Aufgabe 3: Hex-Max

Teilnahme-ID: 61743

Bearbeiter dieser Aufgabe:
Jonas Fennekohl

13. April 2022

1. Lösungsidee: 1
1.1 Modellierung der Situation: 2
1.2 Funktionsweise von Umlegungen: 2
1.3 Mathematischer Idee hinter der Lösung: 2
1.4 Errechnen der Vertauschungen: 3
1.5 Feststellen der besten Vertauschung: 4
1.6 Zusammenfügen der Algorithmen: 6
2. Umsetzung 7
2.1 Speicherung der Vertauschungsmöglichkeiten: 7
2.2 Die Methode hexZahlMaximieren: 8
2.3 Die Methode rekursivHexZahlMaximieren: 8
2.4 Ausgeben der Umlegungen: 9
2.5 Berechnung der Laufzeit: 11
Beispiele 11
Quellcode 19

1. Lösungsidee:

In der Aufgabenstellung haben wir eine hexadezimale Zahl gegeben mit *n* Ziffern und eine Anzahl an möglichen Umlegungen *m*. Das Ziel ist hierbei die gegebene Zahl mit den gegebenen Umlegungen zu maximieren.

Die einzelnen Ziffern des Hexadezimalsystems sind in dieser Aufgabenstellung als Kombination aus Strichen dargestellt. Dabei gibt es vorbestimmte Plätze, wo diese Striche liegen können.

Abbildung 1: Darstellung der Hexadezimalzahlen

1.1 Modellierung der Situation:

Wir können in diesem Beispiel eine Hexadezimalziffer als ein Array aus verschiedenen Boolean/Wahrheitswerten modellieren. Dieses Array hat eine Größe von 7, also die Menge an Positionen, an denen solche Striche liegen können.

Rechts kann man sehen, welchen Index eine Position in unserem Array hat. Diese Modellierung resultiert in einem Array, welche ungefähr so aussieht:

0	1	2	3	4	5	6
$true$	$true$	$false$	$true$	$true$	$false$	$true$

Abbildung 3: Darstellung der Ziffer 2 als Array

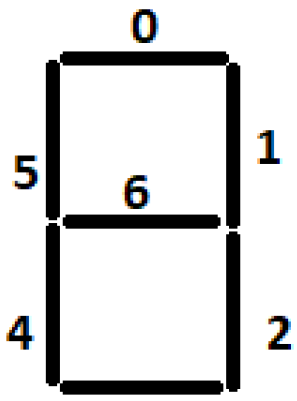

Abbildung 2: Modellierung der Plätze

1.2 Funktionsweise von Umlegungen:

Umlegungen werden von dem Text als das Umlegen von einem Strich an eine andere Position bezeichnet. Dabei können die Striche auch zwischen den Ziffern umgelegt werden(Siehe https://www.einstieg-informatik.de/community/forums/topic/812/40-2-a3-umlegungen). Man kann also insgesamt m Umlegungen machen, wobei die Gesamtzahl der Striche gleichbleiben muss.

Eine Umlegung würde sich in unserer Modellierung als Array also dadurch auszeichen, an einer Position einen Wert true zu false zu ändern und an einer anderen beliebigen Stelle einen Wert von false zu true.

1.3 Mathematischer Idee hinter der Lösung:

Um kurz festzuhalten, was wir bereits über die Lösung wissen. Wir haben eine Hexadezimalzahl mit n Ziffern und m Umlegungen, welche über die ganze Zahl benutzt werden können. Unser Ziel ist es hier, den Wert der Hexadezimalzahl zu maximieren bei gleichbleibender Anzahl der Ziffern und Striche.

Man konvertiert eine Hexadezimalzahl in eine Dezimalzahl, indem man den

$$Wert_{Ziffer\ an\ Stelle\ n} \cdot 16^n + Wert_{Ziffer\ an\ Stelle\ n-1} \cdot 16^{n-1} + \ldots . Wert_{Ziffer\ an\ Stelle\ 0} \cdot 16^0$$

Die Ziffern der Hexadezimalzahl so verrechnet. Aus der Rechenweise ergibt sich auch der Lösungsansatz für die Aufgabe. Nehmen wir an wir hätten eine Situation, wo wir entweder

a) die Ziffer an der Stelle n um 1 Wert erhöhen können oder

b) sämtliche nachfolgende Ziffern(n-1 bis 0) von 0 zu F maximieren können.

Die Differenz zwischen diesen Werten ist immer ≥1, da wenn $b) + 1 \leq a)$. Das Worst-Case Szenario ist hierbei das man als Ausgangszahl 00000 hat und entweder F0000 oder 0FFFF daraus machen kann. Der mathematische Beweis funktioniert wie folgt:

So sieht die Formel für die Gegenüberstellung aus in unsere Annahme aus: $16^n - 1 = \sum_{a=0}^{n-1} 15 \cdot 16^a$

Daraus ergibt sich: $16^n = 15 \cdot 16^{n-1} + 15 \cdot 16^{n-2} \ldots .15 \cdot 16^0$

Vereinfachen wir zuerst: $16^n = 15 \cdot 16^{n-1} \rightarrow 16 \cdot 16^{n-1} = 15 \cdot 16^{n-1}$

Bei dieser Gleichung ergibt sich eine Differenz von: 16^{n-1}

Vereinfachen wir weiter: $16^{n-1} = 15 \cdot 16^{n-2} \rightarrow 16 \cdot 16^{n-2} = 15 \cdot 16^{n-2}$

Dies wiederholt sich solange bis man am Ende angekommen ist:

$$16^1 = 15 \cdot 16^0 \rightarrow 16 = 15 \rightarrow Differenz = 1$$

Und deshalb besteht zwischen den beiden Optionen selbst im Worst-Case eine Differenz von 1.

Dies ist natürlich kein realistisches Szenario, weshalb in unserer Aufgabenumgebung die Differenz immer deutlich größer ausfallen wird. Trotzdem lässt sich daraus ableiten, dass es immer besser ist, die Zahlen, die links stehen, also wenn man die Formel $Wert_{Ziffern} \cdot 16^n$ betrachtet die Zahlen mit einem möglichst großen *n* möglichst stark zu maximieren.

1.4 Errechnen der Vertauschungen:

Nachdem mathematisch bewiesen ist, dass es am besten ist, zu versuchen die Ziffer mit einem möglichst hohen n möglichst stark zu maximieren müssen wir das jetzt noch in eine insgesamt gültige Lösung zusammenfügen.

Das nächste Problem, vor dem wir stehen ist, wie festgestellt werden soll, was eine Vertauschung ist. Eine Vertauschung wird hierbei als das Umwandeln von einer Ziffer zu einer anderen durch eine Kombination aus verschiedenen Umlegungen. Eine solche Vertauschung ist wie folgt aufgebaut:

Gegeben: Ausgangsziffer

Vertauschung = {Position 1, Position 2, Position 3, Position 4}

1. Ziffer in die getauscht wird
2. Anzahl der Umlegungen, die insgesamt benötigt werden, ob innerhalb der Zahl umgelegt wird oder ein Strich aus einer anderen Zahl hinzugefügt oder zu einer anderen Zahl gelegt wird.
3. Anzahl der Umlegungen, welche aus anderen Ziffern zu dieser kommen.
4. Anzahl der Striche, welche von dieser Ziffer zu anderen gehen,

Aufgrund der Art und Weise wie diese Menge konstruiert ist, steht entweder bei 3 ein Wert und bei 4 0 oder andersherum, aber nie in beiden Sachen ein Wert, da diese Umlegungen auch in der Zahl stattfinden können, ohne 2 Umlegungen zu brauchen.

Eine Vertauschung errechnen wir, indem wir uns die Unterschiede zwischen den Listen zweier Ziffern anschauen. Dabei haben wir eine Liste einer Ziffer, die als Ausgangsziffern genutzt wird und eine Liste, welche als Endziffer benutzt wird. Die Vertauschung wird durch zwei Werte berechnet: EinsNull und NullEins. Die Funktionsweise dieses Algorithmus lässt sich am besten an Beispielen erklären.

0	1	2	3	4	5	6
$true$	$true$	$true$	$false$	$true$	$true$	$true$

Abbildung 4: Darstellung von A

0	1	2	3	4	5	6
$true$	$true$	$false$	$true$	$true$	$false$	$true$

Abbildung 5: Darstellung von 2

Hier haben wie die Listendarstellung der beiden Ziffern A und 2. A ist hierbei die Ausgangsziffer und 2 die Endziffer. Der Wert EinsNull hält fest, wie oft bei der Ausgangsziffer true steht, wenn bei der Endziffer false steht. Der Wert NullEins hält fest, wie oft bei der Ausgangsziffer false steht, wenn bei der Endziffer true steht. Für unser Beispiel wären diese Werte: EinsNull = 2, NullEins = 1.

Der Wert für 2 ist der höhere Wert von EinsNull oder NullEins, da insgesamt so viele Umlegungen gebraucht werden. Dem ist so, da sich, wenn z.B. wie hier EinsNull > NullEins ist, die Umlegungen für EinsNull die für NullEins abdecken und die überzähligen Striche von anderen Zahlen kommen/übrig bleiben. Die Unterschiede müssen ja entfernt werden und daher ist die kleinere Zahl in der größeren abgedeckt ist. Wenn beide gleich hoch sind, ist einer der beiden Werte der Wert für 2 und 3 und 4 sind beide 0, da keine Striche zusätzlich gebraucht werden/übrig bleiben.

Der Wert für 3 oder 4 wird auch darüber entschieden, welcher Wert von EinsNull oder NullEins größer ist. Wenn EinsNull größer ist, wird die Differenz zwischen den beiden Werten in 4 reingeschrieben und wenn NullEins größer ist, wird die Differenz der Werte in 3 eingetragen. Der andere Wert von 3 oder 4 ist dann immer 0.

Nach diesem Schema ist die Liste für die Vertauschung von $A \rightarrow 2$ {2,2,0,1}

1.5 Feststellen der besten Vertauschung:

Um die bestmögliche Vertauschung von einer Zahl zu finden, wird zuerst geguckt, ob man die Zahl in die höchste Zahl vertauschen kann(F), dann in die zweithöchste(E). Man kann sich die möglichen Vertauschungen von 6 →? also als Liste vorstellen: Vertauschungen für 6 = {1,2,…,E,F}. In dieser Liste sind dann die Vertauschungen für die Zahl 6 verlegt mit den Informationen, wie man $6 \rightarrow F$ tauscht. In dem Prozess geht man dann die Liste von rechts nach links durch(von der höchsten Ziffer zur niedrigsten) um die beste Vertauschung zu finden.

Dieser Prozess geht so weiter, bis man entweder einen validen Tausch gefunden hat oder der Algorithmus bei der Ursprungszahl ankommt. Das heißt dann das kein möglicher Tausch den Wert der Zahl erhöhen würde.

Dabei stellt sich die Frage, wie überprüft wird, ob eine Vertauschung valide ist, also durchgeführt werden kann. Um die Frage zu beantworten, muss ich erst das Konzept eines Zwischenspeichers einführen. In unserer Menge, die eine Vertauschung darstellt, befinden sich auch Werte dafür, wie viel Striche zusätzlich benötigt werden und wie viele übrig bleiben. Unser Zwischenspeicher nimmt diese Werte auf. Der Zwischenspeicher ist letztendlich also einfach ein Platz, wo gerade nicht

verwendete Striche für die spätere Verwendung aufbewahrt werden. Wenn der Wert unseres Zwischenspeichers positiv ist, bedeutet das, dass so viele Striche noch in die Zahl eingesetzt werden müssen, um wieder auf die Ausgangsmenge an Striche zu kommen. Wenn er negativ ist, bedeutet es das noch so viele Striche aus der Zahl entfernt werden müssen. Der Zwischenspeicher hat beim Start des Algorithmus einen Wert von 0 und muss beim beenden auch wieder einen Wert von 0 haben. Wenn unser Zwischenspeicher ungleich 0 ist, muss das Programm weiterlaufen.

Wenn wir uns die Kombination aus allen Einzelteilen als fertigen Algorithmus anschauen, werde ich noch einmal darauf eingehen, warum der Zwischenspeicher nicht die Regeln von Umlegungen oder die Bedingung das eine Zahl nie ganz geleert wird erfüllt und umsetzt.

Kommen wir jetzt dazu wie überprüft wird, ob ein Tausch valide ist. Dafür habe ich ein Flussdiagramm erstellt.

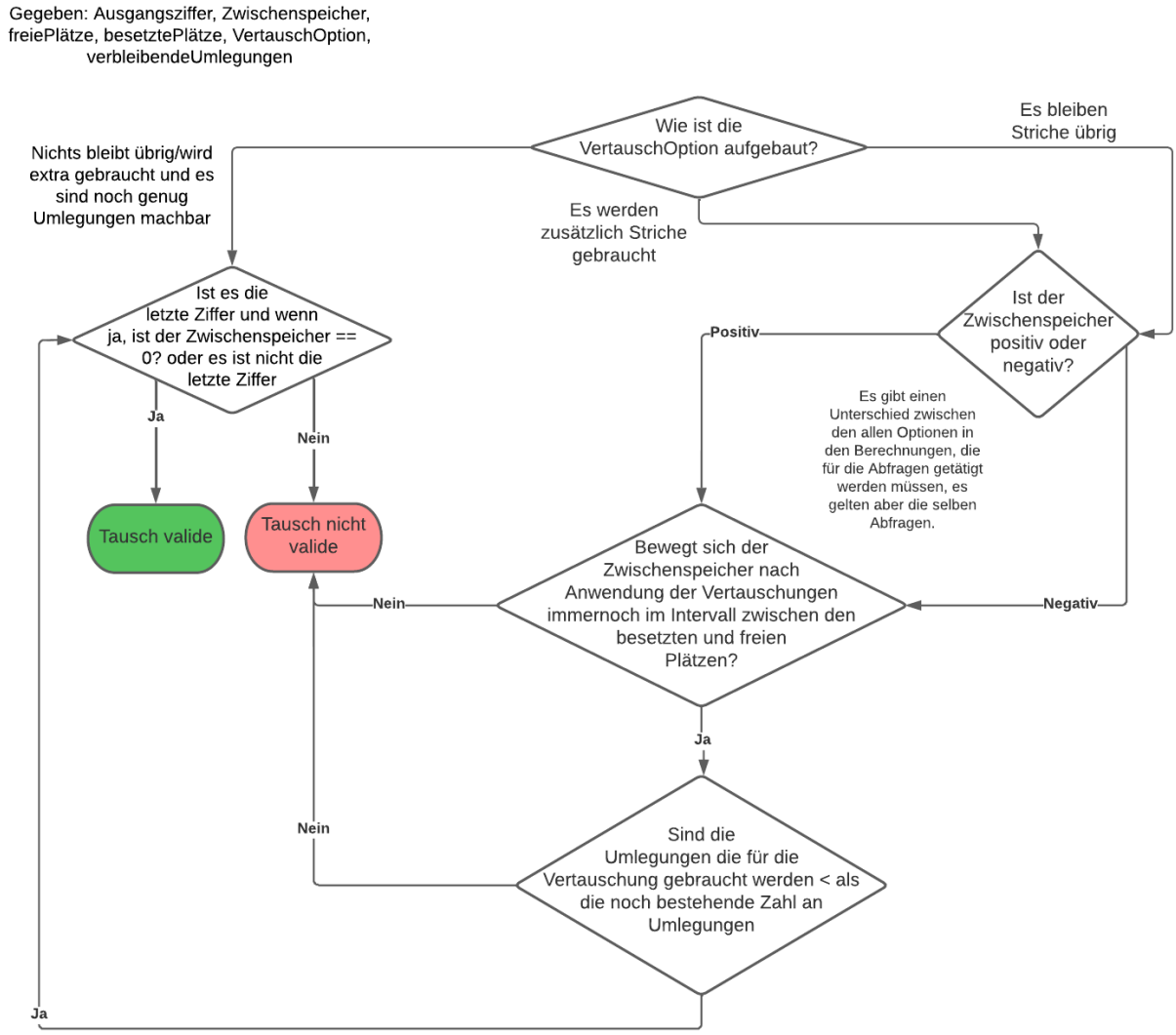

Abbildung 6: Flussdiagramm zur Entscheidung einer validen Vertauschoption

Im Flussdiagramm können wir sehen, wie wir feststellen, ob eine Vertauschoption valide ist, wenn die Vertauschoption, die Anzahl der freien Plätze, die Anzahl der besetzten Plätze, den Zwischenspeicher und die verbleibenden Umlegungen gegeben sind.

Zuerst wird entschieden, welche Art von Vertauschung wir haben, eine Vertauschung innerhalb der Zahl, ohne das Dinge übrigbleiben oder zusätzlich benötigt werden, eine Vertauschung bei welcher zusätzliche Striche benötigt werden oder eine Vertauschung, bei welcher Striche übrig bleiben. Je nachdem welche Entscheidung getroffen wird werden dann verschiedene Berechnungen durchgeführt, um die Werte zu errechnen an denen nachher entschieden wird, ob die Vertauschung valide ist. Dabei werden bei allen drei Optionen die durch die Vertauschung entstehenden freie Plätze und besetzte Plätze neu berechnet. Außerdem wird auch die Anzahl der übrigbleibenden Umlegungen berechnet. Wenn entweder neue Striche gebraucht werden oder Striche übrig bleiben, berechnen wir vorher auch wie viel davon auf den Zwischenspeicher ausgelagert werden kann, erst danach wird dann die Anzahl der verbleibenden Umlegungen berechnen.

Nach den Rechnungen wird überprüft, ob der Zwischenspeicher nach der Vertauschung zwischen den freien Plätzen und den besetzten Plätzen liegt. Hier liegt auch die Funktion der beiden Werte, da sie sicherstellen sollen, dass keine Variante berechnet wird, in der es nicht mehr möglich ist, die Hex-Zahl auszugleichen. Dadurch müssen weniger Berechnungen durchgeführt werden. Nach dieser Abfrage wird außerdem noch überprüft, ob genug Umlegungen für die Vertauschung vorhanden sind. Falls die zu vertauschende Ziffer am Ende der Hexadezimalzahl liegt wird außerdem überprüft, ob am Ende der Vertauschung der Zwischenspeicher bei 0 liegt.

Wenn alle diese Bedingungen erfüllt sind, ist der Tausch valide, sonst nicht.

1.6 Zusammenfügen der Algorithmen:

Am Anfang dieses Algorithmus haben wir eine Hexadezimalzahl mit n Stellen und eine Maximalzahl m der Umlegungen gegeben. Der Algorithmus fängt dann an die Hex-Zahl zu maximieren. Dabei geht er rekursiv vor. Zuerst errechnet er die mit den gegebenen Umlegungen, die Zahl mit den bestmöglichen Vertauschungen, die erstellt werden, könnte. Dies tut er indem er die beste Vertauschung für die n-te Stelle sucht(ganz links) und sich dann nach rechts durch die Zahl arbeitet und dort dann wieder die bestmögliche Vertauschung findet. Dies führt er bis zum Ende der Zahl durch.

Wenn der Algorithmus am Ende der Zahl angekommen ist und der Zwischenspeicher != 0 ist, also entweder mehr oder weniger Striche in der Zahl enthalten sind als in der Ausgangszahl, nimmt er für die letzte Zahl die zweitbeste Option. Wenn danach der Zwischenspeicher immer noch nicht 0 ist, geht er die drittbeste Option durch etc. Das macht der Algorithmus so lange, bis entweder eine valide Lösung gefunden wurde oder alle möglichen Vertauschungen durchgegangen wurden. Wenn alle möglichen Vertauschungen durchgegangen wurden, geht der Algorithmus zurück zur Stelle davor und führt dasselbe Prinzip dort durch.

Es gibt auch noch ein paar Randfälle, die in meinem Algorithmus abgedeckt sind. Wenn der Zwischenspeicher zu einem beliebigen Zeitpunkt entweder gleich den besetzten Plätzen oder den freien Plätzen ist, stoppt mein Algorithmus und ersetzt den Rest der Zahl mit entweder 8(freie Plätze) oder 1(besetzte Plätze).

Am Ende des Algorithmus wird die Endzahl mit der Anfangszahl verglichen, wenn die Endzahl größer ist als die Anfangszahl, wurde die Zahl maximiert, wenn sie kleiner oder gleich ist, ist keine Maximierung möglich.

Ich hatte vorher schon erwähnt, dass mit diesem Algorithmus sichergestellt wird, dass die Bedingungen von Umlegungen erfüllt werden und nie eine Zahl ganz geleert wird. Die Bedingung hinter einer Umlegung ist, dass die Anzahl der Striche gleichbleiben, am Ende jede Stelle der Zahl eine tatsächliche Ziffer steht und dass ein Strich von seiner Position, an eine freie andere Position bewegt wird. Diese Bedingung wird dadurch erfüllt, dass unser Zwischenspeicher am Ende auch nur Striche umlegt, dies aber verzögert tut. Das heißt man führt zuerst immer einen der beiden Schritte aus, also entweder einen Strich dahin zu legen oder irgendwo weg zu nehmen. Damit unsere Zahl als valide gezählt wird muss der Zwischenspeicher ja = 0 sein, also auf dem Ausgangsniveau. Deshalb ist garantiert, dass die Anzahl der Striche gleichbleibt, weshalb jede Umlegung richtig ausgeführt ist, weil jeder Strich der irgendwo weg genommen wurde deshalb auch woanders hingelegt wurde.

Die andere Bedingung die mein Algorithmus/Programm erfüllen musste war, dass nie eine komplett leere Stelle erzeugt wird. In meinem Algorithmus ist das unmöglich, weil Umlegungen nicht einzeln stattfinden, sondern nur um Zahlen direkt zu tauschen. Dadurch sind immer durchgehend in einer Zahl mindestens zwei Striche vorhanden, da es keine Ziffer gibt, welche komplett leer ist, in die mein Algorithmus eine Vertauschung überhaupt überprüft.

2. Umsetzung

2.1 Speicherung der Vertauschungsmöglichkeiten:

Da die Ziffern im Hexadezimalsystem festgelegt sind und damit auch die Vertauschungen von einer Ziffer in die andere Ziffer festgelegt sind, kann man diese im Programmcode selbst speichern.

Die einzelnen Vertauschungen sind ja als Liste gespeichert, in welcher nur ganze Zahlen mit einer festen Länge vorkommen. Das bedeutet, dass man diese Liste als int[] sehen kann, mit der Länge 3. Für jede einzelne der 16 Ziffern im Hexadezimalsystem gibt es 16 Vertauschungen, welche beachtet werden müssen, sogar die Vertauschung von derselben Ziffer zu sich selbst, damit diese auch berücksichtigt werden, um die Zahl zu maximieren, da das die Stelle gleich bleibt auch die beste Möglichkeit seien kann. Es gibt also insgesamt 256 Vertauschungen, die abgespeichert werden müssen. Dabei sind in jeder Vertauschung 4 Integer vorhanden, das bedeutet das insgesamt 1024 Integer abgespeichert werden müssen, um die Vertauschungsmatrix abzuspeichern, welche für die Aufgabe gebraucht wird.

Damit spart man sich jedes Mal die einzelnen Vertauschungen zu errechnen und optimiert damit die Laufzeit. Die Vertauschungsmatrix ist in der Programmiersprache Java als dreidimensionales Integer Array abgespeichert in den Programmcode geschrieben. Dabei werden insgesamt 4096 Bytes(4.096 KB) Arbeitsspeicher gebraucht, um die Integer abzuspeichern. Dies ist aus meiner Sicht effizienter, da man sich mehrere Male die Zeit spart, jedes Mal die Matrix neu zu berechnen, obwohl sie konstant gleich ist, für die Kosten für sehr wenig Arbeitsspeicher. Dies war einer der ersten Optimierungen.

2.2 Die Methode hexZahlMaximieren:

Die Methode hexZahlMaximieren nimmt zuerst die Hexadezimalzahl als int[] entgegen und auch die übrigen Umlegungen als pUebrigeUmlegungen Integer. Dafür wird vorher die Zahl von einem String in ein int[] konvertiert, wobei die Buchstaben ihren respektiven Wert annehmen. Diese Konvertierung findet statt, da die Werte von 0-15 sind, genau den Indexen entsprechen, unter welchen z.B. die Vertauschungen in der Matrix abgespeichert sind. Dann werden in der Methode die in der Zahl freien Plätze und besetzte Plätze berechnet. Die freien Plätze sind alle Plätze, welche keinen Strich haben/im Array einen Wert von false annehmen. Die besetzten Plätze sind alle Plätze, wo Striche vorhanden sind. Davon werden immer 2 pro Stelle abgezogen, da immer mindestens 2 in der Zahl vorhanden seien müssen, um eine 1 zu bilden (Ziffer mit der geringsten Anzahl an Strichen).

Danach werden diese Werte zusammen mit den übrigen Umlegungen und dem Zwischenspeicher, welcher 0 ist and die Methode rekursivHexZahlMaximieren übergeben. In dieser wird dann die bestmögliche Zahl errechnet. Nachdem die Zahl errechnet wurde, werden alle dafür nötigen Vertauschungen in der ArrayList speicherVertauschungen gespeichert. Mit dieser rekonstruiert die Methode die bestmögliche Zahl. Ich habe diesen Weg aus 2 Gründen gewählt. Da mein Programm mit einem Zwischenspeicher arbeitet, der wenn er einen bestimmten Wert erreicht, den Rest der Zahl entweder mit 1 oder 8 auffüllt. Außerdem muss man für einen Teil der Beispiele auch die Zwischenergebnisse konstruieren. Dabei brauche ich auch die Liste der Vertauschungen, da mein Programm dies nicht so kann, sondern dafür eine extra Methode gebraucht wird.

Die Vertauschungen werden dann ausgeführt und falls eine der beiden Grenzen erreicht wird, füllt die Methode den Rest der Zahl auf. Danach wird die maximierte Hexadezimalzahl als int[] zurückgegeben.

2.3 Die Methode rekursivHexZahlMaximieren:

Diese Methode implementiert den Algorithmus, welcher in 1.6 beschrieben wurde. Sie wird in der Methode hexZahlMaximieren aufgerufen, um die Vertauschungen zu berechnen, die zur Maximierung benötigt werden.

Die Methode gibt einen Boolean zurück, welcher true ist falls ein valide Maximierung gefunden wurde oder false wenn es einen Fehler gibt. Als Parameter werden der Methode folgende Werte übergeben:

- int[]hexZahl, Hexadezimalzahl in den respektiven Werten der einzelnen Ziffern.
- int[][] tauschMatrix, Ausschnitt der in 2.1 beschriebenen Matrix, welche die Vertauschungen für alle Zahlen im Hexadezimalsystem festhält. Ausschnitt ist der Teil der Matrix, welche die Vertauschungen für die Ziffer an der aktuellen Stelle der hexZahl angibt. Abfrage: Vertauschungsmatrix[hexZahl[aktuelle Stelle]](zeigt einen der Gründe, warum das Programm die Hexadezimalzahlen in Integern betrachtet.)
- int verbleibendeUmlegungen, die verbleibenden Umlegungen, an der Stelle der hexZahl wo der Algorithmus gerade ist
- int freiePlaetze, freie Plätze in der hexZahl von dieser Stelle bis zum Ende

- int uebrigePlaetze, besetzte Plätze in der hexZahl von dieser Stelle bis zum Ende: Wert von besetzte Plätze nur unter anderem Namen
- int zwischenspeicherVertauschungen, aktueller Stand des Zwischenspeicher(+/-)
- int index, Stelle in der hexZahl, wo sich der Algorithmus gerade befindet, von links nach rechts gezählt.

Dann geht der Algorithmus das Array tauschMatrix vom Ende zum Anfang durch, da der Algorithmus immer zuerst die Ziffern mit dem höchsten Wert überprüft($F \rightarrow 0$). Wenn eine Vertauschung valide ist(siehe 1.5 wie die beste Vertauschung festgestellt wird), dann ruft sich die Methode selbst mit den neuen Werten für die Parameter auf. Um diese Werte zu berechnen, werden neue Instanzen von diesen Werten initialisiert(pFreiePlaetze, pBesetztePlaetze etc.), damit diese Vertauschung nicht zu einem richtigen Ergebnis führt, ohne große Probleme mit denselben Werten weitergerechnet wird, ohne dass sie wieder zurückgerechnet werden müssen. Die Methode ruft sich so lange selbst auf, bis sie am Ende der Zahl angekommen ist. Nach der letzten Vertauschung überprüft die Methode dann, ob der Zwischenspeicher = 0 ist. Dies ist die Abbruchbedingung des rekursiven Aufrufs. Wenn kein valider Tausch errechnet werden kann, gibt die Methode am Ende false zurück.

Wenn ein valides Ergebnis errechnet wurde, schließen sich die ganzen Methodenaufrufe, indem sie auch true zurückgeben. Davor wird die Vertauschung, welche nötig ist, um an die Zahl zu kommen, in der ArrayList speicherVertauschungen abgespeichert.

Diese Liste wird dann in der Methode hexZahlMaximieren ausgelesen und die Vertauschungen angewendet.

2.4 Ausgeben der Umlegungen/des Zwischenstandes:

Um die Umlegungen auszugeben, wird auch die Liste speicherVertauschungen benutzt. Die Vertauschungen werden dabei nacheinander ausgelesen und angewendet indem man sie kombiniert. Da mein Programm mit einem Zwischenspeicher arbeitet, werden die Umlegungen nicht direkt errechnet, weshalb die Umlegungen in einer zusätzlichen Methode errechnet und ausgegeben werden müssen. Da es verschiedenen Arten von Umlegungen gibt, unterscheidet die Methode auch zwischen diesen.

Das Programm unterscheidet dabei zwischen diesen Fällen:

1. Vertauschung[2] == 0 && Vertauschung[3] = 0.

   Dies heißt, dass die Vertauschung nur in der Zahl stattfindet. Dann werden zwischen der Ausgangsziffer und der maximierten Ziffer die Unterschiede gesucht und die Striche so umgelegt, dass sie gleich sind. Dabei wird immer ein Paar an Unterschieden gesucht. Der erste Unterschied nach dem gesucht ist, wo in der Ausgangszahl ein Strich liegt, wo in der maximierten Zahl kein Strich liegt. Von dort wird der Strich dann weggenommen. Er wird zum zweiten Unterschied gelegt, dieser ist dort, wo die Ausgangszahl keinen Strich hat, die maximierte Zahl schon. Diese Umlegung wird dann auf die Ausgangszahl angewendet. Dies wird gemacht, bis zwischen den beiden Zahl keine Unterschiede mehr bestehen.

2. Vertauschung1[2] != 0

Wenn der Wert am Index 2 des Vertauschungsarrays != 0 ist, heißt es, dass zur Ausgangszahl Striche hinzugefügt werden müssen, um die Ziffer zu maximieren. Dann sucht die Methode im Rest der Vertauschungsliste eine Vertauschung, bei welcher der Wert am Index 3 != 0 ist, da das heißt das von der Ausgangszahl noch Striche weggenommen werden müssen, um die maximierte Zahl zu erreichen. Diese Vertauschung heißt Vertauschung2.

Dann findet das Programm auch die Unterschiede in den Ziffern, ähnlich wie zuvor. Dabei wird der Unterschied, bei dem ein Strich weggenommen werden muss in Vertauschung2 gesucht. Der Unterschied bei, der ein Strich hinzugefügt werden muss, wird in Vertauschung1 gesucht. So gleichen sich beide aus und jede Vertauschung verbessert beide Ziffern näher zur Maximierung. Dann wird die Umlegung theoretisch angewendet, indem die Vertauschung in die Konsole ausgegeben wird und die Werte für die beiden Ziffern und die dazugehörigen Vertauschungen geändert, indem die Werte bei Vertauschung1[2] und Vertauschung2[3] beide um 1 verringert werden.

Dies wird so lange gemacht, bis entweder der Wert bei Vertauschung2[3] == 0 oder Vertauschung1[2] == 0 ist. Wenn der Fall eintritt, dass Vertauschung2[3] == 0 && Vertauschung1[2] != 0 ist, dann wird in der Liste der Vertauschungen nach noch einer Vertauschung gesucht, bei welcher Vertauschung[3] != 0 ist. Der Prozess wird dann wiederholt.

Wenn entweder beide =0 sind oder nur Vertauschung1[2] == 0, ist der Umlegungsprozess für Vertauschung1 abgeschlossen.

3. Vertauschung1[3] != 0

Wenn der Wert bei Vertauschung1[3] != 0 ist, geht derselbe Prozess vor, nur dass Vertauschung1 und Vertauschung2 die Rollen tauschen.

Wenn der Wert am Index 3 des Vertauschungsarrays != 0 ist, heißt es, dass bei der Ausgangszahl Striche entfernt, werden müssen, um die maximierte Zahl zu erhalten. Dann sucht die Methode im Rest der Vertauschungsliste eine Vertauschung, bei welcher der Wert am Index 2 != 0 ist, da das heißt das von der Ausgangszahl noch Striche hinzugefügt werden müssen, um die maximierte Zahl zu erreichen. Diese Vertauschung heißt Vertauschung2.

Dann findet das Programm auch die Unterschiede in den Ziffern, ähnlich wie zuvor. Dabei wird der Unterschied, bei dem ein Strich weggenommen werden muss in Vertauschung1 gesucht. Der Unterschied bei, der ein Strich hinzugefügt werden muss, wird in Vertauschung2 gesucht. So gleichen sich beide aus und jede Vertauschung bringt beide Ziffern näher zur Maximierung. Dann wird die Umlegung theoretisch angewendet, indem die Vertauschung in die Konsole ausgegeben wird und die Werte für die beiden Ziffern und die dazugehörigen Vertauschungen geändert, indem die Werte bei Vertauschung1[3] und Vertauschung2[2] beide um 1 verringert werden.

Dies wird so lange gemacht, bis entweder der Wert bei Vertauschung2[2] == 0 oder Vertauschung1[3] == 0 ist. Wenn der Fall eintritt, dass Vertauschung2[2] == 0 &&

Vertauschung1[3] != 0 ist, dann wird in der Liste der Vertauschungen nach noch einer Vertauschung gesucht, bei welcher Vertauschung[2] != 0 ist. Der Prozess wird dann wiederholt.

Wenn entweder beide =0 sind oder nur Vertauschung1[3] == 0, ist der Umlegungsprozess für Vertauschung1 abgeschlossen.

Wenn eine Umlegung in die Konsole ausgegeben wird, sieht dies so aus:

2.1 => 3.4

Dabei ist sind 2.1 und 3.4 beide Beschreibungen von Position in der HexZahl. Schauen wir uns 2.1 an. Die vordere Zahl gibt an, an welchen Index in der HexZahl sich die Ziffer befindet. Die hintere Zahl gibt die Position des Strichs in der Ziffer an(Siehe *Abbildung 2: Modellierung der Plätze*). So werden Koordinaten der Striche angegeben. Die erste Koordinate ist, wo sich der Strich befindet, die zweite Koordinate ist, wo der String hingelegt wird.

2.5 Berechnung der Laufzeit:

Die Laufzeit des Algorithmus in der O-Notation anzugeben ist schwieriger, da es eine rekursive Methode ist, welche von einer Eingabe abhängig ist. Damit ergibt sich eine Art Wahrscheinlichkeitsverteilung, welche angibt, wie oft die Methode wiederholt werden muss, da wir auch noch eine Iterative Funktion einbauen.

Die beste Laufzeit ist O(n), was bedeutet, dass die Liste nur einmal durchgegangen werden muss und die beste und erste Version direkt valide ist.

Die schlechteste Laufzeit O(B-A), B= $\sum_{a=0}^{n} 15 \cdot 16^{a}$, A ist der Wert der ursprünglichen HexZahl. Da alle Möglichkeiten bis zu dem Punkt berechnet werden müssen, wo die ursprüngliche Zahl wieder erreicht wird. Da stellt sich die Frage, wie dieser Wert skaliert.

Wenn wir annehmen, dass die Ziffer einer Hexadezimalzahl zufällig bestimmt ist, hat sie einen durchschnittlichen Wert von 7.5. $(15 - 7.5) \cdot 16^{n}$ skaliert immer noch exponentiell. Das bedeutet, dass der Worst-Case exponentiell skaliert.

Die durchschnittliche Laufzeit kann als
O($p_0 \cdot (n+0) + p_1 \cdot (n+1) + p_2 \cdot (n+2) \ldots + p_{\{B-A]} \cdot (B-A)$) beschrieben werden. Wir können sehen, dass die durchschnittliche Laufzeit von B-A abhängig ist. Das bedeutet, dass (B-A) exponentiell skaliert. Da aber die Wahrscheinlichkeit p auch exponentiell abnimmt, gleicht sich das Wachstum in der Annahme aus. Die Wahrscheinlichkeit nimmt exponentiell ab, da die Wahrscheinlichkeit dafür, das mehre Durchläufe gemacht werden müssen immer kleiner wird, aber nie null annimmt. Unter dem Aspekt kann man annehmen, dass sich der exponentielle Abfall und das exponentielle Wachstum gegenseitig ausgleichen. Das bedeutet das der Rest der Methode abhängig von n skaliert und die durchschnittliche Laufzeit ungefähr als O(n) linear skaliert.

4. Beispiele

Da für die Beispiele 0-2 auch die Belegungen nach jeder Umlegung ausgegeben werden sollten, erkläre ich hier noch kurz die Struktur:

2.1 => 3.4 ist die Umlegung die getätigt wurde. Siehe 2.4.

HexZahl = {1011111 ; 1111001 ; 1101101 ; 0011111 ; } gibt die aktuelle Belegung der HexZahl aus. Die Binärzahlen repräsentieren die einzelnen Ziffern. Eine 0 heißt, dass an der Position gerade kein Strich ist, eine 1 heißt das an der Position gerade ein Strich ist. Die Binärzahl 1011111(6) kann auch als Array aus Booleans gesehen werden. 0=false, 1 = true; ArrayDarstellung = {1,0,1,1,1,1,1}. Index im Array ist die Position aus der Darstellung Abbildung 2.

3.1 hexmax0.txt:

Ausgangszahl:
D24
Anzahl Umlegungen:
3
Maximierte Zahl:
EE4
Umlegungen und Zwischenstand:
HexZahl = {0111101 ; 1101101 ; 0110011 ; }

0.1 => 0.0,

HexZahl = {1011101 ; 1101101 ; 0110011 ; }

0.2 => 0.5,

HexZahl = {1001111 ; 1101101 ; 0110011 ; }

1.1 => 1.5,

HexZahl = {1001111 ; 1001111 ; 0110011 ; }

3.2 hexmax1.txt:

Ausgangszahl:
509C431B55
Anzahl Umlegungen:
8
Maximierte Zahl:
FFFEA97B55
Umlegungen:
HexZahl = {1011011 ; 1111110 ; 1111011 ; 1001110 ; 0110011 ; 1111001 ; 0110000 ; 0011111 ; 1011011 ; 1011011 ; }

0.2 => 3.6,

HexZahl = {1001011 ; 1111110 ; 1111011 ; 1001111 ; 0110011 ; 1111001 ; 0110000 ; 0011111 ; 1011011 ; 1011011 ; }

0.3 => 0.4,

HexZahl = {1000111 ; 1111110 ; 1111011 ; 1001111 ; 0110011 ; 1111001 ; 0110000 ; 0011111 ; 1011011 ; 1011011 ; }

1.1 => 4.0,

HexZahl = {1000111 ; 1011110 ; 1111011 ; 1001111 ; 1110011 ; 1111001 ; 0110000 ; 0011111 ; 1011011 ; 1011011 ; }

1.2 => 4.4,

HexZahl = {1000111 ; 1001110 ; 1111011 ; 1001111 ; 1110111 ; 1111001 ; 0110000 ; 0011111 ; 1011011 ; 1011011 ; }

1.3 => 1.6,

HexZahl = {1000111 ; 1000111 ; 1111011 ; 1001111 ; 1110111 ; 1111001 ; 0110000 ; 0011111 ; 1011011 ; 1011011 ; }

2.1 => 5.5,

HexZahl = {1000111 ; 1000111 ; 1011011 ; 1001111 ; 1110111 ; 1111011 ; 0110000 ; 0011111 ; 1011011 ; 1011011 ; }

2.2 => 6.0,

HexZahl = {1000111 ; 1000111 ; 1001011 ; 1001111 ; 1110111 ; 1111011 ; 1110000 ; 0011111 ; 1011011 ; 1011011 ; }

2.3 => 2.4,

HexZahl = {1000111 ; 1000111 ; 1000111 ; 1001111 ; 1110111 ; 1111011 ; 1110000 ; 0011111 ; 1011011 ; 1011011 ; }

3.3 hexmax2.txt:

Ausgangszahl:

632B29B38F11849015A3BCAEE2CDA0BD496919F8

Anzahl Umlegungen:

37

Maximierte Zahl:

EE4

Umlegungen:

HexZahl = {1011111 ; 1111001 ; 1101101 ; 0011111 ; 1101101 ; 1111011 ; 0011111 ; 1111001 ; 1111111 ; 1000111 ; 0110000 ; 0110000 ; 1111111 ; 0110011 ; 1111011 ; 1111110 ; 0110000 ; 1011011 ; 1110111 ; 1111001 ; 0011111 ; 1001110 ; 1110111 ; 1001111 ; 1001111 ; 1101101 ; 1001110 ; 0111101 ; 1110111 ; 1111110 ; 0011111 ; 0111101 ; 0110011 ; 1111011 ; 1011111 ; 1111011 ; 0110000 ; 1111011 ; 1000111 ; 1111111 ; }
0.2 => 10.0,
HexZahl = {1001111 ; 1111001 ; 1101101 ; 0011111 ; 1101101 ; 1111011 ; 0011111 ; 1111001 ; 1111111 ; 1000111 ; 1110000 ; 0110000 ; 1111111 ; 0110011 ; 1111011 ; 1111110 ; 0110000 ; 1011011 ; 1110111 ; 1111001 ; 0011111 ; 1001110 ; 1110111 ; 1001111 ; 1001111 ; 1101101 ; 1001110 ; 0111101 ; 1110111 ; 1111110 ; 0011111 ; 0111101 ; 0110011 ; 1111011 ; 1011111 ; 1111011 ; 0110000 ; 1111011 ; 1000111 ; 1111111 ; }
0.3 => 10.4,

HexZahl = {1000111 ; 1111001 ; 1101101 ; 0011111 ; 1101101 ; 1111011 ; 0011111 ; 1111001 ; 1111111 ; 1000111 ; 1110100 ; 0110000 ; 1111111 ; 0110011 ; 1111011 ; 1111110 ; 0110000 ; 1011011 ; 1110111 ; 1111001 ; 0011111 ; 1001110 ; 1110111 ; 1001111 ; 1001111 ; 1101101 ; 1001110 ; 0111101 ; 1110111 ; 1111110 ; 0011111 ; 0111101 ; 0110011 ; 1111011 ; 1011111 ; 1111011 ; 0110000 ; 1111011 ; 1000111 ; 1111111 ; }
1.1 => 11.0,
HexZahl = {1000111 ; 1011001 ; 1101101 ; 0011111 ; 1101101 ; 1111011 ; 0011111 ; 1111001 ; 1111111 ; 1000111 ; 1110100 ; 1110000 ; 1111111 ; 0110011 ; 1111011 ; 1111110 ; 0110000 ; 1011011 ; 1110111 ; 1111001 ; 0011111 ; 1001110 ; 1110111 ; 1001111 ; 1001111 ; 1101101 ; 1001110 ; 0111101 ; 1110111 ; 1111110 ; 0011111 ; 0111101 ; 0110011 ; 1111011 ; 1011111 ; 1111011 ; 0110000 ; 1111011 ; 1000111 ; 1111111 ; }
1.2 => 1.4,
HexZahl = {1000111 ; 1001101 ; 1101101 ; 0011111 ; 1101101 ; 1111011 ; 0011111 ; 1111001 ; 1111111 ; 1000111 ; 1110100 ; 1110000 ; 1111111 ; 0110011 ; 1111011 ; 1111110 ; 0110000 ; 1011011 ; 1110111 ; 1111001 ; 0011111 ; 1001110 ; 1110111 ; 1001111 ; 1001111 ; 1101101 ; 1001110 ; 0111101 ; 1110111 ; 1111110 ; 0011111 ; 0111101 ; 0110011 ; 1111011 ; 1011111 ; 1111011 ; 0110000 ; 1111011 ; 1000111 ; 1111111 ; }
1.3 => 1.5,
HexZahl = {1000111 ; 1000111 ; 1101101 ; 0011111 ; 1101101 ; 1111011 ; 0011111 ; 1111001 ; 1111111 ; 1000111 ; 1110100 ; 1110000 ; 1111111 ; 0110011 ; 1111011 ; 1111110 ; 0110000 ; 1011011 ; 1110111 ; 1111001 ; 0011111 ; 1001110 ; 1110111 ; 1001111 ; 1001111 ; 1101101 ; 1001110 ; 0111101 ; 1110111 ; 1111110 ; 0011111 ; 0111101 ; 0110011 ; 1111011 ; 1011111 ; 1111011 ; 0110000 ; 1111011 ; 1000111 ; 1111111 ; }
2.1 => 11.4,
HexZahl = {1000111 ; 1000111 ; 1001101 ; 0011111 ; 1101101 ; 1111011 ; 0011111 ; 1111001 ; 1111111 ; 1000111 ; 1110100 ; 1110100 ; 1111111 ; 0110011 ; 1111011 ; 1111110 ; 0110000 ; 1011011 ; 1110111 ; 1111001 ; 0011111 ; 1001110 ; 1110111 ; 1001111 ; 1001111 ; 1101101 ; 1001110 ; 0111101 ; 1110111 ; 1111110 ; 0011111 ; 0111101 ; 0110011 ; 1111011 ; 1011111 ; 1111011 ; 0110000 ; 1111011 ; 1000111 ; 1111111 ; }
2.3 => 2.5,
HexZahl = {1000111 ; 1000111 ; 1000111 ; 0011111 ; 1101101 ; 1111011 ; 0011111 ; 1111001 ; 1111111 ; 1000111 ; 1110100 ; 1110100 ; 1111111 ; 0110011 ; 1111011 ; 1111110 ; 0110000 ; 1011011 ; 1110111 ; 1111001 ; 0011111 ; 1001110 ; 1110111 ; 1001111 ; 1001111 ; 1101101 ; 1001110 ; 0111101 ; 1110111 ; 1111110 ; 0011111 ; 0111101 ; 0110011 ; 1111011 ; 1011111 ; 1111011 ; 0110000 ; 1111011 ; 1000111 ; 1111111 ; }
3.2 => 16.3,
HexZahl = {1000111 ; 1000111 ; 1000111 ; 0001111 ; 1101101 ; 1111011 ; 0011111 ; 1111001 ; 1111111 ; 1000111 ; 1110100 ; 1110100 ; 1111111 ; 0110011 ; 1111011 ; 1111110 ; 0111000 ; 1011011 ; 1110111 ; 1111001 ; 0011111 ; 1001110 ; 1110111 ; 1001111 ; 1001111 ; 1101101 ; 1001110 ; 0111101 ; 1110111 ; 1111110 ; 0011111 ; 0111101 ; 0110011 ; 1111011 ; 1011111 ; 1111011 ; 0110000 ; 1111011 ; 1000111 ; 1111111 ; }
3.3 => 3 1.0,
HexZahl = {1000111 ; 1000111 ; 1000111 ; 1000111 ; 1101101 ; 1111011 ; 0011111 ; 1111001 ; 1111111 ; 1000111 ; 1110100 ; 1110100 ; 1111111 ; 0110011 ; 1111011 ; 1111110 ; 0111000 ; 1011011 ; 1110111 ; 1111001 ; 0011111 ; 1001110 ; 1110111 ; 1001111 ; 1001111 ; 1101101 ; 1001110 ; 0111101 ; 1110111 ; 1111110 ; 0011111 ; 0111101 ; 0110011 ; 1111011 ; 1011111 ; 1111011 ; 0110000 ; 1111011 ; 1000111 ; 1111111 ; }
4.1 => 16.4,
HexZahl = {1000111 ; 1000111 ; 1000111 ; 1000111 ; 1001101 ; 1111011 ; 0011111 ; 1111001 ; 1111111 ; 1000111 ; 1110100 ; 1110100 ; 1111111 ; 0110011 ; 1111011 ; 1111110 ; 0111100 ; 1011011 ;

1110111 ; 1111001 ; 0011111 ; 1001110 ; 1110111 ; 1001111 ; 1001111 ; 1101101 ; 1001110 ; 0111101 ;
1110111 ; 1111110 ; 0011111 ; 0111101 ; 0110011 ; 1111011 ; 1011111 ; 1111011 ; 0110000 ; 1111011 ;
1000111 ; 1111111 ; }
4.3 => 4.5,
HexZahl = {1000111 ; 1000111 ; 1000111 ; 1000111 ; 1000111 ; 1111011 ; 0011111 ; 1111001 ;
1111111 ; 1000111 ; 1110100 ; 1110100 ; 1111111 ; 0110011 ; 1111011 ; 1111110 ; 0111100 ; 1011011 ;
1110111 ; 1111001 ; 0011111 ; 1001110 ; 1110111 ; 1001111 ; 1001111 ; 1101101 ; 1001110 ; 0111101 ;
1110111 ; 1111110 ; 0011111 ; 0111101 ; 0110011 ; 1111011 ; 1011111 ; 1111011 ; 0110000 ; 1111011 ;
1000111 ; 1111111 ; }
5.1 => 16.6,
HexZahl = {1000111 ; 1000111 ; 1000111 ; 1000111 ; 1000111 ; 1011011 ; 0011111 ; 1111001 ;
1111111 ; 1000111 ; 1110100 ; 1110100 ; 1111111 ; 0110011 ; 1111011 ; 1111110 ; 0111101 ; 1011011 ;
1110111 ; 1111001 ; 0011111 ; 1001110 ; 1110111 ; 1001111 ; 1001111 ; 1101101 ; 1001110 ; 0111101 ;
1110111 ; 1111110 ; 0011111 ; 0111101 ; 0110011 ; 1111011 ; 1011111 ; 1111011 ; 0110000 ; 1111011 ;
1000111 ; 1111111 ; }
5.2 => 17.1,
HexZahl = {1000111 ; 1000111 ; 1000111 ; 1000111 ; 1000111 ; 1001011 ; 0011111 ; 1111001 ;
1111111 ; 1000111 ; 1110100 ; 1110100 ; 1111111 ; 0110011 ; 1111011 ; 1111110 ; 0111101 ; 1111011 ;
1110111 ; 1111001 ; 0011111 ; 1001110 ; 1110111 ; 1001111 ; 1001111 ; 1101101 ; 1001110 ; 0111101 ;
1110111 ; 1111110 ; 0011111 ; 0111101 ; 0110011 ; 1111011 ; 1011111 ; 1111011 ; 0110000 ; 1111011 ;
1000111 ; 1111111 ; }
5.3 => 5.4,
HexZahl = {1000111 ; 1000111 ; 1000111 ; 1000111 ; 1000111 ; 1000111 ; 0011111 ; 1111001 ;
1111111 ; 1000111 ; 1110100 ; 1110100 ; 1111111 ; 0110011 ; 1111011 ; 1111110 ; 0111101 ; 1111011 ;
1110111 ; 1111001 ; 0011111 ; 1001110 ; 1110111 ; 1001111 ; 1001111 ; 1101101 ; 1001110 ; 0111101 ;
1110111 ; 1111110 ; 0011111 ; 0111101 ; 0110011 ; 1111011 ; 1011111 ; 1111011 ; 0110000 ; 1111011 ;
1000111 ; 1111111 ; }
6.2 => 19.5,
HexZahl = {1000111 ; 1000111 ; 1000111 ; 1000111 ; 1000111 ; 1000111 ; 0001111 ; 1111001 ;
1111111 ; 1000111 ; 1110100 ; 1110100 ; 1111111 ; 0110011 ; 1111011 ; 1111110 ; 0111101 ; 1111011 ;
1110111 ; 1111011 ; 0011111 ; 1001110 ; 1110111 ; 1001111 ; 1001111 ; 1101101 ; 1001110 ; 0111101 ;
1110111 ; 1111110 ; 0011111 ; 0111101 ; 0110011 ; 1111011 ; 1011111 ; 1111011 ; 0110000 ; 1111011 ;
1000111 ; 1111111 ; }
6.3 => 6 1.0,
HexZahl = {1000111 ; 1000111 ; 1000111 ; 1000111 ; 1000111 ; 1000111 ; 1000111 ; 1111001 ;
1111111 ; 1000111 ; 1110100 ; 1110100 ; 1111111 ; 0110011 ; 1111011 ; 1111110 ; 0111101 ; 1111011 ;
1110111 ; 1111011 ; 0011111 ; 1001110 ; 1110111 ; 1001111 ; 1001111 ; 1101101 ; 1001110 ; 0111101 ;
1110111 ; 1111110 ; 0011111 ; 0111101 ; 0110011 ; 1111011 ; 1011111 ; 1111011 ; 0110000 ; 1111011 ;
1000111 ; 1111111 ; }
7.1 => 21.6,
HexZahl = {1000111 ; 1000111 ; 1000111 ; 1000111 ; 1000111 ; 1000111 ; 1000111 ; 1011001 ;
1111111 ; 1000111 ; 1110100 ; 1110100 ; 1111111 ; 0110011 ; 1111011 ; 1111110 ; 0111101 ; 1111011 ;
1110111 ; 1111011 ; 0011111 ; 1001111 ; 1110111 ; 1001111 ; 1001111 ; 1101101 ; 1001110 ; 0111101 ;
1110111 ; 1111110 ; 0011111 ; 0111101 ; 0110011 ; 1111011 ; 1011111 ; 1111011 ; 0110000 ; 1111011 ;
1000111 ; 1111111 ; }
7.2 => 7.4,
HexZahl = {1000111 ; 1000111 ; 1000111 ; 1000111 ; 1000111 ; 1000111 ; 1000111 ; 1001101 ;
1111111 ; 1000111 ; 1110100 ; 1110100 ; 1111111 ; 0110011 ; 1111011 ; 1111110 ; 0111101 ; 1111011 ;
1110111 ; 1111011 ; 0011111 ; 1001111 ; 1110111 ; 1001111 ; 1001111 ; 1101101 ; 1001110 ; 0111101 ;

1110111 ; 1111110 ; 0011111 ; 0111101 ; 0110011 ; 1111011 ; 1011111 ; 1111011 ; 0110000 ; 1111011 ; 1000111 ; 1111111 ; }
7.3 => 7.5,
HexZahl = {1000111 ; 1000111 ; 1000111 ; 1000111 ; 1000111 ; 1000111 ; 1000111 ; 1000111 ; 1111111 ; 1000111 ; 1110100 ; 1110100 ; 1111111 ; 0110011 ; 1111011 ; 1111110 ; 0111101 ; 1111011 ; 1110111 ; 1111011 ; 0011111 ; 1001111 ; 1110111 ; 1001111 ; 1001111 ; 1101101 ; 1001110 ; 0111101 ; 1110111 ; 1111110 ; 0011111 ; 0111101 ; 0110011 ; 1111011 ; 1011111 ; 1111011 ; 0110000 ; 1111011 ; 1000111 ; 1111111 ; }
8.1 => 25.2,
HexZahl = {1000111 ; 1000111 ; 1000111 ; 1000111 ; 1000111 ; 1000111 ; 1000111 ; 1000111 ; 1011111 ; 1000111 ; 1110100 ; 1110100 ; 1111111 ; 0110011 ; 1111011 ; 1111110 ; 0111101 ; 1111011 ; 1110111 ; 1111011 ; 0011111 ; 1001111 ; 1110111 ; 1001111 ; 1001111 ; 1111101 ; 1001110 ; 0111101 ; 1110111 ; 1111110 ; 0011111 ; 0111101 ; 0110011 ; 1111011 ; 1011111 ; 1111011 ; 0110000 ; 1111011 ; 1000111 ; 1111111 ; }
8.2 => 25.5,
HexZahl = {1000111 ; 1000111 ; 1000111 ; 1000111 ; 1000111 ; 1000111 ; 1000111 ; 1000111 ; 1001111 ; 1000111 ; 1110100 ; 1110100 ; 1111111 ; 0110011 ; 1111011 ; 1111110 ; 0111101 ; 1111011 ; 1110111 ; 1111011 ; 0011111 ; 1001111 ; 1110111 ; 1001111 ; 1001111 ; 1111111 ; 1001110 ; 0111101 ; 1110111 ; 1111110 ; 0011111 ; 0111101 ; 0110011 ; 1111011 ; 1011111 ; 1111011 ; 0110000 ; 1111011 ; 1000111 ; 1111111 ; }
8.3 => 26.6,
HexZahl = {1000111 ; 1000111 ; 1000111 ; 1000111 ; 1000111 ; 1000111 ; 1000111 ; 1000111 ; 1000111 ; 1000111 ; 1110100 ; 1110100 ; 1111111 ; 0110011 ; 1111011 ; 1111110 ; 0111101 ; 1111011 ; 1110111 ; 1111011 ; 0011111 ; 1001111 ; 1110111 ; 1001111 ; 1001111 ; 1111111 ; 1001111 ; 0111101 ; 1110111 ; 1111110 ; 0011111 ; 0111101 ; 0110011 ; 1111011 ; 1011111 ; 1111011 ; 0110000 ; 1111011 ; 1000111 ; 1111111 ; }
10.1 => 10.5,
HexZahl = {1000111 ; 1000111 ; 1000111 ; 1000111 ; 1000111 ; 1000111 ; 1000111 ; 1000111 ; 1000111 ; 1000111 ; 1010110 ; 1110100 ; 1111111 ; 0110011 ; 1111011 ; 1111110 ; 0111101 ; 1111011 ; 1110111 ; 1111011 ; 0011111 ; 1001111 ; 1110111 ; 1001111 ; 1001111 ; 1111111 ; 1001111 ; 0111101 ; 1110111 ; 1111110 ; 0011111 ; 0111101 ; 0110011 ; 1111011 ; 1011111 ; 1111011 ; 0110000 ; 1111011 ; 1000111 ; 1111111 ; }
10.2 => 10.6,
HexZahl = {1000111 ; 1000111 ; 1000111 ; 1000111 ; 1000111 ; 1000111 ; 1000111 ; 1000111 ; 1000111 ; 1000111 ; 1000111 ; 1110100 ; 1111111 ; 0110011 ; 1111011 ; 1111110 ; 0111101 ; 1111011 ; 1110111 ; 1111011 ; 0011111 ; 1001111 ; 1110111 ; 1001111 ; 1001111 ; 1111111 ; 1001111 ; 0111101 ; 1110111 ; 1111110 ; 0011111 ; 0111101 ; 0110011 ; 1111011 ; 1011111 ; 1111011 ; 0110000 ; 1111011 ; 1000111 ; 1111111 ; }
11.1 => 11.5,
HexZahl = {1000111 ; 1000111 ; 1000111 ; 1000111 ; 1000111 ; 1000111 ; 1000111 ; 1000111 ; 1000111 ; 1000111 ; 1000111 ; 1010110 ; 1111111 ; 0110011 ; 1111011 ; 1111110 ; 0111101 ; 1111011 ; 1110111 ; 1111011 ; 0011111 ; 1001111 ; 1110111 ; 1001111 ; 1001111 ; 1111111 ; 1001111 ; 0111101 ; 1110111 ; 1111110 ; 0011111 ; 0111101 ; 0110011 ; 1111011 ; 1011111 ; 1111011 ; 0110000 ; 1111011 ; 1000111 ; 1111111 ; }
11.2 => 11.6,
HexZahl = {1000111 ; 1000111 ; 1000111 ; 1000111 ; 1000111 ; 1000111 ; 1000111 ; 1000111 ; 1000111 ; 1000111 ; 1000111 ; 1000111 ; 1111111 ; 0110011 ; 1111011 ; 1111110 ; 0111101 ; 1111011 ; 1110111 ; 1111011 ; 0011111 ; 1001111 ; 1110111 ; 1001111 ; 1001111 ; 1111111 ; 1001111 ; 0111101 ; 1110111 ; 1111110 ; 0011111 ; 0111101 ; 0110011 ; 1111011 ; 1011111 ; 1111011 ; 0110000 ; 1111011 ; 1000111 ; 1111111 ; }

12.1 => 29.6,
HexZahl = {1000111 ; 1000111 ; 1000111 ; 1000111 ; 1000111 ; 1000111 ; 1000111 ; 1000111 ; 1000111 ; 1000111 ; 1000111 ; 1011111 ; 0110011 ; 1111011 ; 1111110 ; 0111101 ; 1111011 ; 1110111 ; 1111011 ; 0011111 ; 1001111 ; 1110111 ; 1001111 ; 1001111 ; 1111111 ; 1001111 ; 0111101 ; 1110111 ; 1111111 ; 0011111 ; 0111101 ; 0110011 ; 1111011 ; 1011111 ; 1111011 ; 0110000 ; 1111011 ; 1000111 ; 1111111 ; }
12.2 => 32.0,
HexZahl = {1000111 ; 1000111 ; 1000111 ; 1000111 ; 1000111 ; 1000111 ; 1000111 ; 1000111 ; 1000111 ; 1000111 ; 1000111 ; 1001111 ; 0110011 ; 1111011 ; 1111110 ; 0111101 ; 1111011 ; 1110111 ; 1111011 ; 0011111 ; 1001111 ; 1110111 ; 1001111 ; 1001111 ; 1111111 ; 1001111 ; 0111101 ; 1110111 ; 1111111 ; 0011111 ; 0111101 ; 1110011 ; 1111011 ; 1011111 ; 1111011 ; 0110000 ; 1111011 ; 1000111 ; 1111111 ; }
12.3 => 32.4,
HexZahl = {1000111 ; 1000111 ; 1000111 ; 1000111 ; 1000111 ; 1000111 ; 1000111 ; 1000111 ; 1000111 ; 1000111 ; 1000111 ; 1000111 ; 0110011 ; 1111011 ; 1111110 ; 0111101 ; 1111011 ; 1110111 ; 1111011 ; 0011111 ; 1001111 ; 1110111 ; 1001111 ; 1001111 ; 1111111 ; 1001111 ; 0111101 ; 1110111 ; 1111111 ; 0011111 ; 0111101 ; 1110111 ; 1111011 ; 1011111 ; 1111011 ; 0110000 ; 1111011 ; 1000111 ; 1111111 ; }
13.1 => 13.0,
HexZahl = {1000111 ; 1000111 ; 1000111 ; 1000111 ; 1000111 ; 1000111 ; 1000111 ; 1000111 ; 1000111 ; 1000111 ; 1000111 ; 1000111 ; 1010011 ; 1111011 ; 1111110 ; 0111101 ; 1111011 ; 1110111 ; 1111011 ; 0011111 ; 1001111 ; 1110111 ; 1001111 ; 1001111 ; 1111111 ; 1001111 ; 0111101 ; 1110111 ; 1111111 ; 0011111 ; 0111101 ; 1110111 ; 1111011 ; 1011111 ; 1111011 ; 0110000 ; 1111011 ; 1000111 ; 1111111 ; }
13.2 => 13.4,
HexZahl = {1000111 ; 1000111 ; 1000111 ; 1000111 ; 1000111 ; 1000111 ; 1000111 ; 1000111 ; 1000111 ; 1000111 ; 1000111 ; 1000111 ; 1000111 ; 1111011 ; 1111110 ; 0111101 ; 1111011 ; 1110111 ; 1111011 ; 0011111 ; 1001111 ; 1110111 ; 1001111 ; 1001111 ; 1111111 ; 1001111 ; 0111101 ; 1110111 ; 1111111 ; 0011111 ; 0111101 ; 1110111 ; 1111011 ; 1011111 ; 1111011 ; 0110000 ; 1111011 ; 1000111 ; 1111111 ; }
14.1 => 34.1,
HexZahl = {1000111 ; 1000111 ; 1000111 ; 1000111 ; 1000111 ; 1000111 ; 1000111 ; 1000111 ; 1000111 ; 1000111 ; 1000111 ; 1000111 ; 1000111 ; 1011011 ; 1111110 ; 0111101 ; 1111011 ; 1110111 ; 1111011 ; 0011111 ; 1001111 ; 1110111 ; 1001111 ; 1001111 ; 1111111 ; 1001111 ; 0111101 ; 1110111 ; 1111111 ; 0011111 ; 0111101 ; 1110111 ; 1111011 ; 1111111 ; 1111011 ; 0110000 ; 1111011 ; 1000111 ; 1111111 ; }
14.2 => 36.3,
HexZahl = {1000111 ; 1000111 ; 1000111 ; 1000111 ; 1000111 ; 1000111 ; 1000111 ; 1000111 ; 1000111 ; 1000111 ; 1000111 ; 1000111 ; 1000111 ; 1001011 ; 1111110 ; 0111101 ; 1111011 ; 1110111 ; 1111011 ; 0011111 ; 1001111 ; 1110111 ; 1001111 ; 1001111 ; 1111111 ; 1001111 ; 0111101 ; 1110111 ; 1111111 ; 0011111 ; 0111101 ; 1110111 ; 1111011 ; 1111111 ; 1111011 ; 0111000 ; 1111011 ; 1000111 ; 1111111 ; }
14.3 => 14.4,
HexZahl = {1000111 ; 1000111 ; 1000111 ; 1000111 ; 1000111 ; 1000111 ; 1000111 ; 1000111 ; 1000111 ; 1000111 ; 1000111 ; 1000111 ; 1000111 ; 1000111 ; 1111110 ; 0111101 ; 1111011 ; 1110111 ; 1111011 ; 0011111 ; 1001111 ; 1110111 ; 1001111 ; 1001111 ; 1111111 ; 1001111 ; 0111101 ; 1110111 ; 1111111 ; 0011111 ; 0111101 ; 1110111 ; 1111011 ; 1111111 ; 1111011 ; 0111000 ; 1111011 ; 1000111 ; 1111111 ; }
15.1 => 36.4,

HexZahl = {1000111 ; 1000111 ; 1000111 ; 1000111 ; 1000111 ; 1000111 ; 1000111 ; 1000111 ; 1000111 ; 1000111 ; 1000111 ; 1000111 ; 1000111 ; 1000111 ; 1000111 ; 1011110 ; 0111101 ; 1111011 ; 1110111 ; 1111011 ; 0011111 ; 1001111 ; 1110111 ; 1001111 ; 1001111 ; 1111111 ; 1001111 ; 0111101 ; 1110111 ; 1111111 ; 0011111 ; 0111101 ; 1110111 ; 1111011 ; 1111111 ; 1111011 ; 0111100 ; 1111011 ; 1000111 ; 1111111 ; }
15.2 => 36.6,
HexZahl = {1000111 ; 1000111 ; 1000111 ; 1000111 ; 1000111 ; 1000111 ; 1000111 ; 1000111 ; 1000111 ; 1000111 ; 1000111 ; 1000111 ; 1000111 ; 1000111 ; 1000111 ; 1001110 ; 0111101 ; 1111011 ; 1110111 ; 1111011 ; 0011111 ; 1001111 ; 1110111 ; 1001111 ; 1001111 ; 1111111 ; 1001111 ; 0111101 ; 1110111 ; 1111111 ; 0011111 ; 0111101 ; 1110111 ; 1111011 ; 1111111 ; 1111011 ; 0111101 ; 1111011 ; 1000111 ; 1111111 ; }
15.3 => 15.6,
HexZahl = {1000111 ; 1000111 ; 1000111 ; 1000111 ; 1000111 ; 1000111 ; 1000111 ; 1000111 ; 1000111 ; 1000111 ; 1000111 ; 1000111 ; 1000111 ; 1000111 ; 1000111 ; 1000111 ; 0111101 ; 1111011 ; 1110111 ; 1111011 ; 0011111 ; 1001111 ; 1110111 ; 1001111 ; 1001111 ; 1111111 ; 1001111 ; 0111101 ; 1110111 ; 1111111 ; 0011111 ; 0111101 ; 1110111 ; 1111011 ; 1111111 ; 1111011 ; 0111101 ; 1111011 ; 1000111 ; 1111111 ; }

3.4 hexmax3.txt:

Ausgangszahl:

0E9F1DB46B1E2C081B059EAF198FD491F477CE1CD37EBFB65F8D765055757C6F4796BB8B
3DF7FCAC606DD0627D6B48C17C09

Anzahl Umlegungen:

121

Maximierte Zahl:

FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFAA98BB8
B9DFAFEAE888DD888A88888888888

3.5 hexmax4.txt:

Ausgangszahl:

1A02B6B50D7489D7708A678593036FA265F2925B21C28B4724DD822038E3B4804192322F23
0AB7AF7BDA0A61BA7D4AD8F888

Anzahl Umlegungen:

87

Maximierte Zahl

FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEB8DE88BAA8ADD888898E9BA88AD9898
8F898AB7AF7BDA8A61BA7D4AD8F888:

3.6 hexmax5.txt:

Ausgangszahl:

EF50AA77ECAD25F5E11A307B713EAAEC55215E7E640FD263FA529BBB48DC8FAFE14D5B
02EBF792B5CCBBE9FA1330B867E330A6412870DD2BA6ED0DBCAE553115C9A31FF350C5
DF993824886DB5111A83E773F23AD7FA81A845C11E22C4C45005D192ADE68AA9AA57406
EB0E7C9CA13AD03888F6ABEDF1475FE9832C66BFDC28964B7022BDD969E5533EA4F2E4
EABA75B5DC11972824896786BD1E4A7A7748FDF1452A5079E0F9E6005F040594185EA03B5

A869B109A283797AB31394941BFE4D38392AD12186FF6D233585D8C820F197FBA9F6F063A
0877A912CCBDCB14BEECBAEC0ED061CFF60BD517B6879B72B9EFE977A9D3259632C718
FBF45156A16576AA7F9A4FAD40AD8BC87EC569F9C1364A63B1623A5AD559AAF62520527
82BF9A46104E443A3932D25AAE8F8C59F10875FAD3CBD885CE68665F2C826B1E1735EE2F
DF0A1965149DF353EE0BE81F3EC133922EF43EBC09EF755FBD740C8E4D024B033F0E8F34
49C94102902E143433262CDA1925A2B7FD01BEF26CD51A1FC22EDD49623EE9DEB14C138
A7A6C47B677F033BDEB849738C3AE5935A2F54B99237912F2958FDFB82217C175448AA823
0FDCB3B3869824A826635B538D47D847D8479A88F350E24B31787DFD60DE5E260B265829E
036BE340FFC0D8C05555E75092226E7D54DEB42E1BB2CA9661A882FB718E7AA53F1E606

Anzahl Umlegungen:

1369

Maximierte Zahl:

FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFF88EFA9EBE89EFA99FBDAA8E8EAD88AB899F8E8F9AA9E9A
D88988EDA9A99888EDAD98988888888888888888888888888888888888888888888888888888888
88888888888888888888888888888888888888888888888888888888888888888888888888888888888888888
88888888888888888888888888888888888888888888888888888888888888888888888888888888888888888
8888888888888888888888888888888888888888888888888888

4. Quellcode

4.1 Methode hexZahlMaximieren:

```java
/**
     * Diese Methode maximiert die gegebene Hexadezimalzahl indem sie die angegebenen Umlegungen benutzt. Dafür benutzt sie die Methode @rekursivHexZahlMaximieren
     * Kümmert sich um die richtigen Rahmenbedingungen und sorgt dafür, dass die durch @rekursivHexZahlMaximieren errechneten Vertauschungen angewendet werden
     * @param hexZahl die Ausgangszahl als Int[] mit den Werten der einzelnen Ziffern
     * @param pUebrigeUmlegungen die Anzahl der Umlegungen die noch verfügbar sind.
     * @return Gibt die maximierte HexZahl als Int[] zurück.
     */
    public int[] hexZahlMaximieren(int[] hexZahl, int pUebrigeUmlegungen) {
        freiePlaetze = this.freieStellenFeststellen(hexZahl);
        besetztePlaetze = this.besetztePlaetzeFeststellen(hexZahl);
        uebrigeUmlegungen = pUebrigeUmlegungen;
        zwischenspeicherUmlegungen = 0;
```

```
        boolean suc = this.rekursivHexZahlMaximieren(hexZahl,Vertauschungsmatrix[hexZahl[0]],uebrigeUmlegungen, freiePlaetze, besetztePlaetze, zwischenspeicherUmlegungen,0);
        if(suc == true) {
        for(int i = 0; i<hexZahl.length; i++) {
            int[] vertauschung = vertauschOptionArrayListKonvertieren(speicherVertauschungen.get(i));
            if(vertauschung != null) {
                //Erhaltene Vertauschung auf die Werte anwenden
                hexZahl[i] = vertauschung[0];
                freiePlaetze = freiePlaetze-(7- anzahlStrichProZiffer[hexZahl[i]]);
                besetztePlaetze = besetztePlaetze-(anzahlStrichProZiffer[hexZahl[i]] -2);
                uebrigeUmlegungen = uebrigeUmlegungen-vertauschung[1];
                int NeuZwischenspeicher = zwischenspeicherUmlegungen + vertauschung[2];
                if(zwischenspeicherUmlegungen<0 && NeuZwischenspeicher >0) {
                    uebrigeUmlegungen = uebrigeUmlegungen - (vertauschung[1]-vertauschung[2]);
                }
                NeuZwischenspeicher = zwischenspeicherUmlegungen - vertauschung[3];
                if(zwischenspeicherUmlegungen>0 && NeuZwischenspeicher <0) {
                    uebrigeUmlegungen = uebrigeUmlegungen - (vertauschung[1]-vertauschung[3]);
                }
            }
            if(zwischenspeicherUmlegungen == freiePlaetze) {
                for(int g= i+1; g<hexZahl.length; g++) {
                    hexZahl[g] = 8;
                }
                return hexZahl;
                //Methode um den Rest der Zahl zu f□llen
            }else if((zwischenspeicherUmlegungen)*-1 == besetztePlaetze) {
                //Methode um den Rest der Zahl zu lehren/minimieren
                for(int g= i+1; g<hexZahl.length; g++) {
                    hexZahl[g] = 2;
                }
                return hexZahl;
            }
        }
        return hexZahl;
        }else {
            System.out.println("failed");
        }
        return null;
    }
```

4.2 rekursivHexZahlMaximieren:

```
/**
     * Ist der rekursive Teil, welcher die HexZahl maximiert.
     * @param hexZahl
     * @param tauschMatrix
     * @param verbleibendeUmlegungen
```

```
     * @param freiePlaetze
     * @param uebrigePlaetze
     * @param zwischenspeicherVertauschungen
     * @param index
     * @return
     */
    public boolean rekursivHexZahlMaximieren(int[] hexZahl, int[][] tauschMatrix,
int verbleibendeUmlegungen, int freiePlaetze, int uebrigePlaetze, int zwischenspei-
cherVertauschungen, int index){
        System.out.print(freiePlaetze);
        if(zwischenspeicherVertauschungen == freiePlaetze) {
            ArrayList<Integer> vertauschOption = new ArrayList<Integer>();
            vertauschOption.add(8);
            vertauschOption.add(0);
            vertauschOption.add(0);
            vertauschOption.add(0);
            for(int i = index; i<hexZahl.length; i++) {
                speicherVertauschungen.add(vertauschOption);
            }
            return true;
            //restlichen Zahlen zu 8 machen
        } else if(zwischenspeicherVertauschungen*-1 == uebrigePlaetze) {
            ArrayList<Integer> vertauschOption = new ArrayList<Integer>();
            vertauschOption.add(1);
            vertauschOption.add(0);
            vertauschOption.add(0);
            vertauschOption.add(0);
            for(int i = index; i<hexZahl.length; i++) {
                speicherVertauschungen.add(vertauschOption);
            }
            return true;
            //restlichen Zahlen zu 1 machen
        }
        boolean tauschValide = false;
        int counter = 1;
        while(tauschValide == false && tauschMatrix.length > counter) {
            int[] vertauschOption = tauschMatrix[tauschMatrix.length-counter];
            if(vertauschOption[2] == 0 && vertauschOption[3] == 0 && vertauschOp-
tion[1]<=verbleibendeUmlegungen) {
                int pFreiePlaetze = freiePlaetze-(7- anzahlStrichProZif-
fer[hexZahl[index]]);
                int pBesetztePlaetze = uebrigePlaetze-(anzahlStrichProZif-
fer[hexZahl[index]] -2);
                int pVerbleibendeUmlegungen = verbleibendeUmlegungen-vertauschOp-
tion[1];
                System.out.println(zwischenspeicherVertauschungen + "Verbleibende
Umlegungen: " + pVerbleibendeUmlegungen + ", Zahl: " + vertauschOption[0]);
```

```
                boolean success = false;
                if(hexZahl.length-1 == index) {
                    if(zwischenspeicherVertauschungen == 0 ) {
                        success = true;
                    }
                }else {
                    success = this.rekursivHexZahlMaximieren(hexZahl, Vertau-
schungsmatrix[hexZahl[index+1]], pVerbleibendeUmlegungen, pFreiePlaetze, pBesetzte-
Plaetze, zwischenspeicherVertauschungen, index+1);
                }
                if(success == true) {
                    System.out.println("i"+index);
                    System.out.println(zwischenspeicherVertauschungen + "Verblei-
bende Umlegungen: " + pVerbleibendeUmlegungen + ", Zahl: " + vertauschOption[0]);
                    speicherVertauschungen.add(0, vertauschOptionArrayKonvertie-
ren(vertauschOption));
                    return true;
                }
            }else if(vertauschOption[2]!= 0) {
                if(zwischenspeicherVertauschungen>0) {
                    int pBenutzteVertauschungen = vertauschOption[1]-vertauschOp-
tion[2];
                    if((zwischenspeicherVertauschungen - vertauschOption[2])<0) {
                        pBenutzteVertauschungen = pBenutzteVertauschungen - (zwi-
schenspeicherVertauschungen - vertauschOption[2]);
                    }
                    if((zwischenspeicherVertauschungen-vertauschOption[2] )<=
freiePlaetze && (zwischenspeicherVertauschungen-vertauschOption[2])*-1<= uebrige-
Plaetze && pBenutzteVertauschungen <= verbleibendeUmlegungen) {
                        int pFreiePlaetze = freiePlaetze-(7- anzahlStrichePro Zif-
fer[hexZahl[index]]);
                        int pBesetztePlaetze = uebrigePlaetze-(anzahlStrichePro Zif-
fer[hexZahl[index]] -2);
                        int pVerbleibendeUmlegungen = verbleibendeUmlegungen-(ver-
tauschOption[1]-vertauschOption[2]);
                        int pZwischenspeicher = zwischenspeicherVertauschungen -
vertauschOption[2];
                        if(pZwischenspeicher <0) {
                            pVerbleibendeUmlegungen = pVerbleibendeUmlegungen -
pZwischenspeicher;//neuer Zwischenspeicher wird subtrahiert, da er hier negativ ist
und somit die Anzahl der zusätzlich benutzen Umlegungen angibt
                        }
                        System.out.println(zwischenspeicherVertauschungen + "Ver-
bleibende Umlegungen: " + pVerbleibendeUmlegungen + ", Zahl: " + vertauschOp-
tion[0]);
                        boolean success = false;
                        if(hexZahl.length-1 == index) {
                            if(zwischenspeicherVertauschungen == 0 ) {
```

```
                                success = true;
                            }
                        }else {
                            success = this.rekursivHexZahlMaximieren(hexZahl, Ver-
tauschungsmatrix[hexZahl[index+1]], pVerbleibendeUmlegungen, pFreiePlaetze, pBe-
setztePlaetze, pZwischenspeicher, index+1);
                        }
                        if(success == true) {
                            speicherVertauschungen.add(0, vertauschOptionArrayKon-
vertieren(vertauschOption));
                            return true;
                        }
                    }
                }else if(zwischenspeicherVertauschungen<=0) {
                    if((zwischenspeicherVertauschungen-vertauschOption[2])<= freie-
Plaetze && (zwischenspeicherVertauschungen-vertauschOption[2])*-1<= uebrigePlaetze
&& vertauschOption[1] <= verbleibendeUmlegungen) {
                        int pFreiePlaetze = freiePlaetze-(7- anzahlStrichProZif-
fer[hexZahl[index]]);
                        int pBesetztePlaetze = uebrigePlaetze-(anzahlStrichProZif-
fer[hexZahl[index]] -2);
                        int pVerbleibendeUmlegungen = verbleibendeUmlegungen-ver-
tauschOption[1];
                        int pZwischenspeicher = zwischenspeicherVertauschungen -
vertauschOption[2];
                        System.out.println(zwischenspeicherVertauschungen + "Ver-
bleibende Umlegungen: " + pVerbleibendeUmlegungen + ", Zahl: " + vertauschOp-
tion[0]);

                        boolean success = false;
                        if(hexZahl.length-1 == index) {
                            if(zwischenspeicherVertauschungen == 0 ) {
                                success = true;
                            }
                        }else {
                            success = this.rekursivHexZahlMaximieren(hexZahl, Ver-
tauschungsmatrix[hexZahl[index+1]], pVerbleibendeUmlegungen, pFreiePlaetze, pBe-
setztePlaetze, pZwischenspeicher, index+1);
                        }
                        if(success == true) {
                            speicherVertauschungen.add(0, vertauschOptionArrayKon-
vertieren(vertauschOption));
                            return true;
                        }
                    }
                }
            }else if(vertauschOption[3] != 0){
                if(zwischenspeicherVertauschungen>=0) {
```

```
                    if((zwischenspeicherVertauschungen+vertauschOption[3] )<=
freiePlaetze && vertauschOption[1]<=verbleibendeUmlegungen) {
                        int pFreiePlaetze = freiePlaetze-(7- anzahlStrichProZif-
fer[hexZahl[index]]);
                        int pBesetztePlaetze = uebrigePlaetze-(anzahlStrichProZif-
fer[hexZahl[index]] -2);
                        int pVerbleibendeUmlegungen = verbleibendeUmlegungen-ver-
tauschOption[1];
                        int pZwischenspeicher = zwischenspeicherVertauschungen +
vertauschOption[3];
                        System.out.println(zwischenspeicherVertauschungen + "Ver-
bleibende Umlegungen: " + pVerbleibendeUmlegungen + ", Zahl: " + vertauschOp-
tion[0]);

                        boolean success = false;
                        if(hexZahl.length-1 == index) {
                            if(zwischenspeicherVertauschungen == 0 ) {
                                success = true;
                            }
                        }else {
                            success = this.rekursivHexZahlMaximieren(hexZahl, Ver-
tauschungsmatrix[hexZahl[index+1]], pVerbleibendeUmlegungen, pFreiePlaetze, pBe-
setztePlaetze, pZwischenspeicher, index+1);
                        }
                        if(success == true) {
                            speicherVertauschungen.add(0, vertauschOptionArrayKon-
vertieren(vertauschOption));
                            return true;
                        }
                    }
                }else if(zwischenspeicherVertauschungen<0) {
                    int pBenutzteVertauschungen = vertauschOption[1]-vertauschOp-
tion[3];
                    if((zwischenspeicherVertauschungen + vertauschOption[3])>0) {
                        pBenutzteVertauschungen = pBenutzteVertauschungen + (zwi-
schenspeicherVertauschungen + vertauschOption[3]);
                    }
                    if(((zwischenspeicherVertauschungen+vertauschOption[3]))<=
freiePlaetze && pBenutzteVertauschungen <= verbleibendeUmlegungen){
                        int pFreiePlaetze = freiePlaetze-(7- anzahlStrichProZif-
fer[hexZahl[index]]);
                        int pBesetztePlaetze = uebrigePlaetze-(anzahlStrichProZif-
fer[hexZahl[index]] -2);
                        int pVerbleibendeUmlegungen = verbleibendeUmlegungen-(ver-
tauschOption[1]-vertauschOption[3]);
                        int pZwischenspeicher = zwischenspeicherVertauschungen +
vertauschOption[3];
                        if(pZwischenspeicher >0) {
```

```
                            pVerbleibendeUmlegungen = pVerbleibendeUmlegungen +
pZwischenspeicher;
                        }
                        System.out.println(zwischenspeicherVertauschungen + "Ver-
bleibende Umlegungen: " + pVerbleibendeUmlegungen + ", Zahl: " + vertauschOp-
tion[0]);

                        boolean success = false;
                        if(hexZahl.length-1 == index) {
                            if(zwischenspeicherVertauschungen == 0 ) {
                                success = true;
                            }
                        }else {
                            success = this.rekursivHexZahlMaximieren(hexZahl, Ver-
tauschungsmatrix[hexZahl[index+1]], pVerbleibendeUmlegungen, pFreiePlaetze, pBe-
setztePlaetze, pZwischenspeicher, index+1);
                        }
                        if(success == true) {
                            speicherVertauschungen.add(0, vertauschOptionArrayKon-
vertieren(vertauschOption));
                            return true;
                        }
                    }
                }
            }
            counter++;
        }
        System.out.println(zwischenspeicherVertauschungen + "Verbleibende Umlegun-
gen: " + verbleibendeUmlegungen + ", Zahl: " + hexZahl[index]);

        boolean success = false;
        if(hexZahl.length-1 == index) {
            if(zwischenspeicherVertauschungen == 0 ) {
                success = true;
            }
        }else {
            success = this.rekursivHexZahlMaximieren(hexZahl, Vertauschungs-
matrix[hexZahl[index+1]], verbleibendeUmlegungen, freiePlaetze, uebrigePlaetze,
zwischenspeicherVertauschungen, index+1);
        }
        if(success == true){
            ArrayList<Integer> temp = new ArrayList<Integer>();
            temp.add(hexZahl[index]);
            temp.add(0);
            temp.add(0);
            temp.add(0);
            speicherVertauschungen.add(0,temp);
            return true;
```

```
        }
        return false;
    }
```

4.3 printOutUmlegungen:

```
/**
     * Gibt die Umlegungen an, welche mithilfe der gespeicherten Vertauschungen,
die @param pHexZahl maximieren.
     * Außerdem werden auch die Zwischenstände der HexZahl angegeben zwischen den
einzelnen Umlegungen.
     * Geht dabei iterativ vor.
     * @param pSpeicherVertauschungen
     * @param pHexZahl
     */
    public void printOutUmlegungen(ArrayList<ArrayList<Integer>> pSpeicherVertau-
schungen, int[] pHexZahl){
        ArrayList<char[]> charHexZahl = new ArrayList<char[]>();
        for(int i = 0; i<pHexZahl.length; i++){
            charHexZahl.add(ziffern.get(pHexZahl[i]).toCharArray());
        }
        CharHexZahlAusgeben(charHexZahl);
        for(int i = 0; i< pSpeicherVertauschungen.size(); i++){//geht die Liste der
Vertauschungen durch
            ArrayList<Integer> tempVertauschung = pSpeicherVertauschungen.get(i);
// speichert die aktuelle Vertauschung als ArrayList ab
            if(tempVertauschung.get(2) != 0 || tempVertauschung.get(3)!=
0){//braucht die Vertauschung
                if(tempVertauschung.get(2)!=0){
                    char[] tempZahl1 = charHexZahl.get(i);
                    char[] tempZahl2 = ziffern.get(tempVertauschung.get(0)).toCha-
rArray();
                    while(!charArrEqual(tempZahl1, tempZahl2)){
                        if(tempVertauschung.get(2)>0){
                            for(int h=i+1; h<pSpeicherVertauschungen.size(); h++){
                                if(pSpeicherVertauschungen.get(h).get(3)>0 &&
tempVertauschung.get(2)>0){
                                    char[] tempZahl3 = charHexZahl.get(h);
                                    char[] tempZahl4 = ziffern.get(pSpeicherVertau-
schungen.get(h).get(0)).toCharArray();
                                    while(tempVertauschung.get(2)>0 && pSpeicher-
Vertauschungen.get(h).get(3)>0){
                                        for(int j = 0; j<tempZahl1.length; j++){
                                            if(tempZahl1[j] != tempZahl2[j]){
                                                if(tempZahl1[j] == '0'){// An
Stelle j soll eine 1 in der Finalen Zahl sein, ist gerade 0
                                                    for(int k = 0;
k<tempZahl4.length; k++){
```

```
                                                    if(tempZahl3[k] !=
tempZahl4[k] && tempZahl3[k] == '1'){
                                                        System.out.println(h +
"." + k + "=> " + i +"." + j + ", ");
                                                        tempZahl3[k] = '0';
                                                        tempZahl1[j] = '1';
                                                        charHexZahl.set(i,
tempZahl1);
                                                        charHexZahl.set(h,
tempZahl3);
                                                        CharHexZahlAusge-
ben(charHexZahl);
                                                        tempVertauschung.set(2,
tempVertauschung.get(2) -1);
                                                        tempVertau-
schung.set(21, tempVertauschung.get(1) -1);
                                                        pSpeicherVertauschun-
gen.get(h).set(3, tempVertauschung.get(3) -1);
                                                        pSpeicherVertauschun-
gen.get(h).set(1, tempVertauschung.get(1) -1);
                                                        k=tempZahl2.length;
                                                    }
                                                }
                                            }
                                        }
                                    }
                                } else if(tempVertauschung.get(2) == 0){
                                    h = pSpeicherVertauschungen.size();
                                }
                            }
                        }else{
                            while(!charArrEqual(tempZahl1, tempZahl2)){
                                for(int j = 0; j<tempZahl1.length; j++){
                                    if(tempZahl1[j] != tempZahl2[j]){
                                        if(tempZahl1[j] == '1'){
                                            for(int k = j; k<tempZahl2.length;
k++){
                                                if(tempZahl1[k] != tempZahl2[k] &&
tempZahl1[k] == '0'){
                                                    System.out.println(i+ "." + j +
" => " + i + "." + k + ", ");
                                                    tempZahl1[k] = '1';
                                                    tempZahl1[j] = '0';
                                                    charHexZahl.set(i, tempZahl1);
                                                    CharHexZahlAusgeben(char-
HexZahl);
                                                    k=tempZahl2.length;
```

```
                                        }
                                    }
                                } else if(tempZahl1[j] == '0'){
                                    for(int k = j; k<tempZahl2.length;
k++){
                                        if(tempZahl1[k] != tempZahl2[k] &&
tempZahl1[k] == '1'){
                                            System.out.println(i + "." + k
+ " => " + i + "." + j + ", ");
                                            tempZahl1[k] = '0';
                                            tempZahl1[j] = '1';
                                            charHexZahl.set(i, tempZahl1);
                                            CharHexZahlAusgeben(char-
HexZahl);
                                            k=tempZahl2.length;
                                        }
                                    }
                                }
                            }
                        }
                    }
                } else{
                    char[] tempZahl1 = charHexZahl.get(i);
                    char[] tempZahl2 = ziffern.get(tempVertauschung.get(0)).toCha-
rArray();
                    while(!charArrEqual(tempZahl1, tempZahl2)){
                        if(tempVertauschung.get(3)>0){
                            for(int h=i+1; h<pSpeicherVertauschungen.size(); h++){
                                if(pSpeicherVertauschungen.get(h).get(2)>0 &&
tempVertauschung.get(3)>0){
                                    char[] tempZahl3 = charHexZahl.get(h);
                                    char[] tempZahl4 = ziffern.get(pSpeicherVertau-
schungen.get(h).get(0)).toCharArray();
                                    while(tempVertauschung.get(3)>0 && pSpeicher-
Vertauschungen.get(h).get(2)>0){
                                        for(int j = 0; j<tempZahl1.length; j++){
                                            if(tempZahl1[j] != tempZahl2[j]){
                                                if(tempZahl1[j] == '1'){// An
Stelle j soll eine 0 in der Finalen Zahl sein, ist gerade 1
                                                    for(int k = 0;
k<tempZahl4.length; k++){
                                                        if(tempZahl3[k] !=
tempZahl4[k] && tempZahl3[k] == '0'){
                                                            Sys-
tem.out.println(i+"." + j + " => " + h + "." + k + ", ");
                                                            tempZahl3[k] = '1';
```

```
                                                                tempZahl1[j] = '0';
                                                                k=tempZahl2.length;
                                                                charHexZahl.set(i,
tempZahl1);
                                                                charHexZahl.set(h,
tempZahl3);
                                                                CharHexZahlAusge-
ben(charHexZahl);
                                                                int tempVertauschungs-
Zahl = tempVertauschung.get(3)-1;
                                                                tempVertauschung.set(3,
tempVertauschungsZahl);
                                                                tempVertauschungsZahl =
tempVertauschung.get(1) -1;
                                                                tempVertauschung.set(1,
tempVertauschungsZahl);
                                                                tempVertauschungsZahl =
1;
                                                                pSpeicherVertauschun-
gen.get(h).set(2, pSpeicherVertauschungen.get(h).get(2) -1);
                                                                pSpeicherVertauschun-
gen.get(h).set(1, pSpeicherVertauschungen.get(h).get(1) -1);
                                                                if(tempVertausch-
ung.get(3) == 0){
                                                                    j=tempZahl1.length;
                                                                }
                                                            }
                                                        }
                                                    }
                                                }
                                            }
                                        } else if(tempVertauschung.get(3) == 0){
                                            h = pSpeicherVertauschungen.size();
                                        }
                                    }
                                }else{
                                    while(!charArrEqual(tempZahl1, tempZahl2)){
                                        for(int j = 0; j<tempZahl1.length; j++){
                                            if(tempZahl1[j] != tempZahl2[j]){
                                                if(tempZahl1[j] == '1'){
                                                    for(int k = j; k<tempZahl2.length;
k++){
                                                        if(tempZahl1[k] != tempZahl2[k] &&
tempZahl1[k] == '0'){
                                                            System.out.println(i+"." + j +
" => " + i +"." + k +", ");
                                                            tempZahl1[k] = '1';
```

```
                                    tempZahl1[j] = '0';
                                    charHexZahl.set(i, tempZahl1);
                                    CharHexZahlAusgeben(char-
HexZahl);
                                    k=tempZahl2.length;
                                }
                            }
                        } else if(tempZahl1[j] == '0'){
                            for(int k = j; k<tempZahl2.length;
k++){
                                if(tempZahl1[k] != tempZahl2[k] &&
tempZahl1[k] == '1'){
                                    System.out.println(i+"." + k +
" => " + i+ " 1." + j + ", ");
                                    tempZahl1[k] = '0';
                                    tempZahl1[j] = '1';
                                    charHexZahl.set(i, tempZahl1);
                                    CharHexZahlAusgeben(char-
HexZahl);
                                    k=tempZahl2.length;
                                }
                            }
                        }
                    }
                }
            }
        } else{ //funktioniert
            char[] tempZahl1 = charHexZahl.get(i);
            char[] tempZahl2 = ziffern.get(tempVertauschung.get(0)).toCha-
rArray();
            while(!charArrEqual(tempZahl1, tempZahl2)){
                    for(int j = 0; j<tempZahl1.length; j++){
                        if(tempZahl1[j] != tempZahl2[j]){
                            if(tempZahl1[j] == '1'){
                                for(int k = j; k<tempZahl2.length; k++){
                                    if(tempZahl1[k] != tempZahl2[k] &&
tempZahl1[k] == '0'){
                                        System.out.println( i+ "." + j + " => "
+ i+  "." + k + ", ");
                                        tempZahl1[k] = '1';
                                        tempZahl1[j] = '0';
                                        charHexZahl.set(i, tempZahl1);
                                        CharHexZahlAusgeben(charHexZahl);
                                        k=tempZahl2.length;
                                    }
```

```
                                    }
                                } else if(tempZahl1[j] == '0'){
                                    for(int k = j; k<tempZahl2.length; k++){
                                        if(tempZahl1[k] != tempZahl2[k] &&
tempZahl1[k] == '1'){
                                            System.out.println(i+ "." + k + " => "
+ i + "." + j + ", ");
                                            tempZahl1[k] = '0';
                                            tempZahl1[j] = '1';
                                            charHexZahl.set(i, tempZahl1);
                                            CharHexZahlAusgeben(charHexZahl);
                                            k=tempZahl2.length;
                                        }
                                    }
                                }
                            }
                        }
                    }
                }
            }
        }
```

4.4 Die Vertauschungsmatrix:

```
int[][][] Vertauschungsmatrix = {
            {{0,0,0,0,},{1,4,0,4,},{2,2,0,1,},{3,2,0,1,},{4,3,0,2,},{5,2,0,1,},{6,1
,0,0,},{7,3,0,3,},{8,1,1,0,},{9,1,0,0,},{10,1,0,0,},{11,2,0,1,},{12,2,0,2,},{13,2,0
,1,},{14,2,0,1,},{15,3,0,2,}},
            {{0,4,4,0,},{1,0,0,0,},{2,4,3,0,},{3,3,3,0,},{4,2,2,0,},{5,4,3,0,},{6,5
,4,0,},{7,1,1,0,},{8,5,5,0,},{9,4,4,0,},{10,4,4,0,},{11,4,3,0,},{12,4,2,0,},{13,3,3
,0,},{14,5,3,0,},{15,4,2,0,}},
            {{0,2,1,0,},{1,4,0,3,},{2,0,0,0,},{3,1,0,0,},{4,3,0,1,},{5,2,0,0,},{6,2
,1,0,},{7,3,0,2,},{8,2,2,0,},{9,2,1,0,},{10,2,1,0,},{11,2,0,0,},{12,2,0,1,},{13,1,0
,0,},{14,1,0,0,},{15,2,0,1,}},
            {{0,2,1,0,},{1,3,0,3,},{2,1,0,0,},{3,0,0,0,},{4,2,0,1,},{5,1,0,0,},{6,2
,1,0,},{7,2,0,2,},{8,2,2,0,},{9,1,1,0,},{10,2,1,0,},{11,2,0,0,},{12,3,0,1,},{13,1,0
,0,},{14,2,0,0,},{15,3,0,1,}},
            {{0,3,2,0,},{1,2,0,2,},{2,3,1,0,},{3,2,1,0,},{4,0,0,0,},{5,2,1,0,},{6,3
,2,0,},{7,2,0,1,},{8,3,3,0,},{9,2,2,0,},{10,2,2,0,},{11,2,1,0,},{12,3,0,0,},{13,2,1
,0,},{14,3,1,0,},{15,2,0,0,}},
            {{0,2,1,0,},{1,4,0,3,},{2,2,0,0,},{3,1,0,0,},{4,2,0,1,},{5,0,0,0,},{6,1
,1,0,},{7,3,0,2,},{8,2,2,0,},{9,1,1,0,},{10,2,1,0,},{11,1,0,0,},{12,2,0,1,},{13,2,0
,0,},{14,1,0,0,},{15,2,0,1,}},
            {{0,1,0,0,},{1,5,0,4,},{2,2,0,1,},{3,2,0,1,},{4,3,0,2,},{5,1,0,1,},{6,0
,0,0,},{7,4,0,3,},{8,1,1,0,},{9,1,0,0,},{10,1,0,0,},{11,1,0,1,},{12,2,0,2,},{13,2,0
,1,},{14,1,0,1,},{15,2,0,2,}},
            {{0,3,3,0,},{1,1,0,1,},{2,3,2,0,},{3,2,2,0,},{4,2,1,0,},{5,3,2,0,},{6,4
,3,0,},{7,0,0,0,},{8,4,4,0,},{9,3,3,0,},{10,3,3,0,},{11,4,2,0,},{12,3,1,0,},{13,3,2
,0,},{14,4,2,0,},{15,3,1,0,}},
```

```
            {{0,1,0,1,},{1,5,0,5,},{2,2,0,2,},{3,2,0,2,},{4,3,0,3,},{5,2,0,2,},{6,1
,0,1,},{7,4,0,4,},{8,0,0,0,},{9,1,0,1,},{10,1,0,1,},{11,2,0,2,},{12,3,0,3,},{13,2,0
,2,},{14,2,0,2,},{15,3,0,3,}},
            {{0,1,0,0,},{1,4,0,4,},{2,2,0,1,},{3,1,0,1,},{4,2,0,2,},{5,1,0,1,},{6,1
,0,0,},{7,3,0,3,},{8,1,1,0,},{9,0,0,0,},{10,1,0,0,},{11,2,0,1,},{12,3,0,2,},{13,2,0
,1,},{14,2,0,1,},{15,3,0,2,}},
            {{0,1,0,0,},{1,4,0,4,},{2,2,0,1,},{3,2,0,1,},{4,2,0,2,},{5,2,0,1,},{6,1
,0,0,},{7,3,0,3,},{8,1,1,0,},{9,1,0,0,},{10,0,0,0,},{11,2,0,1,},{12,3,0,2,},{13,2,0
,1,},{14,2,0,1,},{15,2,0,2,}},
            {{0,2,1,0,},{1,4,0,3,},{2,2,0,0,},{3,2,0,0,},{4,2,0,1,},{5,1,0,0,},{6,1
,1,0,},{7,4,0,2,},{8,2,2,0,},{9,2,1,0,},{10,2,1,0,},{11,0,0,0,},{12,2,0,1,},{13,1,0
,0,},{14,1,0,0,},{15,2,0,1,}},
            {{0,2,2,0,},{1,4,0,2,},{2,2,1,0,},{3,3,1,0,},{4,3,0,0,},{5,2,1,0,},{6,2
,2,0,},{7,3,0,1,},{8,3,3,0,},{9,3,2,0,},{10,3,2,0,},{11,2,1,0,},{12,0,0,0,},{13,3,1
,0,},{14,1,1,0,},{15,1,0,0,}},
            {{0,2,1,0,},{1,3,0,3,},{2,1,0,0,},{3,1,0,0,},{4,2,0,1,},{5,2,0,0,},{6,2
,1,0,},{7,3,0,2,},{8,2,2,0,},{9,2,1,0,},{10,2,1,0,},{11,1,0,0,},{12,3,0,1,},{13,0,0
,0,},{14,2,0,0,},{15,3,0,1,}},
            {{0,2,1,0,},{1,5,0,3,},{2,1,0,0,},{3,2,0,0,},{4,3,0,1,},{5,1,0,0,},{6,1
,1,0,},{7,4,0,2,},{8,2,2,0,},{9,2,1,0,},{10,2,1,0,},{11,1,0,0,},{12,1,0,1,},{13,2,0
,0,},{14,0,0,0,},{15,1,0,1,}},
            {{0,3,2,0,},{1,4,0,2,},{2,2,1,0,},{3,3,1,0,},{4,2,0,0,},{5,2,1,0,},{6,2
,2,0,},{7,3,0,1,},{8,3,3,0,},{9,3,2,0,},{10,2,2,0,},{11,2,1,0,},{12,1,0,0,},{13,3,1
,0,},{14,1,1,0,},{15,0,0,0,}}
            }; // 16x16 Matrix, welche die Umlegungen von einer Ziffer zur Anderen
angibt. Dabei ist der Index der Zeile, die Zahl, die verändert wird. Und der Index
der Spalte, die Zahl in die getauscht wird.
```