Aufgabe HexMax - Dokumentation

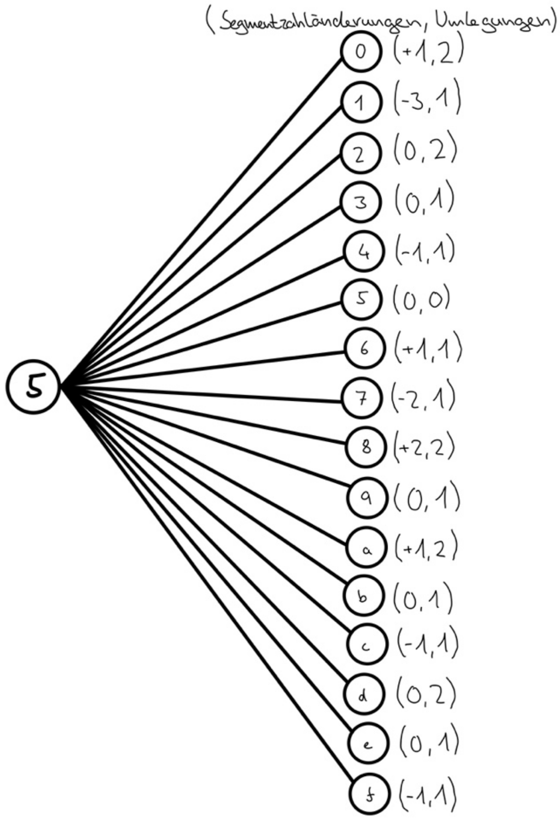
Lösungsansatz:

Mein Programm folgt dem Ansatz zu prüfen, ob aus der Eingabezahl bei der gegebenen Anzahl an Umlegungen, für jede Stelle die größtmögliche Zahl „f" generiert werden kann bzw. welches die maximal generierbare Ziffer ist. Dabei wird geprüft, ob dies, unter Berücksichtigung der stellenübergreifenden Effekte, eine insgesamt valide Lösung darstellt.
Das Programm führt dafür eine Tiefensuche (preorder) in einem Suchbaum durch, der für die Prüfung der jeweiligen Lösungsmöglichkeit dynamisch erzeugt wird. Es werden systematisch mögliche Lösungen ermittelt und auf Validität und Optimalität geprüft.
Um die Effizienz des Programms zu steigern, werden bei der Prüfung des Suchbaums die Prinzipien der Memoisation und des Prunings angewandt, indem nach jedem Prüfvorgang, die für folgende Prüfvorgänge nicht mehr relevanten Abschnitte des Suchbaums ausgeschlossen werden.

Programmatische Umsetzung:

Nach dem Einlesen des Datensatzes aus einer Textdatei werden im ersten Schritt durch das Programm die später analysierten Teil-(Such)bäume erzeugt. Für jede Stelle der Eingabezahl wird ein Teilbaum angelegt, der die Information enthält, wie viele Darstellungssegmente entfernt bzw. hinzugefügt werden müssen, um alle anderen möglichen Zahlen zu erzeugen, sowie wie viele Umlegungen dafür nötig sind.

Abbildung 1 zeigt ein Beispiel für einen solchen Teilbaum für die Zahl „Fünf".
Die Tatsache, dass in diesem Beispiel auch kleinere Zahlen als „Fünf" aufgeführt sind zeigt, dass es sich in diesem Beispiel um den Teilbaum einer der ersten nachfolgenden Stelle der Eingabezahl handelt - bei der Erzeugung des Teilbaums der ersten Stelle der Eingabezahl wäre dies nicht der Fall, da ja explizit nach Möglichkeiten gesucht wird, die Eingabezahl zu vergrößern.
In Klammern wird dargestellt, wie sich die Anzahl der benötigten Segmenten ändert sowie wie viele Umlegungen notwendig sind, um diese umzusetzen.
Eine Änderungen der Anzahl der benötigten Segmente, die von dieser Ziffer benutzt werden, wird dabei nur wenn sie positiv ist, auch als Umlegung gezählt, um eine spätere Doppelzählung bei der entsprechenden Umkehrrelation (Hinzufügen/Entfernen) zu vermeiden.



zur Darstellung der Segmentzahländerung und Anzahl der benötigten Umlegungen

Realisiert wird der Teilbaum, indem ein Objekt (Digit) erstellt wird, dass die Informationen abspeichert und für die spätere Verwendung einfach zugänglich macht.
Anschließend werden die Informationen der einzelnen Teilbäume in Listen zusammengefasst, die den einzelnen Objekten zugeordnet werden. Dabei wird für jede Ziffer der Eingabezahl eine Liste erstellt, die, unter Einbeziehung der Informationen aller nachfolgenden Stellen, für alle möglichen generierbaren Segementzahländerungen speichert, wie viele Umlegungen mindestens notwendig sind, um diese jeweils zu erreichen.
An diesem Punkt ist die Vorbereitungsphase beendet und die eigentliche Tiefensuche nach der optimalen Lösung beginnt.
Prinzipiell bestünde die Möglichkeit, diese Suche rekursiv oder iterativ durchzuführen. Ich habe mich für den iterativen Ansatz entschieden, obwohl die rekursive Lösung minimal schneller und durch die automatische Erzeugung des Callstacks etwas weniger aufwendig ist, um bei zu großen Datensätzen den „Recursion limit exceeded“-Error zu vermeiden.

Die Tiefensuche macht es sich zu Nutze, dass die anfangs erzeugten Teilbäume wiederholt eingesetzt werden können, da die entsprechenden Objekte während der Suche nicht abgeändert werden. Beginnend mit der ersten Ziffer wird für jede Änderungsmöglichkeit dieser Ziffer, der entsprechende Suchbaum der nachfolgenden Ziffer dem Callstack, implementiert als Deque (kleinere Laufzeit für Einfügoperationen), hinzugefügt. Durch dieses Vorgehen wird sozusagen ein großer Gesamtsuchbaum erstellt.

Abbildung 2 zeigt einen allgemeingültigen Gesamtsuchbaum, wobei die Wurzel Z_1 der ersten Ziffer der Eingabezahl, sowie die Verästelungen mit f und 0, der maximalen bzw. minimalen Hexzahl entspricht.
Die Tiefensuche beginnt mit dem höchstmöglichen Ast und fügt dem entsprechenden Objekt die Informationen über die Bilanz an zu viel bzw. zu wenig benutzen Segmenten sowie die übrigen zur Verfügung stehenden Umlegungen hinzu. Eine valide Lösung zeichnet sich dadurch aus, dass diese beiden Bilanzen ausgeglichen sind.

Der Lösungsbaum wird nun dahingehend durchlaufen, dass, beginnend mit der größtmöglichen Zahl (in der Beispiel-abbildung 2 von unten nach oben), geprüft wird, ob dieser Ast eine valide Lösung darstellt.
Während dieses sich wiederholenden Prozesses wird anhand der vorher erzeugten Listen geprüft, ob es noch möglich ist, restliche Segmente mit den verbleibenden Umlegungen „unterzubringen“.

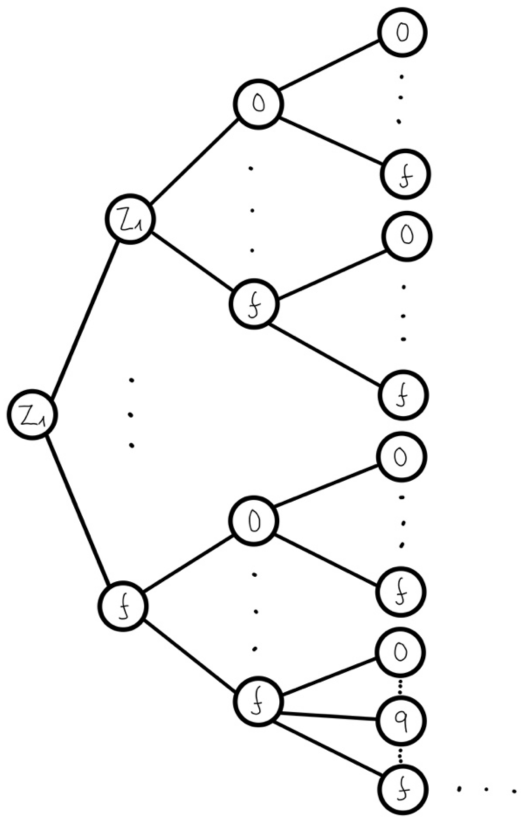


Abbildung 2: Darstellung eines allgemeingültigen Gesamtsuchbaums

Falls dies für einen bestimmten Lösungspfad im Suchbaum nicht mehr möglich ist, wird der Durchlauf abgebrochen. Der Suchbaum wird beschnitten („gepruned“), dieser Pfad nicht weiterverfolgt und einer Liste an Callstack-Elementen hinzugefügt, die zu keiner validen Lösung geführt haben. Durch diese Memoisation wird vermieden, dass bei späteren Durchläufen nicht zielführende Pfade erneut geprüft werden.
Der Vorgang wiederholt sich solange bis eine valide Lösung gefunden ist.
Bei der ersten Ziffer der Eingabezahl werden nur mögliche Ziffern ausprobiert, die größer oder gleich der Ausgangsziffer sind, da sonst die Bedingung einer größeren Gesamtzahl nicht erfüllt wird.
Wenn ein Lösungspfad komplett durchlaufen wird, d.h. bis zur letzten Ziffer der Eingabezahl, nicht unterbrochen wurde, stellt dieser die optimale Lösung dar, da die Prüfung ja in absteigender Größe der möglichen Zahlen erfolgt. Die kleinste mögliche Lösung ist dabei die ursprüngliche Eingabezahl selbst, falls durch Umlegung der Segmente keine Erhöhung der Zahl möglich ist.

Das Ergebnis wird letztendlich in Form einer Textdatei ausgegeben und die Ausführung beendet.
Die Funktionen elaborateConversion() und display() dienen ausschließlich dem Zweck den Umlegungsprozess von der Eingabezahl zur optimalen Ausgabezahl zu visualisieren, so wie dies in der Aufgabenstellung für kleinere Datensätze gefordert ist.

Die Aufgabenstellung lässt sich auf vielseitige Weise inhaltlich erweitern. Nachfolgend sind einige Beispiele von möglichen Vorgaben beschrieben, die in meinem Programm auch zusätzlich implementiert sind:

- Unendlich Umlegungen, aber eingeschränkte Anzahl an Stellen und Segmenten
  - Der einzige Unterschied in der Programm-Logik ist, dass die verbleibenden Umlegungen ignoriert werden und nur nach noch *übrigen* Segmenten geschaut wird.
- Unendlich Segmente, aber limitierte Umlegungen und Ziffern
  - Analog zur ersten Erweiterung wird in diesem Fall nur auf die restlichen Umlegungen geachtet, jedoch nicht auf Segmentunterschiede.
- Unendliche Anzahl an Stellen, aber limitierte Umlegungen und Segmente
  - Hier wird geprüft, wie viele Segmente von anderen Ziffern mit den verfügbaren Umlegungen entfernt werden können, um sie als eine oder mehrere Einsen (nutzt nur zwei Segmente) oder einmalig als Sieben (nutzt drei Segmente, falls die entfernten Segmente ungerade sind) der gegebenen Zahl voranzustellen - es wird das Prinzip verfolgt, dass eine Zahl mit mehr Stellen immer größer ist als eine Zahl mit weniger Stellen, selbst wenn die einzelnen Ziffern kleiner sind – die Maximierung der nachfolgenden Stellen bleibt davon unberührt.

Inklusive dieser Erweiterungen unterschiedet sich das Programm nicht erheblich vom Programm zur ursprünglichen Aufgabenstellung. Die Modifikationen lassen sich jedoch beliebig schichten, z.B. um unendliche Umlegungen und unendliche Ziffern zu ermöglichen. Weiterhin könnte man anstatt nach dem Maximum auch nach dem Minimum suchen. Auch diese Änderungen sind eher trivial und benötigen nur minimale Codeanpassungen, weshalb ich diese nicht implementiert habe.

Laufzeitkomplexität:

O(n, c) = O(constructSubTrees) + O(constructLists) + O(depthFirstSearch)

O(constructSubTrees) = 16 n = n

- Erstellung eines Objektes für jede Ziffer (n-mal)
- Konstante Erstellungszeit (16)

O(constructLists) = (n · (16 + (2c · 2c))) = n · c^2

- Erstellung der Liste für jedes Objekt (n-mal)
- Konstante Iteration über mögliche Änderungen (16)
- Geschachtelte Iteration über mögliche Segmentzahländerungen, abhängig von Umlegungen (-c bis c; 2c), also $2c^2$

O(depthFirstSearch) = n · c · 2c = n · c^2

- Für jede Ziffer (n-mal) gibt es $2c^2$ unterschiedliche Möglichkeiten für Umlegungen und Segmentzahländerungen
- Sobald sie sich doppeln, hilft die Memoisation

O(n, c) = n · c^2

Hierbei werden nicht einzelne Operation, wie das Suchen in einem Array (O(n)) inkludiert. Es ist eine allgemeine Abschätzung der Laufzeit, so lässt sich für diese Abschätzung auf eine allgemein pseudo polynomische und somit wahrscheinlich exponentielle Laufzeit schließen.

Ohne Memoisation würde eine Laufzeitkomplexität von 16^n vorliegen, wegen der 16 Abzweigungen des Baumes von jedem Knoten aus, wobei die Tiefe n wäre.

Trotzdem liegt die Vermutung nah, dass dieses Problem NP-vollständig ist, aufgrund von der rekursiv-exponentiellen Natur.

Speicherkomplexität:

Ohne Memoisation würde der Baum eine Tiefe von n haben und jeder Knoten würde in 16 weiteren resultieren, wodurch der Callsatck einen exponentiellen Speicher von 16^n bräuchte. Mit Memoisation gibt es jedoch keine doppelten Verzweigungen mehr, wodurch eine polynomische Speichernutzung entsteht.

Wichtigsten Auszüge aus dem Programm-Code:

Vorbereitungsfunktionen, zum Erstellen der Objekte und Listen

```python
class Digit: # Klassen, zum Abbilden einer einzelnen Ziffer

    def __init__(self, value, digitNum): # Definition der Attribute einer Ziffer
        self.value = value # Zahlenwert
        self.originalSegments = segments[int(value, 16)] # verwendete Segmente
        self.originalBinary = binary[int(value, 16)] # Binärkodierung
        self.digitNum = digitNum # Stelle der Ziffer in der Zahl
        self.possibleChanges = {} # hashmap, zum Nachschlagen möglicher Änderungen dieser Ziffer und deren Segmentzahländerungen und Umlegungen
        self.bestPossibleList = {} # hashmap, zur Darstellung der minimalen Umlegungen nötig, um eine Segmentzahländerung zu erreichen, bei noch n - digitNum übrigen Ziffern

    def constructTable(self): # Funktion, zur Erstellung der possibleChanges hashmap
        rangeStart = 0
        if self.digitNum == 1:
            rangeStart = int(self.value, 16)
        for hexNum in range(rangeStart, 16): # Iteration über alle Hexzahl (0 bis 15)

            segmentChange = segments[hexNum] -  self.originalSegments

            xorDiff = bin(int(binary[hexNum], 2) ^ int(self.originalBinary, 2))[2:]
            sumAcross = sum([int(x) for x in xorDiff])
            swapChange = (sumAcross + segmentChange) / 2

            change = (segmentChange, swapChange)
            self.possibleChanges[hexNum] = change

def constructSubTrees(hexNum): # Funktion, zur Erstellung der Objekte pro Ziffer

    subTreeObjects = []
    cnt = 1

    for digit in hexNum: # Iteration über alle Ziffern der Eingabezahl
        digitObject = Digit(digit, cnt)
        digitObject.constructTable()
        subTreeObjects.append(digitObject)
        cnt += 1

    return subTreeObjects

def constructLists(subTrees): # Funktion, zur Erstellung der bestPossibleList hashmap
    bestSoFar = {}
    for subTree in subTrees[::-1]: # Iteration über alle erstellten Ziffernobjekte, um diese zu aktualisieren
        best = {}
        for changes in subTree.possibleChanges.values(): # Iteration über alle möglichen Änderungen einer Ziffer, um Minimen zu ermitteln
            if changes[0] in best:
                if changes[1] < best[changes[0]]:
                    best[changes[0]] = changes[1]
            else:
                best[changes[0]] = changes[1]
        tempBestSoFar = {}
        for segChange in best: # Iteration über die hashmap zur Darstellung der minimalen Umlegungen nötig, um eine Segmentzahländerung zu erreichen, bei dieser Ziffer
            if bestSoFar != {}:
                for otherSegChange in bestSoFar: # Iteration über die hashmap zur Darstellung der minimalen Umlegungen nötig, um eine Segmentzahländerung zu erreichen, bei den in der Eingabezahl noch nachfolgenden Ziffern zusammen
                    if segChange + otherSegChange in tempBestSoFar:
                        if best[segChange] + bestSoFar[otherSegChange] < tempBestSoFar[segChange+otherSegChange]:
                            tempBestSoFar[segChange+otherSegChange] = best[segChange] + bestSoFar[otherSegChange]
                    else:
                        tempBestSoFar[segChange+otherSegChange] = best[segChange] + bestSoFar[otherSegChange]
            else:
                bestSoFar = best
        bestSoFar = tempBestSoFar
        tempBestSoFar = {}
        subTree.bestPossibleList = bestSoFar
    return subTrees # Rückgabe der aktualisierten Ziffernobjekte
```

Funktion für die eigentliche Tiefensuche

```python
def depthFirstSearch(subTree, subTrees, swaps): # Funktion zum Ausführen einer Tiefensuche auf dynamisch erzeugten Bäumen

    started = {}
    none = []

    stack = deque()
    stack.append((subTree, (0, 0), []))

    while stack: # Iteration über den sich ständig verändernden Callstack
        nextObject = stack.pop()

        subTree = nextObject[0]

        if subTree == "": # Prüfung, ob noch Ziffern übrig sind
            if nextObject[1][0] == 0 and nextObject[1][1] <= swaps: # Prüfung, ob die Lösung valide ist
                strArray = [hex(x)[2:] for x in nextObject[2]]
                string = "".join(strArray)
                return string
            continue

        attributesId = ",".join([str(x) for x in [nextObject[1], subTree.digitNum]]) # Erstellung einer eindeutigen Identifikation dieser Stelle im Baum, um eine Memoization dieser iterativen Lösung zu ermöglichen

        if len(stack) != 0:
            nextAttributesId = ",".join([str(x) for x in [stack[len(stack)-1][1], stack[len(stack)-1][0].digitNum]]) # Erstellung einer eindeutigen Identifikation des derzeitig nachfolgenden Elements im Callsatck,

        if attributesId in started: # Prüfung, ob der ein vorher begonnener Teilbaum bereits vollständig "umrundet" wurde, ohne Lösung
            for ele in started[attributesId]:
                none.append(ele)
            del started[attributesId]

        if attributesId in none: # Verwendung der Memoisation, Abbrechung wenn bekannt ist, dass dieser Pfad zu keinem Ergebnis führt
            continue

        if len(stack) != 0: # Prüfung, ob auch eine Memoisation dieses Teilbaums noch möglich ist
            if nextAttributesId in started:
                started[nextAttributesId].append(attributesId)
            else:
                started[nextAttributesId] = [attributesId]

        if subTree.digitNum <= len(subTrees): # Prüfung, ob dieser Pfad noch weiter geht
            for newSubTree in subTree.possibleChanges: # Erstellung neuer Callsatckelemente für jede weitere Änderungsmöglichkeit, mit Prüfungen, ob diese zu validen Ergebnissen führen könnten

                newSegmentChange = nextObject[1][0] + subTree.possibleChanges[newSubTree][0]
                if subTree.digitNum == len(subTrees):
                    if newSegmentChange != 0:
                        continue

                newSwapChange = nextObject[1][1] + subTree.possibleChanges[newSubTree][1]
                if newSwapChange > swaps:
                    continue
                if newSwapChange == swaps and newSegmentChange != 0:
                    continue

                if -1 * newSegmentChange not in subTree.bestPossibleList:
                    continue
                else:
                    if newSwapChange + subTree.bestPossibleList[-1 * newSegmentChange] > swaps:
                        continue

                newChosenNumbers = nextObject[2] + [newSubTree]

                if subTree.digitNum <= len(subTrees)-1:
                    newSubTreeToAdd = deepcopy(subTrees[subTree.digitNum])
                    newSubTreeToAdd.value = newSubTree
                    stack.append((newSubTreeToAdd, (newSegmentChange, newSwapChange), newChosenNumbers))
                else:
                    stack.append(("", (newSegmentChange, newSwapChange), newChosenNumbers))
```

Funktionen zur Abbildung der Segmentänderung von Eingabezahl zu Ausgabezahl

```python
def display(num): # Funktion, zur Abbildung einer binäre Siebensegmentkodierung
    start = 0
    end = 7
    parts = []
    rangeEnd = len(num)/7
    for _ in range(int(rangeEnd)):
        parts.append(num[start:end])
        start += 7
        end += 7
    for digit in parts:
        print("\n")
        if digit[0] == "1":
            print("---")
        if digit[5] == "1":
            print("| ", end="")
        else:
            print("  ", end="")
        if digit[1] == "1":
            print("|", end="")
        if digit[6] == "1":
            print("\n---")
        else:
            print("\n")
        if digit[4] == "1":
            print("| ", end="")
        else:
            print("  ", end="")
        if digit[2] == "1":
            print("|", end="")
        if digit[3] == "1":
            print("\n---")
    print("\n")
    print("\n======")

def elaborateConversion(frm, to): # Funktion, zur Abbildung des Übergangs von Eingabezahl zur Ausgabezahl
    changingNum = ""
    str1 = ""
    str2 = ""
    for idx, digits in enumerate(zip(frm, to)):
        bin1 = binaryOfHex(digits[0])
        while len(bin1) != 7:
            bin1 = "0" + bin1
        bin2 = binaryOfHex(digits[1])
        while len(bin2) != 7:
            bin2 = "0" + bin2
        str1 += bin1
        str2 += bin2
    display(str1)
    changingNum = str1

    for idx in range(len(changingNum)):
        digitsInBin = (changingNum[idx], str2[idx])
        if digitsInBin[0] != digitsInBin[1]:

            changingNum = list(changingNum)
            changingNum[idx] = digitsInBin[1]
            changingNum = "".join(changingNum)

            for idx2, digit in enumerate(changingNum):
                if idx2 > idx:
                    if digit == digitsInBin[1] and str2[idx2] != digitsInBin[1]:
                        changingNum = list(changingNum)
                        changingNum[idx2] = digitsInBin[0]
                        changingNum = "".join(changingNum)
                        break
            display(changingNum)
```

Darstellung des Verlaufs der Umlegungen bei den gegebenen Beispielen

hexmax0:

D24
3
→
ee4

```
  |
---
| |
---

---
  |
---
|
---

| |
---
  |

======

---
---
| |
---

---
  |
---
|
---

| |
---
  |

=====

---
|
---
|
---

---
  |
---
|
---

| |
---
  |

======

---
|
---
|
---

---
|
---
|
---

| |
---
  |
```

hexmax1:

509C431B55
8
→
fffea97b55

```
---
|
---
  |
---

---
| |

| |
---

---
| |
---
  |
---

---
|

|
---

| |
---
  |
---
  |
---
  |
---

  |

  |
|
---
| |
---

---
|
---
  |
---

---
```

```
|
---
  |
---

======

---
|
---
|
---

---
| |

| |
---

---
| |
---
  |
---

---
|

|
---

| |
---
  |

---
  |
---
  |
---

  |

  |

|
---
| |
---

---
|
---
  |
---

---
|
---
  |
---


======

…
Restliche Angaben sind beim Ausführen des Programms sichtbar

======

---
|
---
|

---
|
---
|

---
|
---
|
---

---
|
---
|
---

---
| |
---
| |

---
| |
---
  |
---

  |

  |

|
---
| |
---

---
|
---
  |
---

---
|
---
  |
---

======


---
|
---
|

---
|
---
|

---
|
---
|

---
|
---
|
---

---
| |
---
| |

---
| |
---
  |
---

---
  |

  |

|
---
| |
---

---
```

```
|
---
  |
---

---
|
---
  |
---

======
```

hexmax2:

632B29B38F11849015A3BCAEE2CDA0BD496919F8
37
→
ffffffffffffffffd9a9beaee8eda8bda989d9f8

```
---
|
---
| |
---

---
  |
---
  |
---

---
  |
---
|
---

|
---
| |
---

---
  |
---
|
---

---
| |
---
  |
---

|
---
| |
---

---
  |
---
  |
---

---
| |
---
| |
---

---
|
---
|

  |

  |

  |

  |

---
| |
---
| |
---

| |
---
  |


---
| |
---
  |
---

---
| |

| |
---

  |

  |

---
|
---
  |
---

---
| |
---
| |

---
  |
---
  |
---

|
---
| |
---

---
|

|
---

---
| |
---
| |

---
|
---
|
---

---
|
---
|
---

---
```

```
  |
___
|
___

___
|

|
___


  |
___
| |
___

___
| |
___
| |


___
| |

| |
___


|
___
| |
___


  |
___
| |
___


| |
___
  |


___
| |
___
  |
___

___
|
___
| |
___

___
| |
___
  |
___


  |

  |


___
| |
___
  |
___

___
|
___
|


___
| |
___
| |
___

=====




___
|
___
|
___

___
  |
___
| |
___

___
  |
___
|
___


|
___
| |
___

___
  |
___
|
___

___
| |
___
  |
___


|
___
| |
___

___
  |
___
  |
___

___
| |
___
| |
___

___
|
___
|


  |

  |


  |

  |


___
| |
___
| |
___


| |
___
  |


___
| |
___
  |
___

___
| |

| |
___
```

```
  |
  |

---
|
---
  |
---

---
| |
---
| |

---
  |
---
  |
---

|
---
| |
---

---
|

|
---

---
| |
---
| |

---
|
---
|
---

---
|
---
|
---

---
  |
---
|
---

---
|

|
---

  |
---
| |
---

---
| |
---
| |

---
| |

| |
---

|
---
| |
---

  |
---
| |
---

| |
---
  |

---
| |
---
  |
---

---
|
---
| |
---

---
| |
---
  |
---

  |
  |

---
| |
---
  |
---

---
|
---
|

---
| |
---
| |
---
```

```
======
...

Restliche Angaben sind beim Ausführen des Programms sichtbar

======
```

```
---
|
---
|

---
|
---
|

---
|
---
|

---
|
---
|

---
|
---
|

---
|
---
|

---
|
```

```
======

---
|
---
|


---
|
---
|


---
|
---
|


---
|
---
|


---
|
---
|


---
|
---
|


---
|
---
|


---
|
---
|


---
|
---
|


---
|
---
|


---
|
---
|


---
|
---
|


---
|
---
|


---
|
---
|


---
|
---
|


---
|
---
|



  |
---
| |
---

---
| |
---
  |
---

---
| |
---
| |


---
| |
---
  |
---


|
---
| |
---

---
|
---
|
---

---
| |
---
| |


---
|
---
|
---

---
|
---
|
---

---
| |
---
| |
---

---
|
---
|
---


  |
---
| |
---

---
| |
---
| |


---
| |
---
| |
---


|
---
| |
---


  |
---
| |
---

---
| |
---
| |


---
| |
---
  |
---



---
| |
---
```

hexmax3:
0E9F1DB46B1E2C081B059EAF198FD491F477CE1CD37EBFB65F8D765055757C6F4796BB8B3DF7FCAC606DD06
27D6B48C17C09
121
→
ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffefaa98bb8b9dfafeae888dd888ad8ba8ed8888

hexmax4:
1A02B6B50D7489D7708A678593036FA265F2925B21C28B4724DD822038E3B4804192322F230AB7AF7BDA0A6
1BA7D4AD8F888
87
→
ffffffffffffffffffffffffffffffffffffffeb8de88baa8add888898e9ba88ad98988f898ab7af7bda8a61ba7d4ad8f888

hexmax5:
EF50AA77ECAD25F5E11A307B713EAAEC55215E7E640FD263FA529BBB48DC8FAFE14D5B02EBF792B5CCBBE9F
A1330B867E330A6412870DD2BA6ED0DBCAE553115C9A31FF350C5DF993824886DB5111A83E773F23AD7FA81
A845C11E22C4C45005D192ADE68AA9AA57406EB0E7C9CA13AD03888F6ABEDF1475FE9832C66BFDC28964B70
22BDD969E5533EA4F2E4EABA75B5DC11972824896786BD1E4A7A7748FDF1452A5079E0F9E6005F040594185
EA03B5A869B109A283797AB31394941BFE4D38392AD12186FF6D233585D8C820F197FBA9F6F063A0877A912
CCBDCB14BEECBAEC0ED061CFF60BD517B6879B72B9EFE977A9D3259632C718FBF45156A16576AA7F9A4FAD4
0AD8BC87EC569F9C1364A63B1623A5AD559AAF6252052782BF9A46104E443A3932D25AAE8F8C59F10875FAD
3CBD885CE68665F2C826B1E1735EE2FDF0A1965149DF353EE0BE81F3EC133922EF43EBC09EF755FBD740C8E4
D024B033F0E8F3449C94102902E143433262CDA1925A2B7FD01BEF26CD51A1FC22EDD49623EE9DEB14C138A
7A6C47B677F033BDEB849738C3AE5935A2F54B99237912F2958FDFB82217C175448AA8230FDCB3B3869824A
826635B538D47D847D8479A88F350E24B31787DFD60DE5E260B265829E036BE340FFC0D8C05555E75092226E
7D54DEB42E1BB2CA9661A882FB718E7AA53F1E606
1369
→
ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffe88efa9ebe89efa99fbdaa8e8ead88ab899f8e8f9aa9
e9ad88988eda9a99888edad989a8bafd8d888888888888888888888888888888888888888888888888888888888
88888888888888888888888888888888888888888888888888888888888888888888888888888888888888888
88888888888888888888888888888888888888888888888888888888888888888888888888888888888888888
888888888888888888