Aufgabe 3: Hex-Max

DOKUMENTATION

Teilnahme-ID: 60291

Siemen Zielke

7. März 2022

Inhalt

Lösungsidee

Ich trenne die Aufgabe in zwei Teilaufgaben. Als erstes versuche ich die eigentliche Lösung zu berechnen, also die größtmögliche Hex-Zahl zu finden. Danach erst berechne ich den Weg dorthin, unter Berücksichtigung der „Keine-Ziffer-Leeren-Regel“.
Der erste Ansatz, eine Lösung zu finden, hat schon funktioniert, er war jedoch sehr langsam. Deswegen habe ich im zweiten Ansatz einen ganz neuen Algorithmus entwickelt, den ich dann im dritten Ansatz noch einmal verbessert habe, um eine höhere Geschwindigkeit (bzw. eine niedrigere Iterationszahl) zu erreichen.
Diesen dritten Ansatz habe ich dann auch vollständig implementiert.

Ansatz 1

Der erste Ansatz startet bei der höchsten Zahl mit der gleichen Länge wie die Eingangszahl (FFFF…). Er zählt die Anzahl der anders liegenden Stäbchen und prüft, ob diese kleiner gleich der Maximalzahl an Umlegungen ist. Ist sie das, wurde die Lösung gefunden. Wenn nicht, wird die nächst kleinere Zahl ausprobiert, bis die Lösung gefunden wurde. Dieser Ansatz funktioniert zwar sehr sicher, braucht aber mit mittelgroßen Zahlen schon sehr lange.

Ansatz 2

Wichtig für den zweiten Ansatz ist die Tatsache, dass Ziffern, die sich weiter links in der Zahl befinden, einen höheren Wert haben als alle Ziffern rechts daneben zusammen. Das bedeutet, dass

man Ziffern, die weiter links stehen, optimieren kann, ohne auf die rechten Ziffern „Rücksicht“ nehmen zu müssen.

Für den Algorithmus führe ich einen Wert ein: die Balance. Sie ist sozusagen ein Bankkonto, dass verfolgt wie viele Stäbchen aus der Zahl entfernt und zur Seite gelegt wurden bzw. wie viele „Schulden“ man gerade hat. Die Balance ermöglicht es, „halbe“ Züge durchzuführen, die ein Stäbchen aus der Zahl entfernen, aber es noch nicht sofort wieder hinlegen. Wenn eine Aktion die Balance von null entfernt, zähle ich sie als Zug. Nähert sie die Balance null an, zähle ich keinen Zug, da damit ein begonnener halber Zug beendet wird.

Der Algorithmus startet bei der Ziffer ganz links. Er berechnet zunächst, wie viele Züge es kosten würde, die Ziffer zu einem F zu ändern. Wenn es die gegebene Maximalzahl an Zügen zulässt, wird die Ziffer zu einem F geändert. Dabei wird auch die Balance geändert und die Menge an verbrauchten Zügen gespeichert. Wenn nicht, wird ein E ausprobiert usw. Wenn die Ziffer erfolgreich erhöht wurde oder dieselbe Ziffer, die bereits an der Stelle steht, erreicht ist, wird zur nächsten Ziffer weiter rechts gesprungen. Das wird solange wiederholt, bis die Ziffer ganz rechts erreicht wurde. Wenn die Balance dann Null ist, wurde die Lösung gefunden und der Algorithmus ist fertig.

Muss die Balance jedoch noch „ausgeglichen“ werden, wechselt der Algorithmus die Richtung. Die Änderung der letzten Ziffer wird rückgängig gemacht. Dann wird versucht, die größtmögliche Zahl zu finden, die jedoch kleiner sein muss, als beim ersten „Versuch“. Wenn die Ziffer nicht verkleinert werden kann, da mehr Züge dafür nötig wären als noch übrig sind, wird versucht die zweite Ziffer von rechts zu verkleinern. Diese wird natürlich auch vorher auf den Ausgangszustand zurückgesetzt. Sobald eine Ziffer verkleinert wurde wechselt der Algorithmus wieder die Richtung, bewegt sich nach rechts und vergrößert die Ziffern. Das heißt, der Algorithmus läuft im hinteren Teil der Zahl immer wieder hin und her bis irgendwann die Balance Null ist und er sich am rechten Rand befindet.

Mit diesem Vorgehen wird automatisch die höchste Zahl gefunden, da Ziffern immer nur dann verkleinert werden, wenn dafür eine Ziffer weiter links genau so groß bleiben darf, wie sie schon ist.

Ansatz 3

Der zweite Ansatz funktioniert bereits gut bei mittelgroßen bis großen Zahlen. Bei sehr großen Zahlen ist allerdings beim ersten Erreichen des rechten Endes die Balance so hoch, dass es sehr lange dauert, sie auszugleichen. Der dritte Ansatz verhindert, dass die Balance zu hoch wird.

Der Algorithmus berechnet im Voraus für jede Stelle, wie viele Stäbchen in alle Ziffern rechts von ihr hinzugefügt werden könnten und wie viele Stäbchen aus allen Ziffern rechts von ihr entnehmbar wären. Damit werden für jede Stelle zwei Limits festgelegt: erstens wie hoch und zweitens wie niedrig die Balance sein darf, nachdem die Ziffer an dieser Stelle optimiert wurde.

Dann fängt er wieder bei der Stelle ganz links an und versucht diese, wie in Ansatz 2 beschrieben, zu einem F zu ändern. Diesmal nur unter Berücksichtigung der Balance-Limits. Da das Limit für die letzte Stelle null ist, muss die Balance also, wenn diese Stelle erreicht wird, durch Optimierung der Ziffer auf null fallen und der Algorithmus ist fertig.

Der einzige Fall in dem dies nicht funktioniert ist, wenn die Limits nicht eingehalten werden können, da das zu viele Züge kosten würde. Ein Beispiel dafür ist 74 als Eingangszahl mit 3 als Maximalzahl an Umlegungen. Die Limits für die 16er-Stelle wären -1 und 3. Für die 1er-Stelle sind die Limits immer 0. Nun würde die 7 zu einem F optimiert werden. Das kostet 3 Züge und ändert die Balance auf -1. Eine Balance von -1 kann dann bei der 4 aber nicht ausgeglichen werden, da man keinen Stäbchen mehr verschieben darf. Man kann nur noch ein Stäbchen entfernen um die Balance auf 0 zu bringen. Dadurch kann man aber keine existierende Ziffer erzeugen.

Um das Problem zu lösen, geht der Algorithmus eine Stelle zurück nach links. Dort versucht er es mit der nächstkleineren Ziffer. Dabei wird aber bei der Berechnung der Balance und der benötigten Züge so getan, als hätte der erste Versuch nie stattgefunden. Sollte mit den übrigen Zügen keine kleinere Ziffer erreicht werden können, wird noch eine Stelle nach links gegangen und dort dasselbe versucht. (Dieser Weg nach links stoppt spätestens beim linken Ende der Zahl, da die Balance dort 0 ist und man keine Züge verbraucht, wenn man die Ziffer so wie in der Ausgangszahl lässt.)
Sobald eine Ziffer verkleinert werden konnte, geht der Algorithmus wieder nach rechts.

Wegfindung

Wenn die Start- und die Endzahl feststehen, ist es relativ einfach den Weg dazwischen zu berechnen. Die größte Schwierigkeit liegt dabei in der Einhaltung der „Keine-Ziffer-Leeren-Regel“.

Ich berechne zunächst die Positionen aller Stäbchen, die entfernt bzw. dazugelegt werden müssen. Um jedem Stäbchen, das weggenommen wird, einen Ort zuzuweisen, wo es hinkommt, iteriere ich über die Stäbchen, die entfernt werden müssen, und schaue, ob in derselben Ziffer auch ein Stäbchen hinzugefügt werden muss. Dann lege ich das Stäbchen einfach innerhalb derselben Ziffer um, wodurch die Ziffer, selbst wenn es das einzige Stäbchen ist, nicht geleert wird. Wenn kein Stäbchen mehr in die gleiche Ziffer muss, lege ich das entnommene Stäbchen an den ersten Ort in der Hinzufügen-Liste und entferne diesen aus der Liste. Danach gebe ich schließlich den Zwischenstand der Zahl aus und mache weiter mit dem nächsten Stäbchen.

Umsetzung

Hier folgt jetzt die Erläuterung zu meiner Implementierung in Python. Zur Berechnung der größtmöglichen Hex-Zahl benutze ich Ansatz 3.

Datenstruktur

Hex-Zahlen stelle ich durch ein Array dar. Dabei ist jedes Element eine Ziffer beginnend mit der linken Ziffer.
Für die Ziffern definiere ich zwei unterschiedliche Formate:
Das eine Format enthält den Ziffernwert als Integer, deswegen nenne ich sie Int.
Im anderen Format wird jede Ziffer durch eine 7-bit-Zahl dargestellt, in der jedes Bit für ein Segment der Anzeige steht, und zwar von links nach rechts in der in Abbildung 1 dargestellten Reihenfolge. Dieses Format nenne ich Seven-Segment-Number oder kurz SSN. Die Hex-Ziffer 7 wäre beispielsweise 1010010.
Um von Int zu SSN umzuwandeln, verwende ich ein konstantes Array (SSN_CODE) der Länge 16, welches die SSN aller Ziffern in der Reihenfolge 0 → F enthält. Dann ist SSN_CODE[A_int] = A_ssn.

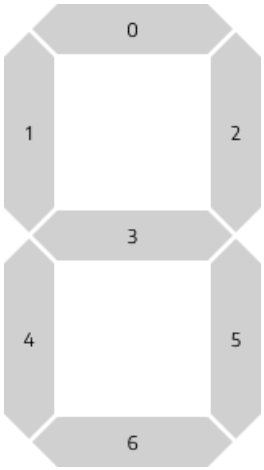


Abbildung 1

Solution.py

Dieser Abschnitt beschreibt die Implementierung der Findung der besten Hex-Zahl im Modul solution.py. Das Modul besteht aus einer Funktion, der die Eingangszahl *initial_digit* und das Umlegungslimit *max_moves* übergeben werden.

In der Funktion werden zunächst die Balance-Limits vorberechnet. Dafür iteriere ich von rechts nach links über die Ziffern in INITIAL_NUM und zähle die gesetzten Bits in der SSN-Darstellung. Die Ziffer mit den meisten Segmenten ist 8 mit sieben Segmenten und die mit den wenigsten ist 1 mit zwei Segmenten. Deswegen ist *7 - bit_count = Limit nach oben* (Balance darf bei der Ziffer nicht höher sein) und *2 - bit_count = Limit nach unten* (Balance darf nicht niedriger sein). Für alle Ziffern ab der

zweiten Stelle addiere ich noch die Limits der Ziffer rechts daneben dazu, da alle Ziffern rechts daneben beim Balance ausgleichen „mithelfen“ können. Die berechneten Balance Limits speichere ich in den Arrays *balance_max_limits* und *balance_min_limits*. Danach entferne ich das erste Element aus den beiden Arrays, sodass alle Elemente eine Stelle nach links rutschen und jedes Element für seine Stelle angibt, wie viel Balance rechts daneben ausgeglichen werden kann.

Vor dem „Hauptteil“ des Algorithmus werden noch ein paar Variablen initialisiert: Es wird eine Int-Zahl *optimized_num* generiert, die jedoch zunächst an jeder Stelle die unmögliche Ziffer G bzw. 16 hat und im Laufe des Programms sich der optimalen Zahl annähern wird. Da der Algorithmus es ermöglichen muss, Züge rückgängig zu machen und damit auch die Balance und die verbrauchten Züge zurückzusetzen, speichere ich sie in den Listen *balance_digitwise* und *moves_digitwise*. Darin beschreibt jedes Element nur die relative Änderung, die durch die dazugehörige Ziffer „verursacht“ wurde, sodass zu jeder Zeit die Summe aller Elemente einer der Listen die Balance bzw. (bis dahin verbrauchten) Moves ergibt.

In der Haupt-while-Schleife wird dann über die Stellen von links nach rechts iteriert und für jede Stelle die Funktion *optimize_digit()* ausgeführt. Ihr wird die Position der Stelle und die bevorzugte Ziffer (in der Regel F) übergeben. Die Funktion probiert nacheinander aus, die aktuelle Ziffer durch die Ziffern F bis 0 zu ersetzten. Das Ersetzten durch eine Ziffer gelingt nicht, wenn dadurch entweder die Balance die für die Ziffer festgelegten Limits überschreiten oder die Maximalzüge (Moves) nicht mehr ausreichen würden.

```
mask = initial_digit_ssn ^ digit_attempt_ssn
remove_sticks = (initial_digit_ssn & mask).bit_count() #count sticks to remove
add_sticks = (digit_attempt_ssn & mask).bit_count() #count sticks to add
new_balance = balance
moves = 0
moves += max(min(new_balance, 0) + remove_sticks, 0)
new_balance += remove_sticks
moves += max(min(-new_balance, 0) + add_sticks, 0)
new_balance -= add_sticks
```

Auszug 1 (aus solution.py)

Um dies zu errechnen, wird das bitweise XOR der SSN-Darstellungen der beiden Ziffern gebildet. Mit dieser Maske werden dann per bitweisem AND die jeweiligen Bits in den beiden Ziffern extrahiert, die sich beim Wechsel zur anderen Ziffer ändern würden. Zählt man nun die Bits in der „alten“ Zahl, bekommt man die Anzahl der Stäbchen, die weggenommen werden müssten. Genau so sind die Bits in der „neuen“ Zahl, wie viele Stäbchen hinzugefügt werden müssten. Die unveränderten Stäbchen wurden durch die Maske herausgefiltert.

Daraus werden dann die Moves, die das Entfernen der Stäbchen kosten würde, berechnet. Dabei ist zu berücksichtigen, dass nur Stäbchenbewegungen, die den Betrag der Balance erhöhen würden, als Move zählen, da Bewegungen, die den Betrag verringern, „angefangene“ Moves vollenden.
Es gibt also vier zu unterscheidende Fälle (hier für die Stäbchen, die der Zahl entnommen werden):

1. Die Balance ist größer als 0:
   Durch min(new_balance, 0) wird so gerechnet als wäre sie 0, also wie in Fall 2.
2. Die Balance ist 0:
   remove_sticks wird zur Balance addiert und das Ergebnis sind die benötigten Moves.
3. Die Balance ist negativ, aber so nah an 0, dass sie durch Addieren von remove_sticks 0 erreicht oder überschreitet:
   Da remove_sticks zur Balance addiert wird, ist das Ergebnis entsprechend der durch die Balance „aufgenommen“ Bewegungen kleiner.

4. Die Balance kann alle Bewegungen „aufnehmen“, ist danach aber immer noch negativ: Negative Werte werden durch max(…, 0) herausgefiltert, sodass 0 Moves als Ergebnis herauskommt.

Bevor dieselben Fälle auf die zur Zahl hinzugefügten Stäbchen angewandt werden können, müssen die entnommenen Stäbchen zur Balance addiert werden. Die Berechnung für die Stäbchen, die der Zahl hinzugefügt werden, geschieht analog zu der oberen, mit dem einzigen Unterschied, dass Anfangs die Balance negiert wird, um den umgekehrten Effekt zu bewirken.

Jetzt wird überprüft, ob sich die „neue“ Balance zwischen den Limits befindet und ob die Moves nicht mehr als die noch zur Verfügung stehenden Züge sind und dementsprechend entweder zur nächsten Ziffer gesprungen oder die Änderung in *optimized_num*, *moves_digitwise* und *balance_digitwise* gespeichert. Dann gibt die Funktion True zurück. Falls die Schleife ganz durchläuft, ohne dass eine der Ziffern funktioniert hat, wird False zurückgegeben.
Der Rückgabewert der Funktion wird in der Haupt-while-Schleife überprüft. Ist er False, wird der Pointer zwei Ziffern zurückgesetzt. In der nächsten Iteration wird er zuerst wieder einen nach vorne gesetzt, sodass man insgesamt eine Ziffer zurück gegangen ist.
Da das Attribut *preferred_digit* der Funktion *optimize_digit()* die nächstkleinere Zahl nach der aktuellen Ziffer in *optimized_num* ist, werden also kleiner Ziffern ausprobiert als beim letzten Mal. (Das ist auch der Grund dafür, dass optimized_num am Anfang mit Gs / 16en befüllt wird, sodass anfangs immer bei F angefangen wird.)
Wenn der Pointer 0 erreicht, ist die While-Schleife fertig, und *optimized_num* wird zurückgegeben.

Way.py

Dieser Abschnitt beschreibt die Wegfindung von der Ausgangszahl zur optimierten Zahl im Modul way.py. Das Modul besteht aus einer Funktion, der die Start- und Endzahl übergeben werden. Als erstes wird durch XOR eine Maske für die Veränderung jeder Ziffer erstellt und mit dieser die von der Start- zur Endzahl geänderten Bits extrahiert und in *rems* bzw. *adds* gespeichert. Eine 1 in rems bedeutet also, dass das Segment an der Position ausgeschaltet und eine 1 in adds, dass das Segment an der Position eingeschaltet werden muss.

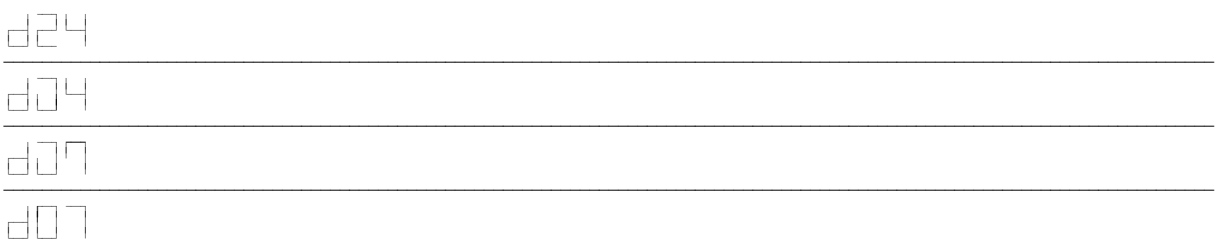
Ich iteriere also über die Stellen in rems und innerhalb jeder Stelle dann auch über die Bits (mit RIGHT SHIFT). Wenn ein Bit gesetzt ist, also ein Stäbchen entfernt werden soll, suche ich zunächst an der aktuellen Stelle in adds nach einem gesetzten Bit, also nach einem Ort, wo ein Stäbchen hinmuss. (Das Verschieben eines Stäbchens innerhalb einer Ziffer geht vor.) Wird keins gefunden, suche ich von vorne alle Ziffern durch, bis ein Bit in adds bzw. ein Ort, wo ein Stäbchen hinmuss, gefunden ist. Dann setzte ich die beiden Bits in rems und adds auf 0 und führe die Verschiebung in current_num aus.

Als letztes generiere ich mit dem Modul ssn_drawer eine Ascii-Darstellung (siehe Beispielausgaben) von current_num und hänge sie mit allen anderen Zwischenschritten zu einem String zusammen, der zurückgegeben wird.

Beispiele

Ich habe hier die kompletten Wege für hexmax[0-2].txt und die Ergebniszahlen für hexmax[3-5].txt eingefügt. Die Wege für hexmax[3-5].txt und auch noch einmal für hexmax[0-2].txt finden sich in output[0-5].txt

```
--------------------------------------------------------------------------------------------
Start: d24
...3 steps ...
End:   d07
--------------------------------------------------------------------------------------------
```



Ausgabe 1: hexmax0.txt

```
----------------------------------------------------------------------------------------------
Start: 509C431b55
...8 steps ...
End:   FFFEA97b55
----------------------------------------------------------------------------------------------
```

Ausgabe 2: hexmax1.txt

```
----------------------------------------------------------------------------------------------
Start: 632b29b38F11849015A3bCAEE2CdA0bd496919F8
...37 steps ...
End:   FFFFFFFFFFFFFFFFd9A9bEAEE8EdA8bdA989d9F8
----------------------------------------------------------------------------------------------
```

Ausgabe 3: hexmax2.txt

```
Start:
0E9F1db46b1E2C081b059EAF198Fd491F477CE1Cd37EbFb65F8d765055757C6F4796bb8b3dF7FCAC606dd0627d6b48
C17C09
...121 steps ...
End:
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFAA98bb8b9dFAFEAE888dd888Ad8bA8
EA8888
----------------------------------------------------------------------------------------------
```

Ausgabe 4: hexmax3.txt

```
----------------------------------------------------------------------------------------------
Start:
1A02b6b50d7489d7708A678593036FA265F2925b21C28b4724dd822038E3b4804192322F230Ab7AF7bdA0A61bA7d4A
d8F888
...87 steps ...
End:
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEb8dE88bAA8Add888898E9bA88Ad98988F898Ab7AF7bdA8A61bA7d4A
d8F888
----------------------------------------------------------------------------------------------
```

Ausgabe 5: hexmax4.txt

```
----------------------------------------------------------------------------------------------
Start:
EF50AA77ECAd25F5E11A307b713EAAEC55215E7E640Fd263FA529bbb48dC8FAFE14d5b02EbF792b5CCbbE9FA1330b8
67E330A6412870dd2bA6Ed0dbCAE553115C9A31FF350C5dF993824886db5111A83E773F23Ad7FA81A845C11E22C4C4
5005d192AdE68AA9AA57406Eb0E7C9CA13Ad03888F6AbEdF1475FE9832C66bFdC28964b7022bdd969E5533EA4F2E4E
AbA75b5dC11972824896786bd1E4A7A7748FdF1452A5079E0F9E6005F040594185EA03b5A869b109A283797Ab31394
941bFE4d38392Ad12186FF6d233585d8C820F197FbA9F6F063A0877A912CCbdCb14bEECbAEC0Ed061CFF60bd517b68
79b72b9EFE977A9d3259632C718FbF45156A16576AA7F9A4FAd40Ad8bC87EC569F9C1364A63b1623A5Ad559AAF6252
052782bF9A46104E443A3932d25AAE8F8C59F10875FAd3Cbd885CE68665F2C826b1E1735EE2FdF0A1965149dF353EE
0bE81F3EC133922EF43EbC09EF755Fbd740C8E4d024b033F0E8F3449C94102902E143433262CdA1925A2b7Fd01bEF2
6Cd51A1FC22Edd49623EE9dEb14C138A7A6C47b677F033bdEb849738C3AE5935A2F54b99237912F2958FdFb82217C1
75448AA8230FdCb3b3869824A826635b538d47d847d8479A88F350E24b31787dFd60dE5E260b265829E036bE340FFC
0d8C05555E75092226E7d54dEb42E1bb2CA9661A882Fb718E7AA53F1E606
...1369 steps ...
End:
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFF88EFA9EbE89EFA99FbdAA8E8EAd88Ab899F8E8F9AA9E9Ad88988EdA9A99888EdAd989A8bAFd8A8888
8888888888888888888888888888888888888888888888888888888888888888888888888888888888888888888888
8888888888888888888888888888888888888888888888888888888888888888888888888888888888888888888888
888888888888888888888888888888888888888888888888888888888888
----------------------------------------------------------------------------------------------
```

Ausgabe 6: hexmax5.txt

Interessant sind die Laufzeiten des Programmes. Ich habe keine genauen Messungen vorgenommen, aber bei hexmax[0-4].txt läuft das Programm so schnell durch, dass es aussieht, als wäre es sofort fertig. Bei hexmax5.txt läuft der erste Teil genauso schnell durch. Nur die Wegberechnung dauert etwa 15 Sekunden. Die Wegfindung ist also viel langsamer als die Lösungsfindung. Ich vermute, dass das an dem doch sehr langen String und der damit verbundenen großen Datenmenge, die in den Arbeitsspeicher geschrieben werden muss, liegt.

Quellcode

Der gesamte Quellcode befindet sich in hexmax.py.zip. Er umfasst 6 Module: __main__.py, solution.py, way.py, ssn_drawer.py, ascii_box.py und ssncode.py. Ersteres kommuniziert mit dem Nutzer, liest die Aufgaben ein, ruft die Berechnung auf und speichert das Ergebnis in einer Datei. solution.py und way.py sind die Hauptteile des Algorithmus und deswegen auch schon im Abschnitt Umsetzung besprochen und hier noch einmal vollständig eingefügt. ssn_drawer.py und ascii_box.py

sind für das Generieren einer Ascii-Darstellung einer Sieben-Segment-Anzeige zuständig. ssncode.py enthält einfach das Array SSN_CODE, mit dem Int-Ziffern zu SSN Ziffern umgewandelt werden.

Beim Ausführen des Programms muss der Pfad zur Eingangs-Datei als Startargument übergeben werden. Optionale Startargumente sind -p zum Ausgeben des Weges in die Konsole und -pf <Dateipfad> um das Ergebnis gefolgt vom Weg in der angegebenen Datei zu speichern.

```
from ssncode import SSN_CODE

def get_best_number(initial_num, max_moves):
    #setup initial constants
    INITIAL_NUM = initial_num
    MAX_MOVES = max_moves
    DIGITS = len(INITIAL_NUM)

    #precalculate balance limits for each digit
    ssncode_stick_counts = [x.bit_count() for x in SSN_CODE]
    max_stick_count = max(ssncode_stick_counts)
    min_stick_count = min(ssncode_stick_counts)
    balance_max_limits = [0]
    balance_min_limits = [0]
    rev_num = INITIAL_NUM[:]
    rev_num.reverse()
    for digit in rev_num:
        balance_max_limits.insert(0, max_stick_count - SSN_CODE[digit].bit_count() +
balance_max_limits[0])
        balance_min_limits.insert(0, min_stick_count - SSN_CODE[digit].bit_count() +
balance_min_limits[0])
    balance_max_limits.pop(0)
    balance_min_limits.pop(0)
    del ssncode_stick_counts, max_stick_count, min_stick_count, rev_num

    #setup initial variables
    optimized_num = [0x10] * DIGITS #number which approaches solution
    moves_digitwise = [0] * DIGITS #moves each digit caused (according to current
optimized_num state)
    balance_digitwise = [0] * DIGITS #balance change each digit caused (according to current
optimized_num state)
    pointer = DIGITS #pointer to currently processed digit

    #calculate best number
    def optimize_digit(pointer, preferred_digit):
        moves_left = MAX_MOVES - sum(moves_digitwise[:-pointer-1])
        balance = sum(balance_digitwise[:-pointer-1])
        initial_digit = INITIAL_NUM[-pointer-1]
        initial_digit_ssn = SSN_CODE[initial_digit]
        for digit_attempt in range(preferred_digit, -1, -1): #try each digit (preferred to 0)
decreasing attempted digit if it wasnt possible
            #calculate consequences of changing digit to attempt
            digit_attempt_ssn = SSN_CODE[digit_attempt]
            mask = initial_digit_ssn ^ digit_attempt_ssn
            remove_sticks = (initial_digit_ssn & mask).bit_count() #count sticks to remove
            add_sticks = (digit_attempt_ssn & mask).bit_count() #count sticks to add
            new_balance = balance
            moves = 0
            moves += max(min(new_balance, 0) + remove_sticks, 0)
            new_balance += remove_sticks
            moves += max(min(-new_balance, 0) + add_sticks, 0)
            new_balance -= add_sticks
            #check if digit change is possible
            if moves > moves_left: continue #costs to much moves -> lower attempt
            if new_balance > balance_max_limits[-pointer-1] or new_balance <
balance_min_limits[-pointer-1]: continue #pushes balance to high -> lower attempt
            #change digit
            optimized_num[-pointer-1] = digit_attempt
            moves_digitwise[-pointer-1] = moves
            balance_digitwise[-pointer-1] = new_balance - balance
            return True
        return False #cant change digit (somthing a digit before went wrong)
```

```
    while pointer > 0:
        pointer -= 1 #decreas pointer
        if not optimize_digit(pointer, optimized_num[-pointer-1] - 1):
            pointer += 2 #increase pointer cause something went wrong (to redo last digit)

    return optimized_num, sum(moves_digitwise)
```

Quellcode 1: solution.py

```
import ssn_drawer
import ascii_box
from ssncode import SSN_CODE

def generate_way_ascii(start_num, end_num, max_line_width = 70):
    #prepare variables
    start_num = [SSN_CODE[x] for x in start_num]
    current_num = start_num[:]
    end_num = [SSN_CODE[x] for x in end_num]
    masks = [a ^ b for a, b in zip(start_num, end_num)]
    rems = [x & m for x, m in zip(start_num, masks)]
    adds = [x & m for x, m in zip(end_num, masks)]
    del masks
    #print start number
    way = ssn_drawer.generate_ascii_display(current_num, max_width=max_line_width)
    #calculate way
    def add_stick_to_digit(digit):
        add = adds[digit] << 1
        for i in range(7):
            add = add >> 1
            if add & 1:
                return digit, i
        return None#
    for rem_digit in range(len(current_num)): #iterates over digits in rems
        rem = rems[rem_digit] << 1
        for i in range(7): #iterates over segments in current digit
            rem = rem >> 1
            if rem & 1: #segment is turned on -> has to turn off
                rem_instr = (rem_digit, i)
                add_instr = add_stick_to_digit(rem_digit) #try turn different segment in same
digit on
                if add_instr == None: #no segment in same digit has to turn on -> search in
other digits
                    for add_digit in range(len(current_num)):
                        add_instr = add_stick_to_digit(add_digit)
                        if add_instr != None: break
                rems[rem_instr[0]] &= ~(1 << rem_instr[1]) #turn off segment in rems
                current_num[rem_instr[0]] &= ~(1 << rem_instr[1]) #turn off segment in current
num
                adds[add_instr[0]] &= ~(1 << add_instr[1]) #turn off segment in adds
                current_num[add_instr[0]] |= 1 << add_instr[1] #turn on segment in current num
                way += f"\n{ascii_box.generate_pixel([0, 2, 0, 2]) *
max_line_width}{ssn_drawer.generate_ascii_display(current_num, [add_instr], max_line_width)}"
    return way

# print(generate_way_ascii([int(x, 16) for x in list("73fe7d782")], [int(x, 16) for x in
list("fffffd782")]))
```

Quellcode 2: way.py