei der Iteration von der größten bis zur kleinste Option ausprobiert wird. Sollte das Paar aus *ziffer* und iteriertem Schlüssel bereits in dem *tempOptionen* vorhanden sein, wird sofort zur nächsten Option (bzw. Schlüssel) übergegangen (*continue*). Daraufhin muss auch die beschriebene Logik der maximierenden Optionen $\mathbb{M}_k$ beachtet werden (Unterabschnitt 1.3). Solange in der gesamten Darstellung noch keine Umlegungen getätigt wurden, darf keine Option gewählt werden, welche den Wert der *ziffer* verkleinert (gemäß 1.3 $\Rightarrow a_k$ wurde noch nicht festgelegt). Sobald also über alle Schlüssel iteriert wurde, welche die aktuelle Ziffer vergrößern und noch keine Umlegungen getätigt wurden, bleibt die aktuelle Ziffer unverändert und es wird zur nächsten Ziffer fortgeschritten. In diesem Zuge lässt sich auch eine beschriebene Optimierung einbringen (Unterabschnitt 1.6). Nicht nur wenn noch keine Umlegungen getätigt wurden, sollte nur die maximierenden Optionen $\mathbb{M}$ in Frage kommen. Auch wenn $\Delta S = 0$, kommen wie in Unterabschnitt 1.6 beschrieben, nur die maximierenden Optionen in Frage. Sollte die Iteration somit der Schlüssel die *ziffer* erreichen, wird ein neuer Funktionsaufruf mit erhöhtem Index (*index+1*) der *maxZiffer()*-Fuktion zurückgegeben.

2.4.3 Umlegen einer Ziffer

In der Iteration über alle Schlüssel, nachdem die genannten Bedingungen überprüft wurden, wird der *ziffer* in den iterierten Schlüssel umgelegt. Dieser Schritt entspricht dabei Unterabschnitt 1.2 aus der Lösungsidee. Zunächst wird dafür eine Kopie der durchgeführten *schritte* bzw. Umlegungen im aktuellen Zweig angelegt. Der Prozess des Umlegens besteht wie bereits in Unterabschnitt 1.2 angedeutet, aus einer Iteration über alle Segmentindizes. In dieser Iteration lassen sich problemlos die beschriebenen Fälle mit *if*-Abfragen umsetzen (Unterabschnitt 1.2). Die *temporäre Unbestimmtheit* wird dabei mit zwei Instanzen der *NoneType*-Klasse an den entsprechenden Stellen in Kopie der *schritte*-Liste umgesetzt. Mit der Kopie wird dabei baumübergreifende Mutation verhindert. Zusätzlich muss dem eintretenden Fall entsprechend, der Segmentumsatz ΔS bzw. *übrigerUmsatz* verändert werden.
Hierbei ist die in Unterabschnitt 1.5 beschriebene Logik relevant: Im Fall, dass ein Segment in b_i nicht

vorhanden ist, wird zunächst die letzte Umlegung mit *temporär unbesimmtem* p_u aufgefüllt, insofern solch eine vorhanden ist. Daraufhin wird die aufgefüllte Umlegung mit der *insert()* Funktion vor die anderen *unbestimmen* Umlegungen verschoben, sodass weiterhin problemlos die letzte Umlegung aufgefüllt werden kann. Somit wird verhindert, dass Darstellungen komplett „geleert" werden (Unterabschnitt 1.5).
Nach jedem Segment, für welches der Koeffizient a_i mit dem Schlüssel b_i verglichen wird, wird Bedingung 2 aus Unterabschnitt 1.4 überprüft (Anzahl an Umlegungen $\leq m$). Dies kann ohne Probleme mit der Länge der Kopie der *schritte*-Liste vollzogen werden.
Im Falle, dass Bedingung 2 verletzt wird, wird die Kombination aus *ziffer* und ausprobiertem Schlüssel in die Kopie der Liste der *tempOptionen* eingefügt und die Iteration über die Segmente abgebrochen, da die Maximalanzahl an Umlegungen m bereits überschritten wurde.
Sollte die Iteration über alle Segmente ohne Abbruch abgeschlossen werden, wurde Bedingung 2 nicht verletzt. Dieser Fall kann mit der *for-else* Logik umgesetzt werden. Die aktuelle Ziffer wurde erfolgreich umgelegt und es kann somit zur nächsten Ziffer übergegangen werden. Dies wird mit einem neuen Funktionsaufruf der *maxZiffer()*-Funktion mit einem erhöhten Index (*index+1*) bewirkt. Zudem wird dabei im Falle, dass eine Option gewählt wurde, welche nicht mit der *ziffer* übereinstimmt, eine leere Liste als *tempOptionen* übergeben, da durch die Umlegung neue Umstände (ΔS und m') entstehen und somit womöglich bislang fehlgeschlagene Optionen wieder nutzbar werden. Dieser erneute rekursive Funktionsaufruf gibt eine Liste an Umlegungen zurück welche der Lösung entsprechen, insofern es eine valide Lösung in diesem Zweig des Brute-Force-Baumes gibt. Sollte diese Liste nicht leer sein und somit die Lösung des Problems darstellen, wird diese ein weiteres Mal zurückgegeben. Andernfalls ist die gewählte Option mit welcher der Zweig fortgesetzt wurde in die Liste der fehlgeschlagenen Optionen (*tempOptionen*) einzufügen. Es wird mit der nächstgeringeren Option fortgefahren.

Sollte während der Iteration über alle Schlüssel und somit alle Optionen, bislang keine Umlegungsliste (Lösung) zurückgegeben worden sein, wird anschließend eine leere Liste zurückgegeben.

2.5 Erweiterungen

2.5.1 Minimierung der Hexadezimalzahl (*HexMin*)

Für die Umsetzung der Minimierung der Hexadezimalzahl, muss wie bereits in Unterunterabschnitt 1.7.1 geschildert lediglich die Reihenfolge, in welcher die verschiedenen Optionen ausprobiert werden, umgekehrt werden.
Zunächst muss jedoch als Indikator der Minimierung, ein zusätzliches boolesches Argument übergeben werden (*min*). Sollte dieser Wert wahr sein, wird bei der Iteration über alle Schlüssel, die *reversed()* Funktion verwendet, und somit die Reihenfolge der Schlüssel umgekehrt. Die vorherige Reihenfolge ist aufsteigend (Unterunterabschnitt 2.4.2), weswegen mit dieser Operation das gewünschte Resultat, eine absteigende Iteration der Schlüssel, entsteht.

2.5.2 Weitere Zahlensystem (z.B. *BinMax*, *DezMax*)

Die weiteren Zahlensysteme können gemäß Unterunterabschnitt 1.7.2 problemlos durch weitere *Dictionaries* wie bereits Unterabschnitt 2.1, hinzugefügt werden. Zudem wird in der rekursiven Funktion *maxZiffer()*, das zu benutzende *Dictionary* als weiteres Parameter übergeben. Die Wahl des Zahlensystems findet Umsetzung über eine Flagge beim Ausführen des Programms, welche in der *main()*-Funktion in das entprechende anzuwendene *Dictionary* übersetzt wird und folgend über die *maximieren()*-Funktion an die *maxZiffer()*-Funktion übergeben wird.

3 Zeitkomplexität

Die Analyse der Zeitkomplexität stellt sich als kompliziert heraus. Zunächst handelt es sich um einen rekursiven Brute-Force-Ansatz, mit welchem viele Möglichkeiten, um die Gesamtzahl H zu vergrößern ausproiert werden. An der Baumstruktur (Abb. 1) lässt sich passend die Anzahl an neuen Vergrößerungsmöglichkeiten pro Ziffer a_i ablesen. Da für jede Ziffer, 15 Optionen zur Veränderung vorliegen, ergibt sich die Anzahl an Möglichkeiten zur Vergrößerung der Hexadezimalzahl im Extremfall als 15^n, wobei n die Anzahl an Ziffern beschreibt. Sollte ein Algorithmus vorliegen, mit welchem alle diese Möglichkeiten ausprobiert werden, um eine Lösung zu finden ergibt sich eine asymptotische Laufzeit von $O(15^n)$. Für eine beliebige Basis b (Unterunterabschnitt 1.7.2) ergibt sich dabei $O((b-1)^n)$.

Da die umgesetzte Lösungsidee jedoch einige Optimierungen und Abwandlungen eines reinen Brute-Force-Algorithmus (Beschneidungen des Baumes) einbindet, muss eine tiefergreifende Analyse durchgeführt werden. Dafür muss eine noch stärkere Abstraktion des Problems vollzogen werden. Dabei ist interessant, ob die geschliderten Optimierungen möglicherweise eine polynomielle asymptotische Laufzeit bewirken. Zudem stellt sich die Frage, ob es für das *HexMax* Problem überhaupt eine Lösung in polynomieller Laufzeit gibt und ob es sich somit überhaupt umgangssprachlich effizient lösen lässt:

$$HexMax \in P \vee NPC \qquad \text{mit } P \neq NP \tag{5}$$

Für eine mögliche Reduktion zum Beweis der *NP*-Vollständigkeit, muss das *HexMax* zunächst als Entscheidungsproblem anstatt als Optimierungsproblem formuliert werden.

3.1 Formulierung als Entscheidungsproblem

Eine Abwandlung als Entscheidungsproblem könnte lauten: „Ist es möglich, mit maximal m Umlegungen die eingegebene Hexadezimalzahl H_1 mindestens so groß zu machen wie H_2?“, wobei H_2 eine weitere Eingabe ist. Es ist im Folgenden wichtig, dass eine Lösung dieses Entscheidungsproblems (*HexMaxDecision*) auch zu einer Lösung des unveränderten *HexMax*-Problem beiträgt und somit später gefolgert werden kann, dass *HexMaxDecision* in der gleichen Komplexitätsklasse ist, wie *HexMax*.
Das *HexMax*-Problem lässt sich beispielsweise mit einer *binären Suche* in Kombination mit *HexMaxDecision* lösen. So kann mit einer Variation von H_2 die maximale Zahl $H_{2,max}$ ermittelt werden, welche an das *HexMaxDecision*-Problem übergeben werden kann und eine Ja-Instanz zurückgibt. Wenn H_1 beispielsweise aus 2-Ziffern besteht und $m = 3$, dann wird zunächst das *HexMaxDecision*-Problem beispielsweise mit $H_2 = 4E$ aufgerufen. Sollte es eine Ja-Instanz zurückgeben, wird als nächstes für H_2 die Zahl auf der $4E + \frac{(FF-4E)}{2}$ eingesetzt. Mit diesem Schema wird sich immer weiter an den Maximalwert für H_2 und somit an die Lösung des *HexMax*-Problems herangetastet.
Die *binäre Suche* hat dabei eine Zeitkomplexität von $O(\log n)$, dadurch, dass bei einer geschickten Wahl, der mögliche Bereich bei jeder Instanz halbiert wird. n entspricht hierbei jedoch nicht mehr der Anzahl an Ziffern, sondern der Anzahl an möglichen verschiedenen Zahlen welche für H_2 eingesetzt und wächst somit mit der Anzahl an Ziffern von H_1 exponentiell (Anzahl an möglichen Zahlen $= 16^{\text{Anzahl an Ziffern}}$). Wenn das n somit weiterhin als Anzahl der Ziffern definiert ist, ergibt sich:

$$O(\log 16^n) = O(4n) = O(n) \tag{6}$$

Es ist wichtig, dass *HexMaxDecision* über Polynomialzeit (hier mit $O(n)$) mit *HexMax* zusammenhängt. Man kann von einer Polynomialzeitreduktion von $HexMax \leq_p HexMaxDecision$ sprechen und damit *HexMaxDecision* mindestens genauso schwer (laufzeittechnisch) ist wie *HexMax*.

3.2 *NP-Zertifikat*

Als Nächstes soll für *HexMaxDecision* gezeigt werden, dass es sich in NP befindet ($HexMaxDecision \in NP$). Dafür wird ein *NP-Zertifikat* benötigt, mit welchem eine Ja-Instanz des *HexMaxDecision*-Problems in Polynomialzeit verifiziert werden kann (NP-Komplexitätsklasse). Ein plausibles Zeritfikat wären z.B. eine Liste der Umlegungen, welche für $H_1 \geq H_2$ durchgeführt werden müssen. Diese Umlegungen wären mit einer Iteration in Polynomialzeit auf die Eingabe H_1 anzuwenden. Nun kann die Ja-Instanz verifiziert werden, indem überprüft wird, ob die Hexadezimalzahl, welche nach dem Umlegen entsteht, mindestens so groß ist, wie H_2. Es folgt $HexMaxDecision \in NP$.

3.3 Fomalisierung des Entscheidungsproblems

Jede Option b_i, in die ein Koeffizient a_i umgelegt werden kann, hat verschiedene Werte. Zum Einen entsteht ein Segmentumsatz $\Delta S'$, zum Anderen wird eine Anzahl an Umlegungen m' benötigt. Da jede Umlegung das Wegnehmen eines Segmentes beinhaltet, wird im Folgenden eine Umlegung verbraucht, wenn ein Segment weggenommen wird. Außerdem lässt sich jede Option mit einem Gewinn bzw. Verlust p_i für die Gesamtzahl H beschreiben. Dieser lässt sich dabei gemäß Unterabschnitt 1.1 als:

$$p_i = (b_i - a_i) \cdot 16^i \tag{7}$$

ausdrücken. Somit lässt sich jede Option durch eine Tripel der Form $o_i = (p, \Delta S', m')$, $o_i \in \mathbb{O}_i$ beschreiben. $\Delta S'$ und m' sind dabei rein von der Darstellung von a_i und b_i abhängig, weswegen kein eindeutiger algebraischer Körper für zur Modellierung der Tripel gefunden werden kann. Durch Brute-Force kann herausgefunden werden, dass es insgesamt 173 einzigartige Tripel o gibt, wobei p_i dabei ausschließlich mit 16^0 berechnet wurde. Jedoch lässt sich daran erkennen, dass es nur begrenzt viele Möglichkeiten für die Abhängigkeiten bzw. Tripel von $\Delta S'$,m',p gibt. Davon abgesehen, lassen sich die Bedingungen des Problems ausdrücken:

$$\begin{aligned}
\text{Die Hexadezimalzahl muss mindestens so groß wie } H_2 \text{ werden: } & \sum_{i=1}^{n} \sum_{j \in \mathbb{O}_i} p_j \cdot x_{i,j} > H_2 \\
\text{Die Anzahl an Segmenten darf nicht verändert werden: } & \sum_{i=1}^{n} \sum_{j \in \mathbb{O}_i} \Delta S'_j \cdot x_{i,j} = 0 \\
\text{Keine Überschreitung der Umlegungen: } & \sum_{i=1}^{n} \sum_{j \in \mathbb{O}_i} m'_j \cdot x_{i,j} \leq m \\
\text{Maximal ein Tripel (Option) pro Ziffer: } & \sum_{j \in \mathbb{O}_i} x_{i,j} = 1 \quad i \in \{1, ..., n\} \\
\text{Jede Option darf nur einmal gewählt werden: } & x_{i,j} \in \{0,1\}
\end{aligned} \tag{8}$$

Eine Ja-Instanz kann somit durch ein Zutreffen all dieser Bedingungen abgebildet werden.

3.4 Ähnlichkeit zum *Mehrfachauswahl-Rucksack-Problem*

Es fällt auf, dass die formulierte Abstraktion, bei Vernachlässigung der Einschränkungen durch die begrenzte Anzahl an existierenden Tripel, Ähnlichkeit zum *Mehrfachauswahl-Rucksack-Problem (MCKP)* aufweist. Es ist bekannt: $MCKP \in NPC$. Das Entscheidungsproblem des *MCKP* lässt sich als:

$$\begin{aligned}
\sum_{i=1}^{n} \sum_{j \in N_i} p_j \cdot x_{i,j} &> V \\
\sum_{i=1}^{n} \sum_{j \in N_i} w_j \cdot x_{i,j} &\leq W \\
\sum_{j \in N_i} x_{i,j} &= 1 \quad i \in \{1, ..., n\} \\
x_{i,j} &\in \{0,1\}
\end{aligned} \tag{9}$$

definieren (Rucksackproblem).
Der einzige Unterschied besteht in $\Delta S'$, was als weiterer Parameter für das *HexMaxDecision* hinzukommt.

3.5 Polynomialzeit-Reduktion

Wenn es möglich ist, das *MCKP* mithilfe vom *HexMaxDecision*-Problem zu lösen (Polynomialzeit-Reduktion), folgt $HexMaxDecision \in NPC \Rightarrow HexMax \in NPC$, was bedeuten würde, dass das *HexMax*-Problem nicht „einfach“ zu lösen ist. Dafür muss die Eingabe jeder *MCKP*-Instanz in Polynomialzeit in die Eingabe von *HexMaxDecision* umgeformt werden. Daraufhin wird die Ausgabe von *HexMaxDecision* ebenfalls als Lösung für *MCKP* zurückgegeben. Nötig ist dabei jedoch, dass jede Ja-Instanz und Nein-Instanz von *HexMaxDecision*, auch der jeweiligen Instanz von *MCKP* mit dem transformierten Input entspricht.

Die Reduktion ($MCKP \leq_p HexMaxDecision$) verläuft aufgrund der starken Ähnlichkeit sehr unkompliziert. Ein großes Problem dabei ist jedoch die Einschränkung durch die begrenzten Tripel, welche hierbei jedoch weiterhin vernachlässigt werden. Die Eingabe von *MCKP* ließe sich in Polynomialzeit auf die Eingabe von *HexMaxDecision* reduzieren, indem lediglich für jede Option j der Klasse N_i, eine weitere Zahl ($\Delta S'$) eingefügt wird. Für alle Optionen gilt $\Delta S'_j = 0$. Durch diese Transformation, würde Zeile 2 in den Bedingungen des *HexMaxDecision*-Problems für eine Ja-Instanz eliminiert werden. Außerdem gilt für jede Option $m'_j = w_j$ und $p_j = p_j$. Die Probleme sind nach der Eliminierung von $\Delta S'$ in *HexMaxDecision* identisch. Somit kann *MCKP* auf *HexMaxDecision* reduziert werden.
Insgesamt ist diese Reduktion jedoch aufgrund der Einschränkungen und begrenzten Kombinationen an

$m', \Delta S', p$, welche konsequent ignoriert wurden, nicht so einfach möglich. Formal ist also weiterhin unsicher, inwiefern $HexMaxDecision \in NPC \vee P$ bzw. $HexMax \in NPC \vee P$.
Interessant ist dennoch die Ähnlichkeit und der nahezu gelungene Beweis der NP-Vollständigkeit des *HexMax*-Problems, welcher zur starken Vermutung verleitet, dass $HexMax \in NPC$ und somit die Optimierungen des Brute-Force-Verfahrens (Unterabschnitt 1.6), bereits eine angebrachte Lösung bieten.

3.6 Entwicklung der Reallaufzeit

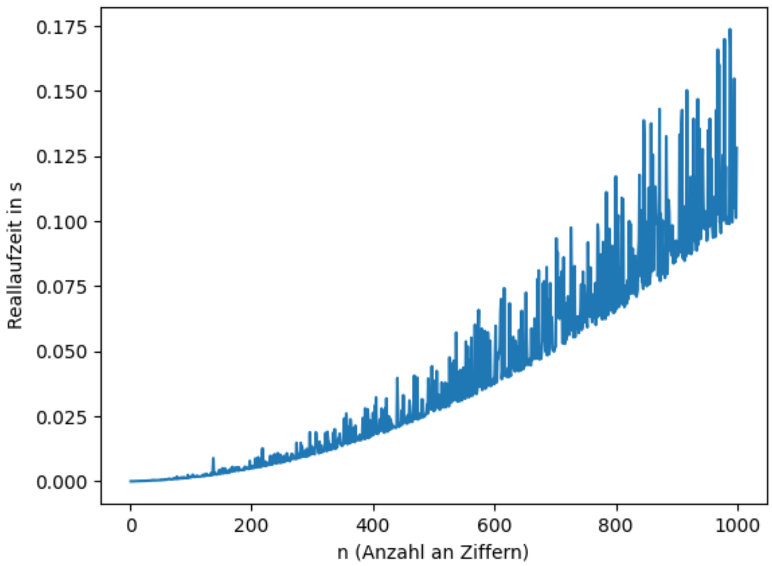

Abbildung 3: Reallaufzeit in Abhängigkeit der Anzahl an Ziffern n.
Randomisierung von H und $m = [\lfloor n * 0.5 \rfloor, \lfloor n * 1.5 \rfloor]$

4 Beispiele

4.1 Randinformationen

Das Programm kann durch `python A3.py hexmax[X].txt` ausgeführt werden. Dabei können wahlweise folgende Flaggen angefügt werden:

- `-d` Ausgabe des Zwischenstandes in der Konsole in visueller Form
- `-min` Erweiterung: Minimierung statt Maximierung der Eingabe
- `-bin` Erweiterung: Maximierung bzw. Minimierung einer Binärzahl ($\mathbb{Z}_2$), die Eingabe muss manuell entsprechend angepasst werden.
- `-dec` Erweiterung: Maximierung bzw. Minimierung einer Dezimalzahl ($\mathbb{Z}_{10}$), die Eingabe muss manuell entsprechend angepasst werden.

Alle Lösungen werden zusätzlich in einer Datei mit dem Präfix *ergebnis_* abgegeben.
Bei der Beurteilung der tatsächlichen Laufzeit, muss die Ineffizienz der Programmiersprache *Python*, sowie die Nutzung der CPU: *Intel Core i7-4770 3.40 GHz* beachtet werden.
Aus Platzgründen, werden lediglich die **getätigten Umlegungen** anstatt des geforderten Zwischenstandes in der Dokumentation angegeben. Diese lassen jedoch ohne Probleme auf den jeweiligen Zwischenstand schließen, welcher sich zudem durch einen einfachen *print()* Befehl in der *maximieren()*-Funktion oder visuell durch die Flagge `-d` ausgeben ließe. Auf der BWINF-Webseite, wird auch von einer „Auflistung von Umlegungen" gesprochen.
Jede Umlegung besteht dabei aus einer Liste mit vier Elementen. Die ersten beiden beschreiben den „Herkunftsort" bzw. p_u (Index der Ziffer, Index des Segmentes) des umgelegten Segmentes, während das dritte

und vierte Element die gleichen Angaben für p'_u. Die Segmentindizes werden gemäß Unterabschnitt 2.1 zugewiesen.

4.2 BWINF-Beispiele

hexmax0.txt

`python A3.py hexmax0.txt`
`Umlegungen:`
`[[0, 1, 0, 0], [0, 2, 0, 5], [1, 1, 1, 5]]`
Lösung:
`EE4`
3/3 Umlegungen benötigt
— 0.0010 Sekunden —

hexmax1.txt

`python A3.py hexmax1.txt`
`Umlegungen:`
`[[0, 2, 0, 4], [0, 3, 1, 6], [1, 1, 2, 4], [1, 2, 3, 6], [1, 3, 4, 0], [2, 1, 4, 4], [2, 2, 5, 5], [2, 3, 6, 0]]`
Lösung:
`FFFEA97B55`
8/8 Umlegungen benötigt
— 0.0020 Sekunden —

hexmax2.txt

`python A3.py hexmax2.txt`
`Umlegungen: [[0, 2, 1, 4], [0, 3, 1, 5], [1, 1, 2, 5], [1, 2, 3, 0], [1, 3, 4, 5], [2, 1, 5, 4], [2, 3, 6, 0], [3, 2, 7, 4], [3, 3, 7, 5], [4, 1, 10, 0], [4, 3, 10, 4], [5, 1, 10, 5], [5, 2, 10, 6], [5, 3, 11, 0], [6, 2, 11, 4], [6, 3, 11, 5], [7, 1, 11, 6], [7, 2, 13, 0], [7, 3, 13, 4], [8, 1, 14, 4], [8, 2, 15, 6], [8, 3, 16, 3], [10, 1, 16, 4], [10, 2, 16, 6], [11, 1, 17, 1], [11, 2, 19, 5], [12, 1, 21, 6], [12, 2, 25, 2], [12, 3, 25, 5], [13, 1, 26, 6], [13, 2, 29, 6], [14, 1, 32, 0], [14, 2, 32, 4], [14, 3, 34, 1], [15, 1, 36, 3], [15, 2, 36, 4], [15, 3, 36, 6]]`
Lösung:
`FFFFFFFFFFFFFFFFD9A9BEAEE8EDA8BDA989D9F8`
37/37 Umlegungen benötigt
— 0.0020 Sekunden —

hexmax3.txt

`python A3.py hexmax3.txt`
Lösung:
`FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF`
`AA98BB8B9DFAFEAE888DD888AD8BA8EA8888`
121/121 Umlegungen benötigt
— 0.0042 Sekunden —

hexmax4.txt

`python A3.py hexmax4.txt`
Lösung:
`FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEB8DE88BAA8ADD888898E9BA88`
`AD98988F898AB7AF7BDA8A61BA7D4AD8F888`

87/87 Umlegungen benötigt
— 0.0050 Sekunden —

hexmax5.txt

`python A3.py hexmax5.txt`
Lösung:
```
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF88EFA9EBE89EFA99FBDAA8E8EAD88AB89
9F8E8F9AA9E9AD88988EDA9A99888EDAD989A8BAFD8A88888888888888888888
8888888888888888888888888888888888888888888888888888888888888888
8888888888888888888888888888888888888888888888888888888888888888
8888888888888888888888888888888888888888888888888888888888888888
8888888888888888888888888888888888888888
```
1369/1369 Umlegungen benötigt
— 0.2202 Sekunden —
Kommentierungen:
Bei solch einem großen Beispiel und einer so geringen reellen Laufzeit, wird deutlich, dass die Optimierungen des Brute-Force-Ansatzes greifen, .
Außerdem wird bereits deutlich das Schema der Idee klar. Die ersten Koeffizienten werden maximiert (F), während die Segmentumsatz mit den hinteren Ziffern ausgeglichen wird. Interessant dabei ist, dass bei diesem Beispiel offensichtlich deutlich mehr Segmente frei werden, als in den vorderen Ziffern gebraucht werden (in den hinteren Ziffern entstehen 8en). Dieser Beobachtung lässt sich womöglich verallgemeinern, da der Durchschnitt an Segmenten in der Darstellung einer Ziffer im Hexadezimalsystem, bei gleicher Verteilung der Ziffern, bei $\overline{s} = 5.25$ liegt und die Darstellung Ziffer der F aus 4 Segmenten besteht, ist es insgesamt wahrscheinlicher, dass Segmente weggenommen werden müssen, um eine Ziffer in ein F umzulegen.

4.3 Eigene Beispiele

hexmax_custom0.txt

Kommentierungen:
Zunächst sollten einige Grenzfälle untersucht werden. In dieser Eingabe ist keine Maximierung möglich ($\forall a_i : a_i = 15$):
```
FFF
100
```
Es greift der Fall, dass für jeden Koeffizienten die maximierenden Optionen $\mathbb{M}$ fehlschlagen und somit keine Vergrößerung von H stattfinden kann.
`python3 hexmax_custom0.txt`
Lösung:
`FFF`
0/100 Umlegungen benötigt
— 0.0004 Sekunden —

hexmax_custom1.txt

Kommentierungen:
In dieser Eingabe ist keine Maximierung möglich, da zu wenige Umlegungen vorhanden sind:
`4FF`

`1`

Es werden die entscheidenden Umlegungen des Koeffizienten a_2 Optionen ausprobiert, schlagen jedoch aufgrund von einer insuffizienten Anzahl an Umlegungen fehl.

`python3 hexmax_custom1.txt`

Lösung:

`4FF`

0/1 Umlegungen benötigt

— 0.0006 Sekunden —

hexmax_custom2.txt

Kommentierungen:

Das Phänomen aus *hexmax5.txt* lässt sich als weiteres Beispiel instrumentalisieren. Wenn in den vorderen Ziffern, viele 8en (1000x) vorhanden sind, und somit zwangsweise Segmente weggenommen werden müssen, um diese zu vergrößern, und zudem in den hinteren Ziffern lediglich eine 4 vorhanden ist, welche die Eigenschaft besitzt, dass jene nicht durch das Anlegen eines Segmentes zu einer validen Ziffer gemacht werden können, lässt sich mit $m = 1$ die Reallaufzeit im Vergleich zu *hexmax5.txt* ($n = 1000$, hier $n = 1001$) stark vergrößern:

```
888888888888888888888888888888888888888888
888888888888888888888888888888888888888888
8888888888888888888888888888888888888888888888888
8888888888888888888888888888888888888888888888888
8888888888888888888888888888888888888888888888888
8888888888888888888888888888888888888888888888888
8888888888888888888888888888888888888888888888888
8888888888888888888888888888888888888888888888888
8888888888888888888888888888888888888888888888888
8888888888888888888888888888888888888888888888888
8888888888888888888888888888888888888888888888888
8888888888888888888888888888888888888888888888888
8888888888888888888888888888888888888888888888888
8888888888888888888888888888888888888888888888888
8888888888888888888888888888888888888888888888888
8888888888888888888888888888888888888888888888888
8888888888888888888888888888888888888888888888888
8888888888888888888888888888888888888888888888888
8888888888888888888888888888888888888888888888888
8888888888888888888888888888888888888888888888888
8888888888888888888888888888888884
1
```

Dieses ineffiziente Verhalten liegt womöglich daran, dass trotz der Optimierung der Liste der *tempOptionen* (Unterabschnitt 1.6, die Iteration über die Schlüssel für jede 8 stattfindet, jedoch die Iteration über die Segmente vermieden wird. Eine mögliche Optimierung um solche Fälle zu vermeiden, wäre, ein Überspringen gleicher Ziffern, beim Fehlschlagen vom Umlegen. Wenn beispielsweise das Umlegen der ersten 8 in alle maximierenden Optionen $\mathbb{M}$ fehlschlägt, sollte direkt zur 4 übergegangen werden.

`python3 hexmax_custom2.txt`

Lösung:

```
888888888888888888888888888888888888888888
888888888888888888888888888888888888888888
8888888888888888888888888888888888888888888888888
8888888888888888888888888888888888888888888888888
8888888888888888888888888888888888888888888888888
8888888888888888888888888888888888888888888888888
8888888888888888888888888888888888888888888888888
8888888888888888888888888888888888888888888888888
8888888888888888888888888888888888888888888888888
8888888888888888888888888888888888888888888888888
8888888888888888888888888888888888888888888888888
8888888888888888888888888888888888888888888888888
```

```
888888888888888888888888888888888888888888888888
888888888888888888888888888888888888888888888888
888888888888888888888888888888888888888888888888
888888888888888888888888888888888888888888888888
888888888888888888888888888888888888888888888888
888888888888888888888888888888888888888888888888
888888888888888888888888888888888888888888888888
888888888888888888888888888888888888888888888888
888888888888888888888888888888884
```

0/1 Umlegungen benötigt
— 1.4360 Sekunden —

hexmax_custom3.txt

Kommentierungen:
Die Bedingung, dass die Darstellungen der Ziffern nicht komplett „geleert“ werden darf (Unterabschnitt 1.5) soll verifiziert werden:

```
1188
8
```

Die Lösung für diesen Input ist `FFFA`. Dabei ist jedoch relevant, dass für das erste F, nicht die zwei Segmente der zweiten 1 benutzt werden, und somit diese Darstellung geleert wird. Der in Unterabschnitt 1.5 Vorgang, dass das Segment innerhalb der zweiten 1 liegen bleibt, wird demonstriert. Dabei wird auch deutlich, dass durch die Reihenfolge der Umlegungen zuerst Segmente beim zweiten F angelegt werden. Ohne diese beschriebene Reihenfolge, würde die Darstellung der zweiten Ziffer vollständig geleert werden.

`python3 hexmax_custom2.txt`

Lösung:

`1188`

8/8 Umlegungen benötigt
— 0.0005 Sekunden —

5 Quellcode

```python
# Für Messung der Laufzeit
from time import time
# Zum Überprüfen ob Dateien exisitieren
```

```
from os.path import exists
# Zur Übergabe von Argumenten im Terminal
from sys import argv
# Für sehr große Eingaben, muss
# das Rekursionslimit hochgestellt werden (Maximale Rekursionstiefe)
# => Entspricht im Sachzusammenahang der Anzahl an Ziffern in der Eingabedatei
from sys import setrecursionlimit
setrecursionlimit(2000)


# Dictionary den Segmenten des Sieben-Segment-Displays (SSD) (von F bis 0)
# Zum Konvertieren einer Hexadezimalzahl (Key) in eine Liste mit den
# Segmenten, welche vorhanden sind (Value)
# => 1: Segment ist an; 0: Segment ist aus
# Indizes starten beim obersten Segment (0) und folgen dem
# Urzeigersinn => Index 6 ist das mittlere Segement (siehe Dokumentation)
hexInSSD = {
    "F": [1, 0, 0, 0, 1, 1, 1],
    "E": [1, 0, 0, 1, 1, 1, 1],
    "D": [0, 1, 1, 1, 1, 0, 1],
    "C": [1, 0, 0, 1, 1, 1, 0],
    "B": [0, 0, 1, 1, 1, 1, 1],
    "A": [1, 1, 1, 0, 1, 1, 1],
    "9": [1, 1, 1, 1, 0, 1, 1],
    "8": [1, 1, 1, 1, 1, 1, 1],
    "7": [1, 1, 1, 0, 0, 0, 0],
    "6": [1, 0, 1, 1, 1, 1, 1],
    "5": [1, 0, 1, 1, 0, 1, 1],
    "4": [0, 1, 1, 0, 0, 1, 1],
    "3": [1, 1, 1, 1, 0, 0, 1],
    "2": [1, 1, 0, 1, 1, 0, 1],
    "1": [0, 1, 1, 0, 0, 0, 0],
    "0": [1, 1, 1, 1, 1, 1, 0],
}
# Das Äquivalent für das Dezimalsystem (Erweiterung)
decInSSD = {
    "9": [1, 1, 1, 1, 0, 1, 1],
    "8": [1, 1, 1, 1, 1, 1, 1],
    "7": [1, 1, 1, 0, 0, 0, 0],
    "6": [1, 0, 1, 1, 1, 1, 1],
    "5": [1, 0, 1, 1, 0, 1, 1],
    "4": [0, 1, 1, 0, 0, 1, 1],
    "3": [1, 1, 1, 1, 0, 0, 1],
    "2": [1, 1, 0, 1, 1, 0, 1],
    "1": [0, 1, 1, 0, 0, 0, 0],
    "0": [1, 1, 1, 1, 1, 1, 0],
}
# Das Äquivalent für das Binärsystem (Erweiterung)
binInSSD = {
    "1": [0, 1, 1, 0, 0, 0, 0],
    "0": [1, 1, 1, 1, 1, 1, 0],
}
# Funktion für rekursives Vorgehen zum Maximieren einer Hexadezimalzahl im SSD
# maxUmlegungen: Umlegungen, die maximal getätigt werden dürfen
# hexZahl: Hexadezimalzahl, die umwandelt werden soll
# index: Index der aktuellen Ziffer in der Hexadezimalzahl (Standardmäßig 0)
# übrigerUmsatz: Segemente, die nach dem umwandeln übrig sind (Standardmäßig 0)
# schritte: Liste der Schritte/Umlegungen, die getätigt werden.
#     Schritt: [IndexAlt, SegmentIndexAlt, IndexNeu,
# SegmentIndexNeu] (Standardmäßig leer)
# tempOptionen: Liste an Optionen welche bereits ausprobiert wurden
und gescheitert sind (für Optimierung) (Standardmäßig leer)
# min: True, wenn die Zahl minimiert werden soll (Standardmäßig False)
def maxZiffer(maxUmlegungen, hexZahl, index=0, übrigerUmsatz=0,
    schritte=[], tempOptionen=[], min = False, inSSD = hexInSSD):
    # Check ob alle Ziffern umwandelt wurden => man ist am Ende der Hexzahl angekommen
    if index >= len(hexZahl):
        # Check ob Segmente übrig sind => Die Lösung ist nicht valide
        # => Es müssen alle Segmente verwendet werden
        if übrigerUmsatz != 0:
            return []
        # Die Lösung ist valide und die Schritte können zurückgegeben werden
        return schritte
```

```python
    # Check ob zu viele Segmente übrig sind (die Segemente können
    # keines Falls in den "hinteren" Ziffern untergebracht werden)
    # => Check ob der Umsatz größer ist, als es leere Segmente gibt
    if übrigerUmsatz > (7 * len(hexZahl[index:]))-sum([sum(inSSD[i]) for i in hexZahl[index:]]):
        # Die Anzahl der ''Lücken'' in der hinteren Ziffern ist
        # größer als die Anzahl der übrigen Segmente
        return []
    # Check ob zu viele Segement im Voraus verwendet wurden (die
    # Segmente können keines Falls von den "hinteren" Ziffern genommen werden)
    # => Check ob der Umsatz kleiner ist (negative Zahl), als es
    # gefüllte Segmente gibt, wenn in jeder Ziffer am Ende noch
    # mindestens zwei Segmente sein müssen (=1)
    if übrigerUmsatz < (-sum([sum(inSSD[i]) for i in hexZahl[index:]]) + 2*len(hexZahl[index:])):
        # Die Anzahl der ''Lücken'' in den bereits umegelegten
        # Ziffern ist größer als die Anzahl der übrigen Segmente in den hinteren Ziffern
        return []
    # Kopie der ausgeschiedenen Optionen um Mutation zu vermeiden
    tempOptionenNeu = tempOptionen.copy()
    # Festlegen der aktuellen Ziffer der Hexzahl
    ziffer = hexZahl[index]
    # Iteration über alle anderen Hexziffern von F bis 0
    for i in (reversed(inSSD.keys()) if min else inSSD.keys()):
        # Überprüfen, ob die Option bereits bei einer anderen Ziffer ausgeschieden ist
        if (ziffer, i) in tempOptionenNeu:
            continue
        # Check ob man bei der aktuellen Ziffer angekommen ist
        # Es folgen somit niedrigere Hexziffern
        # => nur fortfahren, wenn bereits Umlegungen getätigt worden sind,
        # Die erste Ziffer, die umgelegt wird, darf nicht verringert werden,
        # sonst wird die gesamte Hexadezimalziffer verringert (siehe Stellenwertsystem)
        if i == ziffer and (len(schritte) == 0 or übrigerUmsatz == 0):
            # Die aktuelle Ziffer bleibt unverändert ... es wird mit der nächsten fortgefahren
            return maxZiffer(maxUmlegungen, hexZahl,index+1, übrigerUmsatz, schritte,
                tempOptionen=tempOptionenNeu,inSSD=inSSD,min=min)
        # Die aktuell übrigen Segmente entsprechen dem Segmentumsatz
        übrigeSegmente = übrigerUmsatz
        # Kopie der Schritte um Mutation zu vermeiden
        schritteNeu = schritte.copy()
        # Iteration über alle Segmente der Ziffern
        for segment in range(7):
            # Check ob das Segment von i in der Ausgangsziffer fehlt
            if inSSD[i][segment] > inSSD[ziffer][segment]:
                # Check ob Segmente übrig sind
                if übrigeSegmente > 0:
                    # Es gibt noch Segmente, die verwendet werden können
                    # die "Zielposition" (Ziffernindex,
                    # Segmentindex) der übrigen Segmente wird beim
                    # entsprechenden Segment hinzugefügt
                    # (war vorher noch nicht bestimmt) (Siehe Format der schritte-Liste)
                    schritteNeu[len(schritteNeu)-übrigeSegmente][2] = index
                    schritteNeu[len(schritteNeu)-übrigeSegmente][3] = segment
                    # Es wird keine neue Umlegung gebraucht
                else:
                    # Es gibt keine Segmente, die verwendet werden können
                    # Es muss eine neue Umlegung mit unbestimmter
                    # Herkunftsposition hinzugefügt werden
                    schritteNeu.append([None, None, index, segment])
                # Ein "übrigesSegment" wird verwendet (selbst wenn keine übrig sind =>
                # möglicherweise kann es in der nächsten Ziffer erzeugt werden)
                übrigeSegmente -= 1
            # Check ob ein Segment von i in der Ausgangsziffer zu viel ist
            elif inSSD[i][segment] < inSSD[ziffer][segment]:
                # Check ob mehr Segmente verwendet wurden, als frei geworden sind
                if übrigeSegmente < 0:
                    # Es gibt Umlegungen mit unbestimmter Herkunftsposition
                    # Die Herkunftsposition wird auf die aktuelle
                    # Position (Ziffernindex, Segmentindex) gesetzt
                    # Das aktuelle Stäbchen wird für den letzten Schritt, ohne
                    # Herkunftsposition verwendet,
                    # Damit die Darstellung von "ziffer" nicht komplett geleert wird (siehe Doc)
                    neueUmlegung = schritteNeu.pop(len(schritteNeu)-1)
                    neueUmlegung[0] = index
                    neueUmlegung[1] = segment
```

```
                    schritteNeu.insert(len(schritteNeu)+übrigerUmsatz,neueUmlegung)
                    # Es wird keine neue Umlegung gebraucht
                else:
                    # Es gibt keine Umlegungen mit unbestimmter Herkunftsposition
                    # Es muss eine neue Umlegung mit unbestimmter
                    # Zielposition hinzugefügt werden
                    schritteNeu.append([index, segment, None, None])
                # Ein "übrigesSegment" wird hinzugefügt
                übrigeSegmente += 1
            # Die Umformung ist nicht möglich, wenn die übrige Umlegungen nicht genug sind
            # (Es darf nicht mehr schritte als Umlegungen geben)
            if len(schritteNeu) > maxUmlegungen:
                tempOptionenNeu.append([ziffer,i])
                break

        else:
            # Wird ausgeführt wenn nicht gebreakt wurde
            # Es kann mit der nächsten Ziffer fortgefahren werden
            # Leere tempOptionen, wenn die Ziffer umgelegt wurde,
            # weil dadurch möglicherweise neue Optionen möglich werden
            result = maxZiffer(maxUmlegungen, hexZahl,
                               index+1, übrigeSegmente,
                                schritte=schritteNeu,
                                tempOptionen=([] if i != ziffer else tempOptionenNeu),
                                inSSD=inSSD, min=min)

            # Wenn eine Lösung gefunden wurde, wird diese zurückgegeben
            # Es ist die grö ß tmögliche, da von F nach 0 iteriert wird
            if len(result) > 0:
                return result
            else:
                #print(tempOptionenNeu)
                # Andernfalls wird die gewählte Option zu den nicht
                # gefundenen Optionen hinzugefügt
                tempOptionenNeu.append((ziffer,i))
    # Es wurde keine Lösung gefunden
    return []

# Funktion zur Initialisierung des Lösungsprozesses und Anzeige
# hexZahl: Hexadezimalzahl, welche maximiert werden soll (String)
# maxUmlegungen: maximale Anzahl an Umlegungen
# zwischenstandAnzeige: True, wenn die Zwischenstände angezeigt werden sollen
# min: True, wenn die hexZahl minimiert statt maximiert werden soll
def maximieren(hexZahl, maxUmlegungen, zwischenstandAnzeige=False, min=False, inSSD=hexInSSD):
    # Ermitteln der nötigen Umlegungen zur Maximierung der
    # Hexadezimalzahl mithilfe von "maxZiffer"
    ergebnis = maxZiffer(maxUmlegungen, hexZahl, min=min, inSSD=inSSD)
    # Ausgabe des Zwischenstandes in roher Form
    #print("\nZwischenstand (jede Subliste steht für eine Ziffer):\n\n" + str(ssd) + "\n")
    # Maximale Hexadezimalzahl muss aus den Umlegungen "zurückgewonnen" werden
    ergebnisString = ""
    # Check ob überhaupt Umlegungen getätigt wurden
    if len(ergebnis) > 0:
        # Es wurden Umlegungen getätigt
        # Initialisierung der SSA (Liste von Datstellungen von Ziffern)
        ssd = [inSSD[i].copy() for i in hexZahl]
        # Ausagbe der Starthexzahl wenn gewünscht
        if zwischenstandAnzeige:
            printSSD(ssd)
        # Iteraion über die ermittleten Umlegungen
        for schritt in ergebnis:
            # Durchführen der Umlegung
            ssd[schritt[0]][schritt[1]] = 0
            ssd[schritt[2]][schritt[3]] = 1
            # Ausgabe des Zwischenstandes in roher
            # Form
            #print("\n" + str(i+1) + ": " + str(ssd) + "\n")
            # Ausgabe der SSA wenn gewünscht
            if zwischenstandAnzeige:
                printSSD(ssd)
        # Iteration über alle Ziffern in der SSA
        for anzeige in ssd:
            # Hinzufügen der Ziffer im Stringformat (Umformung über die Dictionary s.o.)
```

```
                ergebnisString += list(inSSD.keys())[list(inSSD.values()).index(anzeige)]
        else:
            # Es wurden keine Umlegungen getätigt
            # Es ist bereits die maximale Hexadezimalzahl
            ergebnisString = hexZahl

    # Zurückgeben der Lösung und der benötigten umlegungen
    return ergebnisString, len(ergebnis)

# Funktion zum Lesen des Inputs
def parseInput():
    # Überprüfung ob kein Argument angegeben wurde
    if(len(argv) == 1):
        # Fragen nach Eingabedatei
        file = input("Eingabedatei eingeben:")
    else:
        # Lesen der Eingabedatei aus den Argumenten
        file = argv[1]
    # Überprüfung ob die Eingabedatei existiert
    if(not exists(file)):
        # Ausgabe eines Fehlers
        print("\033[1;31mDatei nicht gefunden\033[0m")
        return None
    # Öfnnen der Eingabedatei (im Lesemodus)
    with open(file=file, mode="r") as data:
        # Lesen aller Zeilen der Eingabedatei
        inhalt = data.readlines()
        # Bereinigen der Zeilen (Zeilenumbrüche entfernen)
        inhalt = [i.replace("\n", "") for i in inhalt]
        # Hexadezimalzahl
        hexZahl = inhalt[0]
        # Maximalzahl an Umlegungen
        m = int(inhalt[1])
        # Zurückgeben der gelesenen Daten (...und File, für Benennung der Ergebnisdatei)
        return hexZahl, m, file

# Hauptfunktion (Ausführung des Programms)
def main():
    # Sicheres Lesen des Inputs
    try:
        hexZahl, m, file = parseInput()
    except Exception as e:
        # Ausgabe des Fehlers
        print("Input konnte nicht gelesen werden: {}".format(e))
        return
    # Lesen der Flagge -d für Zwischenstand anzeigen
    zwischenstandAnzeige = "-d" in argv
    # Lesen der Flagge -min für Minimierung statt Maximierung
    min = "-min" in argv
    # Lesen der Flagge -bin für binäres Zahlensystem
    binary = "-bin" in argv
    # Lesen der Flagge -dec für dezimales Zahlensystem
    decimal = "-dec" in argv
    # Maximieren der eingelesenen Hexadezimalzahl mit maximal m Umlegungen
    lösung, umlegungen = maximieren(hexZahl, m,
        zwischenstandAnzeige, min=min,
        inSSD=(binInSSD if binary else (decInSSD if decimal else hexInSSD)))
    # Schreiben der Lösungsdatei
    with open("ergebnis_" + file, "w") as f:
        # Ausgabe der Lösungszahl
        print("Lösung:", lösung)
        # Ausgabe der benötigten Umlegungen
        print("{}/{} Umlegungen benötigt".format(umlegungen, m))
        # Schreiben der Lösung
        f.write(lösung + "\n" + "{}/{}".format(umlegungen, m))
```