Aufgabe 3: HexMax

Bearbeiter/-in dieser Aufgabe:
Joshua Benning

26. August 2022

Inhaltsverzeichnis

Aufgabenstellung

Severin hat heute in der Schule das Hexadezimalsystem kennengelernt. Er verwendet nun kurze Stäbchen, um die sechzehn Ziffern darzustellen, wie in einer Siebensegmentanzeige. Für die Darstellung einer Ziffer stehen also sieben Positionen zur Verfügung; jede Position ist entweder durch ein Stäbchen belegt oder sie ist frei. Severin will nun das folgende, von der Lehrerin gestellte Rätsel lösen: Gegeben ist eine Hexadezimalzahl (kurz: Hex-Zahl) mit n Ziffern. Gesucht ist die größte Hex-Zahl mit der gleichen Anzahl an Ziffern, die aus der gegebenen Zahl durch eine zusätzlich gegebene Maximalzahl an Umlegungen erzeugt werden kann. „Umlegung“ bedeutet, ein Stäbchen von seiner bisherigen Position an eine andere, bislang freie Position zu bewegen. Eine

einzelne Umlegung muss noch keine gültige Hex-Zahl ergeben, aber das Ergebnis nach allen Umlegungen muss eine gültige Hex-Zahl in der Siebensegmentdarstellung sein. Zu keiner Zeit darf die Darstellung einer Ziffer komplett „geleert“ werden. Die gegebene Maximalzahl an Umlegungen muss nicht ausgeschöpft werden. Hier ein Beispiel: Gegeben sind die Zahl D24 und die Maximalzahl an Umlegungen 3. Die gesuchte Hex-Zahl ist EE4.

Schreibe ein Programm, das eine Hex-Zahl sowie die Maximalzahl m an Umlegungen einliest und die größte Hexadezimalzahl ermittelt, die mit höchstens m Umlegungen erzeugt werden kann. Das Programm soll nach jeder Umlegung den Zwischenstand, also die aktuelle Belegung der Positionen ausgeben. Wende dein Programm mindestens auf alle Beispiele an, die du auf den BWINF-Webseiten findest, und dokumentiere die Ergebnisse.

Begriffe

Im Folgenden wird häufig die Bewegung eines Streichholzes an eine andere Position beschrieben. Laut Aufgabenstellung ist die Bewegung eines Stäbchen von der Position A zur Position B eine “**Umlegung**”. Im folgenden wird dies allerdings als “**Zug**” bezeichnet, eine "**Umlegung**" beschreibt die Änderung der Belegung einer Position. Führt man also einen Zug (A → B) aus, entspricht das 2 Umlegungen da Position A von belegt zu frei und Position B von frei zu belegt geändert wird.

Dies ist notwendig, da innerhalb der Algorithmen nie Züge einzelner Stäbchen berechnet werden, sondern nur Umlegungen von / hin zu einer neutralen Position. Stäbchen werden also aus Ziffern entfernt und später erst, an anderer Stelle, wieder verwendet.

1 Lösungsidee

Um die größtmögliche Zahl mit n Umlegungen zu finden, habe ich 2 verschiedene Ansätze entwickelt. Lösungsidee 1 war mein ursprünglicher Ansatz der das Problem zwar löst, aber aufgrund seiner im Vergleich zu Lösungsidee 2 sehr hohen Laufzeit, nicht jedes Beispiel lösen konnte.

1.2 Lösungsidee 1: BackTrack

Um die größte Zahl mit maximal n Umlegungen zu finden, liegt ein Rücksetzungsverfahren (BackTrack-Algorithmus) nahe. Ausgehend von der größtmöglichen Zahl (Am Beispiel D24: FFF) versucht man dabei nur soweit vom Maximum abzuweichen wie es für eine valide Lösung notwendig ist.

1.3 Lösungsidee 2: Caching

Eine potentielle Lösung zu validieren lässt sich einfach rekursiv implementieren anhand den “benötigten” und “überbleibenden” Streichhölzern nach einer Umlegung. Basierend auf diesen Ergebnissen kann man potentielle Lösungen sehr früh ausschließen und sich eine erhebliche Menge Berechnungen einsparen.

2 Umsetzung (Allgemein)

Die Implementation erfolgt in Java. Zunächst wird die zu maximierende Zahl mit den dazugehörigen maximalen Umlegungen aus der entsprechenden .txt-Datei ausgelesen. Die verwendeten Hex-Ziffern werden dabei anhand eines Enums dargestellt. Die Einträge dieses Enums umfassen dabei den Wert der Hexziffer im dezimalen System (int), das Symbol der Ziffer im hexadezimalen System (str) und die zur Darstellung in der Siebensegementanzeige nötigen "Balken" (bool[]).

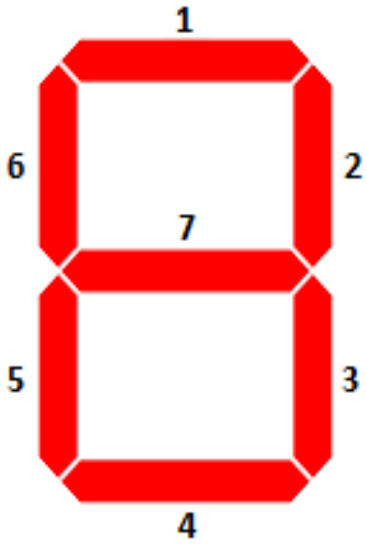

Abbildung 1: Siebensegementanzeige mit beschrifteten Segmenten

```
C(12,"C",new boolean[]{true,false,false,true,true,true,false}),
B(11,"B",new boolean[]{false,false,true,true,true,true,true}),
A(10,"A",new boolean[]{true,true,true,false,true,true,true}),
```

Programmauszug 1: Auszug aus dem Enum "Num"; Modellierung der Hex-Zahlen A, B und C

Die umgewandelten Ziffern werden dann in ein Array zusammengefasst welches die gegebene Zahl repräsentiert. Im folgenden wird die Anhzahl der maximalen Umlegungen als n bezeichnet, die Länge der gegebenen Ziffer als l, die gegebene Ziffer als G und die m-te Ziffern von G als G_m bezeichnet. m entspricht hierbei dem Index der Ziffer im Array, daher 0 <= m < l.

2.1 Umsetzung Lösunsidee 1: BackTrack

Wie bereits in der Beschreibung der Lösungsidee kurz erwähnt, handelt es sich bei dem Rücksetzungsverfahren um einen einfachen rekursiven Algorithmus. Ausgehend von der größtmöglichen Zahl B (B = {F, F, … F} wird versucht durch die betragsmäßig kleinste Änderung die Zahl B hin zu einer validen Lösung zu ändern.

Am in der Aufgabenstellung erwähnten Beispiel "D24 mit 3 Umlegungen" würde man hier ausgehend von der Zahl FFF alle Optionen für die dritte Stelle überprüfen. Lässt sich ausgehend von FF? keine valide Lösung finden, verringert man die nächste Stelle und überprüft alle 16

Optionen ausgehend von FE?. Man verringert also stets die letzte Ziffer, sollte dies nicht möglich sein greift man auf die vorherige zurück. Für das Beispiel "D24 mit 3 Umlegungen" ergibt sich so als Lösung EE4.

2.1.1 Pseudocode

```
1     int l < Länge der Ausgangszahl
2     Num[l] B < Ergebnis, Liste mit l-mal dem Element F
3     function addDigit :
4           if len(B) == l and B ist valide then
5                 return < valide Lösung wurde gefunden
6           end if
7           if letzte Ziffer von B ist 0 and B ist nicht valide then
8                 entferne die letzte Ziffer von B
9                 verringere die letzte Ziffer von B um 1
10                füge B ein F hinten an
11                addDigit()
12                return
13          end if
14          verringere die letzte Ziffer von B um 1
15          addDigit()
16    end function
```

2.1.2 Laufzeit

Die Laufzeit dieses Ansatzes ist sehr groß und hängt von der Größe der größten darstellbaren Zahl ab. Desto kleiner die Zahl, desto größer die Laufzeit da vom Maximum ausgehend jede Möglichkeit bis hin zur Lösung überprüft wird. Für den Extremfall, dass die gesuchte Zahl 0 wäre (auch wenn nach Aufgabenstellung nicht möglich) entspricht die Laufzeit der eines BruteForce-Algorithmuses (16 Optionen pro Stelle, l Stellen). Die Laufzeit beträgt daher bis zu $\mathbf{(16n)^l}$

2.2 Umsetzung Lösungsidee 2: Caching

Lösungsidee 2 entstand aus dem Versuch das Laufzeitproblem des ersten Ansatzes zu lösen. Die extrem hohe Laufzeit des Rücksetzungsverfahren resultiert daraus, das von der größtmöglichen Zahl der selben Länge bis hin zur gesuchten Zahl jede Zahl überprüft wird. Um dies zu verhindern, habe ich versucht einzelne “Ausgangssituationen” auszuschließen ohne alle Zahlen zu überprüfen.

Dies ist möglich, wenn man sich für jede Stelle i und verfügbare Stäbchen r die mindestens nötigen Umlegungen zwischenspeichert. So kann man zum Beispiel wenn man nach zwei Umlegungen an

den Stellen 0 und 1 noch 4 Stäbchen und 2 Umlegungen zur Verfügung hat, überprüfen ob ausgehend von den bisherigen Umlegungen überhaupt noch eine Lösung möglich ist.

Berechnung der mindestens nötigen Umlegungen

Pseudocode

MIT:

Φ = Die Menge aller Hex-Ziffern in betragsmäßig absteigender Reihenfolge
***Num.remaining(num2)** = Die Anzahl an verbleibenden Stäbchen wenn eine Num in eine andere umgewandelt wird (Kann negativ sein, wenn Stäbchen "benötigt" werden)*

```
1     Int l // Länge der gegebenen Ziffer
2     Cache (int, int) → int C // Cache; Key bestehend aus 2 Integern,
                                   Value ein Integer
3     function midestNötigeMoves(int digit, int barsAvailable) :
4           Integer cacheValue = C.get(digit, barsAvailable)
5           if cacheValue != null then
6                 return cacheValue
7           end if
8           if digit == l and barsAvailable != 0 then
9                 return MAX_INTEGER_VALUE // invalide Lösung trotz 10
                                                 vollständiger Länge
10          end if
11          if digit == l and barsAvailable == 0 then
12                return 0 < valide Lösung vollständiger Länge
13          end if
14          Integer result = MAX_INTEGER_VALUE
15          for n in Φ do
16                berechne die nötigen Umlegungen um die Stelle digit zur
                  Ziffer n zu ändern
17                berechne wieviele Züge zu einer Lösung nötig sind
                  (Rekursion)
18                if benötigteZüge < result then
19                      result = benötigteZüge
20                end if
21          end for
22    end function
```

Ermittlung der Lösung ohne Rücksetzungsverfahren

Basierend auf diesem Cache lässt sich zwar einerseits das BackTracking-Verfahren optimieren, andererseits kann aber deutlich schneller mit einem Greedy-Algorithmus das Ergebnis ermitteln. Wie bereits beim Rücksetzungsverfahren, versucht man auch hier von oben herab die erste valide Zahl zu finden.

```
public void generateResult() {
    long m = 0; // benötigte Züge
    int b = 0; // verfgbare Stäbchen
    Num[] result = new Num[num.length];
    for (int i = 0; i < num.length; i++) {
        Num current = this.num[i];
        for (Num potentialNewNum : current.getAllBiggerOnes()) {
            long newM = m + current.posDiff(potentialNewNum);
            int newB = b + current.remaining(potentialNewNum);
            long integer = this.necessaryMovesToNextValidNum(i, newB);
            if (newM + integer <= 2 * n) {
                result[i] = potentialNewNum;
                m = newM;
                b = newB;
                break;
            }
        }
    }
}
```

Programmauszug 2: Greedy-Algorithmus zur Bestimmung der größtmöglichen Zahl

Dabei wird der Signifikanz entsprechend von links nach rechts die jeweils größtmögliche Ziffer in das Num-Array result übertragen. Bedingung hierfür ist, dass die Summe der bisher verwendeten Umlegungen ("m"), der für den Zug benötigten Umlegungen ("current.posDif(...)") und der mindestens nötigen Umlegungen zur Vollendung der Lösung ("moves") kleiner als das Maximum ("2 * n") ist.

Beispiele

Die Ergebnisse waren, wenn beide Algorthmen eins zurückgeben, identisch. Ab Beispiel 3 lieferte allerdings das Rücksetzungsverfahren auch nach 8 Stunden Laufzeit kein Ergebnis. Zu den Beispielen wird deshalb ein Ergebnis und die Laufzeit der beiden Lösungsansätze angegeben.

Beispiel 0

```
Ausgangszahl:D24

Züge:3

Ergebnis: EE4

Backtrack: 0.004s

Caching: 0.0362409s
```

Beispiel 1

Ausgangszahl:509C431B55

Züge:8

Ergebnis: FFFEA97B55

Backtrack: 0.037s

Caching: 0.0691755s

Beispiel 2

Ausgangszahl:632B29B38F11849015A3BCAEE2CDA0BD496919F8

Züge:37

Ergebnis: FFFFFFFFFFFFFFFFD9A9BEAEE8EDA8BDA989D9F8

Backtrack: 16.627s

Caching: 0.2746378s

Beispiel 3

Ausgangszahl:0E9F1DB46B1E2C081B059EAF198FD491F477CE1CD37EBFB65F8D76505575
7C6F4796BB8B3DF7FCAC606DD0627D6B48C17C09

Züge:121

Ergebnis:FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
AA98BB8B9DFAFEAE888DD888AD8BA8EA8888

Backtrack: Kein Ergebnis nach 3h

Caching: 0.4875622s

Beispiel 4

Ausgangszahl:1A02B6B50D7489D7708A678593036FA265F2925B21C28B4724DD822038E3
B4804192322F230AB7AF7BDA0A61BA7D4AD8F888

Züge:87

Ergebnis:FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEB8DE88BAA8ADD888898E9BA88
AD98988F898AB7AF7BDA8A61BA7D4AD8F888

Backtrack: Kein Ergebnis nach 3h

Caching: 0.4651972s

Beispiel 5

```
Ausgangszahl:EF50AA77ECAD25F5E11A307B713EAAEC55215E7E640FD263FA529BBB48DC
8FAFE14D5B02EBF792B5CCBBE9FA1330B867E330A6412870DD2BA6ED0DBCAE553115C9A31
FF350C5DF993824886DB5111A83E773F23AD7FA81A845C11E22C4C45005D192ADE68AA9AA
57406EB0E7C9CA13AD03888F6ABEDF1475FE9832C66BFDC28964B7022BDD969E5533EA4F2
E4EABA75B5DC11972824896786BD1E4A7A7748FDF1452A5079E0F9E6005F040594185EA03
B5A869B109A283797AB31394941BFE4D38392AD12186FF6D233585D8C820F197FBA9F6F06
3A0877A912CCBDCB14BEECBAEC0ED061CFF60BD517B6879B72B9EFE977A9D3259632C718F
BF45156A16576AA7F9A4FAD40AD8BC87EC569F9C1364A63B1623A5AD559AAF6252052782B
F9A46104E443A3932D25AAE8F8C59F10875FAD3CBD885CE68665F2C826B1E1735EE2FDF0A
1965149DF353EE0BE81F3EC133922EF43EBC09EF755FBD740C8E4D024B033F0E8F3449C94
102902E143433262CDA1925A2B7FD01BEF26CD51A1FC22EDD49623EE9DEB14C138A7A6C47
B677F033BDEB849738C3AE5935A2F54B99237912F2958FDFB82217C175448AA8230FDCB3B
3869824A826635B538D47D847D8479A88F350E24B31787DFD60DE5E260B265829E036BE34
0FFC0D8C05555E75092226E7D54DEB42E1BB2CA9661A882FB718E7AA53F1E606

Züge:1369

Ergebnis:FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFF88EFA9EBE89EFA99FBDAA8E8EAD88AB899F8E8F9AA9E9AD889
88EDA9A99888EDAD989A8BAFD8A8888888888888888888888888888888888888888888888
8888888888888888888888888888888888888888888888888888888888888888888888888
8888888888888888888888888888888888888888888888888888888888888888888888888
888888888888888888888888888888888888888888888888888888888888

Backtrack: Kein Ergebnis nach 3h

Caching: 3.2876436s
```

Quellcode

Berechnung der mindestens nötigen Umlegungen

```
/**
 * @param digit Stelle an der eine valide Zahl gebildet werden soll
 * @param barsAvailable Momentane "Anzahl" an verfügbaren Stäbchen, kann negativ sein, wenn für vorherige Umlegungen noch Stäbchen benötigt werden
 * @return Die mindestens nötigen Umlegungen um an einer bestimmten Stelle mit einer bestimmten Anzahl verfügbarer Stäbchen, eine valide Num zu bilden
 */
private long necessaryMovesToNextValidNum(int digit, int barsAvailable) {
    Integer key = Objects.hash(digit, barsAvailable);
    Long cacheValue = this.fCache.get(key);
    if (cacheValue != null) return cacheValue;
    if (digit == this.num.length && barsAvailable != 0)
        return Integer.MAX_VALUE; // Length-wise complete solution, but invalid
    if (digit == this.num.length)
        return 0;               // Length-wise complete AND valid solution

    Num currentNum = this.num[digit];
    long result = Integer.MAX_VALUE;
    for (Num num : Num.values()) {
        int barsAvailable1 = barsAvailable + currentNum.remaining(num);
        long i = necessaryMovesToNextValidNum(digit + 1, barsAvailable1) + currentNum.posDiff(num);
        if (result == 0) return result;
        if (i < result) result = i;
    }
    this.fCache.put(Objects.hash(digit, barsAvailable), result);
    return result;
}
```

Num-Enum

```
public enum Num {

    F(15, "F", new boolean[]{true, false, false, false, true, true, true}),
    E(14, "E", new boolean[]{true, false, false, true, true, true, true}),
    D(13, "D", new boolean[]{false, true, true, true, true, false, true}),
    C(12, "C", new boolean[]{true, false, false, true, true, true, false}),
    B(11, "B", new boolean[]{false, false, true, true, true, true, true}),
    A(10, "A", new boolean[]{true, true, true, false, true, true, true}),
    NINE(9, "9", new boolean[]{true, true, true, true, false, true, true}),
    EIGHT(8, "8", new boolean[]{true, true, true, true, true, true, true}),
    SEVEN(7, "7", new boolean[]{true, true, true, false, false, false, false}),
    SIX(6, "6", new boolean[]{true, false, true, true, true, true, true}),
    FIVE(5, "5", new boolean[]{true, false, true, true, false, true, true}),
    FOUR(4, "4", new boolean[]{false, true, true, false, false, true, true}),
    THREE(3, "3", new boolean[]{true, true, true, true, false, false, true}),
    TWO(2, "2", new boolean[]{true, true, false, true, true, false, true}),
    ONE(1, "1", new boolean[]{false, true, true, false, false, false, false}),
    ZERO(0, "0", new boolean[]{true, true, true, true, true, true, false});
}
```

Backtrack-Ansatz

```
public void addDigit() {
    if (valid(nums)) {
        return;
    }
    if (nums[nums.length - 1] == Num.ZERO && !valid(nums)) {
        for (int length = numLength - 1; length > 0; length--) {
            if (nums[length] == Num.ZERO && nums[length - 1] == Num.ZERO) {
                nums[length - 1] = Num.F;
            }
            if (nums[length] == Num.ZERO) {
                nums[length] = Num.F;
                nums[length - 1] = nums[length - 1].getLower();
            }
        }
        addDigit();
        return;
    }
    nums[nums.length - 1] = nums[nums.length - 1].getLower();
    addDigit();
}

private boolean valid(Num[] potentialSolution) {
    int totalMoves = 0; // Benötigte Umlegungen
    int barsAvailable = 0; // Verfügbare Stäbchen

    for (int i = 0; i < numLength; i++) {
        // Einzelne Positionen "vergleichen" und insgesamt benötigte Züge zusammenrechnen
        totalMoves += originalNum[i].posDiff(potentialSolution[i]);
        barsAvailable += originalNum[i].remaining(potentialSolution[i]);
    }
    // Wenn die maxmalen Umlegungen nicht überschritten werden
    // und keine Stäbchen zu viel / zu wenig vorhanden sind, ist die Lösung valide
    return totalMoves <= 2 * maxMoves && barsAvailable == 0;
}
```

Quellen / Material

Aufgabestellung: https://bwinf.de/bundeswettbewerb/40/2/ [01.09.2022 ~ 18:52 Uhr]

Beispieldaten: *https://bwinf.de/bundeswettbewerb/40/2/* [01.09.2022 ~ 18:52 Uhr]