Aufgabe 5

Team-ID: 00112

Team-Name: LonelyFlo4TheWin

Bearbeitet von:
Florian Bange

22. November 2021

Inhaltsverzeichnis

1 Loesungsidee

Um dieses Problem loesen zu koennen muss man fuer jedes Gewicht von 10 bis 10.000, in 10er Schritten herrausfinden, ob es durch angegebene Gewichtsstuecke auf einer Markwaage messbar ist. Damit es das ist, muss es moeglich sein die gegeben Gewichtsstuecke auf einer Waage so anzuordnen, dass sich aus der Differenz der rechten und der linken Seite der Waage, das Gewicht bildet (sofern die Ware auf der linken Seite liegt). Man rechnet also die Summe der Gewichte der rechten Seite minus die Summe der Gewichte der rechten Seite. Diese Differenz muss das Gewicht der Ware ergeben. Denn dann ist die Gleichung

$$(linkesGewicht) + (GewichtDerWare) = (RechtesGewicht) \tag{1}$$

wahr und die Waage im Gleichgewicht (wenn die Ware das vorausgesetzte Gewicht wirklich hat).
Um dieses Problem zu loesen „errechne“ ich zuerst die Potenzmenge (die Menge aller Teilmengen) der Menge der gegebenen Gewichte und errechne zu jeder Teilmenge die Summe der Gewichte.

Um die Gleichung fuer ein bestimmtes Warengewicht zu finden tue ich Folgendes:

1. Ich gehe die Potenzmenge durch und schaue, ob das Warengewicht in der Potenzmenge als Summe enthalten ist. Ist dies der Fall ist die gefundene Teilmenge die Menge der Gewichte fuer die rechte Seite der Waage.

2. Ist das Warengewicht nicht als Summe in der Potentzmenge enthalten, gehe ich die Potenzmenge erneut durch, und errechne fuer jede Teilmenge die Differenz zwischen der Summe der Teilmenge und dem Gewicht. Wenn die Differenz ebenfalls als Summe in der Potenzmenge enthalten ist, uberpruefe ich zuletzt, ob in beiden Mengen zusammen kein Element haeufiger existiert als es durch die gegebenen Gewichte moeglich ist. Ist dem so, sind die beiden Teilmengen die Menge der Gewichte der rechten und linken Seite der Waage (in dieser Reihenfolge).

Fuer alle Warengewichte, die ich so nicht erreichen konnte, muss ich nun bestimmen, wie nah man mit den gegebenen Gewichten an das Warengewicht herankommt. Dazu tue ich Folgendes fuer alle noch nicht gefundenen Warengewichte:

1. Ich finde das naechst kleinere genau bestimmbare Warengewicht (kann wenigstens 0 sein).

2. Ich finde das naechst groeßere genau bestimmbare Warengewicht. Wenn es keines gibt, nehme ich das naechste Element der Potenzmenge zum Warengewicht.

3. Nun bestimme ich das genau bestimmbaren Warengewicht der beiden zuvor erwaehnten, welches naeher am gesuchten Warengewicht liegt.

4. Anschließend gehe ich vom gesuchten Warengewicht in beide Richtungen gleichzeitig weg, bis das kleinere genau bestimmbaren Warengewicht erreicht wird. Fuer jedes dabei erzielte Gewicht ueberpruefe ich, mit Hilfe der zuvor geschilderten Methode, ob das Gewicht darstellbar ist. Ist es darstellbar, wurde ein naeheres genau bestimmbares Warengewicht gefunden und dies ist somit das naechste genau bestimmbare Warengewicht zum gesuchten Warengewicht.

5. Wurde kein darstellbares Gewicht zwischen (gesuchtes Warengewicht - Kleinere Differenz) und (gesuchtes Warengewicht + Kleinere Differenz) gefunden, ist das naehere genau bestimmbare Warengewicht, dass der beiden zuvor bestimmten, das naeher am gesuchten Warengewicht ist.

2 Umsetzung

Die Loesungsidee habe ich in Java 8 implementiert.
Am Ende der nun beschriebenen Umsetzung kann man fuer jedes Gewicht (von 10 bis 10.000 in 10er-Schritten) eine Direkte oder Indirekte Loesung aus einem Objekt der „GewichteChecker“ Klasse bekommen. Genaueres dazu spaeter.

2.1 Initialisierung

In der Initialisierungs Methode wird die eine Datei ausgelesen, die vorher angegeben werden kann. Aus dieser wird aus jeder Zeile das Gewicht als Long und die Anzahl des Gewichts (Integer) entnommen. Diese Werte werden in eine HashMap mit dem Namen „weights“ gesetzt, welche von den Gewichten (Long) auf eine Anzahl (Integer) zeigt.

2.2 Die „addables“-HashMap

Nun erstelle ich eine HashMap mit der Potenzmenge, „addables“. Bei dieser zeigt die Summe (Long) auf die Liste (List<Long>), welche diese Summe ergibt. Um diese HashMap zu bekommen, benutze ich die selbstgeschriebene „CombinationFinder“ - Klasse. Diese Klasse bekommt in den Konstruktor die Liste aller Gewichte (mit doppelten Gewichten) uebergeben und hat die Methode „getPossibleCombinations“, dessen Rueckgabewert ich nun „addables“ zuweise.
Die Liste aller Gewichte erstelle ich zuvor. Die Laenge dieser Liste weise ich dem Field „maxWeights“ zu.

Zusaetzlich wird eine Liste „addablesKeys“ mit den Keys (Schluesseln) der „addables“-HashMap initialisiert. Diese Liste wird nun mit der von Java implementierten Methode Collections:sort aufsteigend sortiert. Außerdem wird der groeßte Wert aus „addablesKeys“, bzw. die Summe aller Gewichte als „biggestAddable“ (long, field) gespeichert.

2.2.1 Die „CombinationFinder“ Klasse

Diese Klasse ist einzig und allein dazu da, eine HashMap erschaffen, welche je von einer Summe auf einer Liste zeigt, die diese Summe erzeugt. Dazu bekommt die „CombinationFinder“ Klasse eine Liste an Long uebergeben. Anschließend erschafft sie alle Teilmengen dieser Liste und traegt die Summe jeder Teilmenge mit der Liste in die HashMap ein. Die HashMap heißt „possibleCombinations“ und kann mit der Methode „getPossibleCombinations“ erhalten werden.
Dies Funktioniert, wie folgt:
Im Konstruktor der Klasse wird zunaechst die Liste aller Gewichte als field „input“ und die HashMap „possibleCombinations“ mit einer lehren HashMap initialisiert. Nun wird die Methode „setPossibleAdditions“ aufgerufen. Diese Methode iteriert von 0 bis zu (Groeße der input-Liste - 1). In jedem Durchgang wird eine einelementige List („list“) mit dem aktuellen Wert in einer Liste initialisiert und es werden die die Methoden „add“ und „getAll“ mit der List als Uebergabeparameter aufgerufen.
Die „add“-Methode durchlaeuft folgende Schritte:

1. Eine uebergebene Liste an Indexen wird in eine Liste aus Elementen der „input“-Liste verwandelt, indem ich jeden Index der uebergebenen Liste durch das Element der „input“-Liste an der Stelle des Index ersetzte. Diese heißt „actualList“.

2. Die Summe dieser Liste wird durch einen Java LongStream ermittelt. Die Summe wird „sum“ genannt und hat den Datentyp Long.

3. Wenn diese Summe noch nicht als key in der „possibleCombinations“-HashMap enthalten ist, wird die Summe („sum“ mit dem value „actualList“) in „possibleCombinations“ „gepackt“.

In der rekursiven Methode „getAll“ wird Folgendes getan:

1. Das letzte Element der uebergebenen Liste wird als „lastElement“ initialisiert.

2. Wenn („lastElement“ + 1) groeßer/gleich der groeße der „input“-Liste ist, wird die Methode beendet

3. Anschließend wird von („lastElement“ + 1) bis zu (Groeße der input-List - 1) iteriert. Dabei wird jedes Mal eine Copy der uebergebenen List erstell. Zu dieser wird der momentane Wert der Schleifenvariable hinzugefuegt. Mit dieser Liste wird nun „add“ und „getAll“ aufgerufen.

2.3 „Direkte“ Loesungen

Eine Direkte Loesung ist die Loesung aus einer Linken und einer Rechten Liste an Gewichten bzw. longs, die wenn man die Summe der linken Liste von der Summe der rechten Liste abzieht, einen bestimmtes Warengewicht erreichen. Diese direkten Loesungen werden in meinem Programm als „ListResult“ gespeichert. Ein ListResult enthaelt zwei Listen, bzw. die linke und die rechte Liste.

Um alle Direkten Loesungen zu finden wird die Methode „getDirectResults“ benutzt. Diese Methode soll fuer alle Warengewicht zwischen 10 und (inklusive) 10.000 in 10er-Schritten eine Loesung finden, welche das Warengewicht darstell.
In der Methode wird fuer jedes Warengewicht zwischen 10 und (inklusive) 10.000 in 10er-Schritten das „ListResult“ „result“ mit dem Rueckgabewert der Methode „getDirectResult“ initialisiert. Diese bekommt das Warengewicht als Uebergabeparamether.
Wenn der Rueckgabewert nicht „null“ ist, wird das result mit dem Warengewicht als key in die „directResults“-HashMap getan.

2.3.1 Die „getDirectResult“-Methode

```
1 private ListResult getDirectResult(long weight)
```

Die „getDirectResult“ Methode bekommt als Uebergabeparamether ein Warengewicht und soll ein ListResult zurueckgeben, welches das Warengewicht formt.
Als erstes wird ueberprueft, ob die HashMap „addables“ (also die HashMap mit allen, durch Addition der „weights“, erreichbaren Werten) den key weight enthaelt. Wenn dies der Fall ist, wird eine ListResult mit einer leeren Liste als linke Liste und dem value von „weight“ aus „addables“ als rechte Liste zurueckgegeben.
Wenn „addables“ weight nicht als key enthaelt, wird der Rueckgabewert der Methode „getListResult“ mit dem Uebergabewert weight zurueckgegeben.

2.3.2 Die „getListResult“-Methode

```
1 private ListResult getListResult(long weight)
```

Die Methode „getListResult“ gibt entweder ein ListResult oder null zurueck.
Dafuer geht sie durch die Liste „addablesKeys“ (welche alle Summen der Potenzmenge aller angegebenen Gewichten, mit doppelten Gewichten, enthaelt) und tut fuer jedes Element der „addablesKeys“ Folgendes (das momentane Elements aus „addablesKeys“ wird dabei „current“ genannt):

1. Bekomme die Differenz („dif“) zwischen current und dem Gewicht „weight“, indem weight von current angezogen wird.
2. Wenn „dif“ negativ ist, wird dieser Durchgang uebersprungen.
3. Wenn dif ebenfalls Element von „addablesKeys“ ist, wird die Liste „left“ mit dem value von „dif“ aus „addables“ und die Liste „right“ mit dem value von „current“ aus „addables“ initialisiert.
4. Nun wird nur noch ueberprueft, ob die beiden Listen („left“ und „right“) angegebene Gewichte haeufiger beinhalten als moeglich (also, ob ein Gewicht, in beiden Listen zusammen, zu oft benutzt wird). Wenn dies nich der Fall ist, wird ein ListResult mit „left“ als Linke und „right“ als rechte List zurueckgegeben. Um zu ueberpruefen, ob kein Gewicht zu oft benutzt wird, benutze ich die Methode „hasNoDoubles“.

Sollte in der Iteration kein passendes ListResult gefunden werden, wird null zurueckgegebeben.

2.3.3 Die „hasNoDoubles“-Methode

```
1 private boolean hasNoDoubles(List<Long> left, List<Long> right)
```

Die „hasNoDoubles“-Methode gibt als Wahrheitswert zurueck, ob zwei uebergebene Listen („left“ und „right“) kein Gewicht zu haeufig enthalten.
Dazu wird zunaechst ueberprueft, ob die Summe der Laenge beider uebergebener Listen groeßer ist, als „maxWeights“ (die maximal moegliche). Ist dies der Fall, wird false zurueckgegeben. False wird außerdem zurueckgegeben, wenn die Liste „doubles“, welche ListResults speichert, eine ListResult enthaelt, welche die Listen „left“ und „right“ als linke und rechte Liste enthaelt.

Wenn die Methode „noDoubles“ true zurueckgibt, wird true zurueckgegeben. Die Methode „noDoubles“ zaehlt die Anzahl alle Werte von den beiden Listen in einer HashMap und vergleicht anschließend fuer jeden key dieser HashMap, ob der Wert des keys (also die anzahl, wie oft dieser key in beiden Listen vorgekommen ist) hoeher ist, als der Wert der HashMap „weights“ (welche die am Anfang gespeicherten „wirklichen“ Werte fuer jedes Gewicht speichert). Wenn das der Fall ist, wird false zurueckgegeben. Nachdem alle in beiden Listen vorkommenen Werte durchgegangen wurden, wird true zurueckgegeben.
Nun ist in den beiden Listen eindeutig mindestens ein Element zu haeufig vorgekommen. Somit wird nun zur Liste „doubles“ ein ListResult der uebergebenen Listen hinzugefuegt und anschließend false zurueckgegeben.

2.4 „Indirekte“ Loesungen

Eine Indrekte Loesung besteht aus einem Warengewicht, das so nah, wie moeglich an ein anderes herankommen soll, und den zwei zugehoerigen Listen (linke und rechte Liste), die das Warengewicht formen. In meinem Programm wird die Indirekte Loesung durch ein „IndirectResult“ dargestellt. Diese Klasse erbt von „ListResult“ und speichert zusaetzlich das „n“ Warengewicht („closestWeight“).

In der Methode „getClosestsResults“ wird fuer alle Warengewichte, zwischen 10 und (inklusive) 10.000 in 10er-Schritten, welche noch keine „direkte“ Loesungen haben, eine „Indirekte“ Loesung gesucht.
Fuer jedes Warengewicht zwischen 10 und (inklusive) 10.000 in 10er-Schritten wird das „IndirectResult“ gefunden, indem Folgendes getan wird (dabei wird das aktuelle Warengewicht „weight“ genannt):

1. Wenn die „directResults“-HashMap „weight“ als key enthaelt, wird dieser Durchgang uebersprungen. Das bedeutet, dass nur Warengewichte analysiert werden, die noch keine direkte Loesung haben.

2. Nun wird ein „IndirectResult“ „result“ durch den Rueckgabewert der „getClosestResult“-Methode mit dem Uebergabewert „weight“ initialisiert.

3. Anschließend wird das „naechste“ Gewicht („closestWeight“ aus result entnommen und als „closestWeight“ mit dem Datentyp long gespeichert.

4. Jetzt wird ueberprueft, ob „cache“ (dies ist eine HashMap, welche bereits erkannte „naechste“ Gewichte speichert, indem sie von einem Long, also dem Gewicht, auf ein ListResult zeigt), „closestWeight“ als key enthaelt. Wenn dies nicht der Fall ist, wird „closestWeight“ mit dem value „result“ in „cache“ getan. (Dies ist moeglich, da IndirectResult von ListResult erbt.)

5. Zuletzt wird „weight“ mit dem value „result“ in die HashMap „closestResults“ getan. „closestResults“ zeigt von einem Long auf ein IndirectResult und enthaelt alle Warengewicht (als key), welche keine direkte Loesung haben. Diese zeigen jeweils auf ihr IndirectResult, also auf ihre naechste Loesung.

2.4.1 Die „getClosestResult“-Methode

```
1 private IndirectResult getClosestResult(long weight)
```

Die „getClosestResult“ Methode gibt die naechste Loesung fuer ein Warengewicht zurueck.
Dazu wird als erstes nach dem naechst kleineres Warengewicht gesucht, das in der „directResults“ HashMap als key exestiert. Dieses wird „closestBelow“ genannt. Dafuer wird in 10er-Schritten von (weight-10) bis 10 runtergezaehlt. In jedem Durchgang wird ueberprueft, ob „directResults“ das aktuelle Gewicht als key enthaelt. Ist dies der Fall wird closestBelow auf den Wert der Schleifenvariable gesetzt und die Schleife abgebrochen. Sollte in „directResults“ kein passendes Element gefunden werden, ist „closestBelow“ 0.

Als naechstes wird das naechst groeßere darstellbare Warengewicht gesucht. Also der naechst groeßere key aus „directResults“. Dieses wird „closestAhead“ genannt. Dazu wird fast das gleiche Verfahren benutzt, mit dem Unterschied, dass von (weight + 10) bis einschließlich 10.000 (ebenfalls in 10er-Schritten) gezaehlt wird. Aufgrund dessen, dass es moeglich ist, dass kein naechst groeßeres darstellbares Warengewicht exestiert, wird ueberprueft, ob „closestAhead“ seinen Wert veraendert hat. War dies nicht der Fall, wird „closestAhead“ auf den Rueckgabewert der Methode „getClosestBiggerWeight“ gesetzt. Diese gibt das naechste Element der „addablesKeys“ Liste zu weight zurueck. Ebenfalls wird in diesem Fall der boolean „check“ von false auf true gesetzt.

Nachdem „closestAhead“ nun sicher einen Wert hat, wird jeweils die Differenz zwischen „weight“ und „closestBelow“, bzw. „closestAhead“ als „difBelow“ und „difAhead“ initialisiert. Da weight groeßer sein kann als closestAhead, wird hier der absolute Wert von (closestAhead - weight) berechnet, fuer „difBelow“ wird (weight - closestBelow) berechnet.

Jetzt wird ueberprueft, ob closestAhead gleich „biggestAddable“ (der Summe aller vorgegebenen Gewichte) ist und difAhead kleiner/gleich difBelow ist. Ist dies der Fall bedeutet das, dass der groeßte erreichbare Wert (also die Summe aller Gewichte) am naechsten an weight dran ist. Ist dies der Fall, wird ein IndirectResult mit „closestAhead“ als naechste Loesung, einer leeren Liste als linke Liste und der Liste aller Gewichte zurueckgegeben.

Nun wird die kleinere Differenz der beiden als „smallerDif“ initialisiert. Dann gehe ich von 1 bis (exklusive) „smallerDif“ in 1er-Schritten. Die Schleifenvariable heißt „i“. Dabei tue ich Folgendes:

1. Ich berechne die Gewichte i Schritte links und rechts von „weight“. („w1“ und „w2“)
2. Ich ueberpruefe, ob „cache“ (die HashMap, welche bereits erkannte „naechste“ Gewichte speichert) „w1“ als key gespeichert hat. Ist dies der Fall wird ein IndirectResult mit „w1“ als naechste Loesung und den Listen des values von „w1“ aus „cache“ zurueckgegeben.
3. Ich ueberpruefe, ob „cache“ „w2“ als key gespeichert hat. Ist dies der Fall wird ein IndirectResult mit „w2“ als naechste Loesung und den Listen des values von „w2“ aus „cache“ zurueckgegeben.
4. Als naechstes wird die Methode „getIndirectResult“ einmal mit dem Uebergabewert „w1“ und einmal mit dem Uebergabewert „w2“ benutzt. Solfern das zurueckgegebene IndirectResult nicht null ist, wird dieses zurueckgegeben.

Wenn bis jetzt keine Loesung zurueckgegeben wurde, bedeutet das, dass keine moegliche Anordnung von Gewichten exestiert, welche naeher an „weight“ ist, als „closestAhead“ oder „closestBelow“.
Nach der Schleife wird „closest“ bestimmt. Dies ist die naechst moegliche Loesung (also „closestAhead“ oder „closestBelow“) mit der kleineren Differenz zu weight.
Danach wird das ListResult („result“) bestimmt, dass zu „closest“ gehoert:

1. Wenn „closest“ 0 ist, wird result zu einem ListResult mit zwei Leeren Listen.
2. Wenn der boolean „check“ false ist, wird result zu dem value von „closest“ der „directResults“ HashMap.
3. Wenn der boolean check true ist, wird zunaechst ueberprueft, ob „directResults“ den key „closest“ enthaelt. Ist dies der Fall, wird result zu dem value von „closest“ der „directResults“ HashMap. Sonst wird result zu dem value von „closest“ der „addables“ HashMap.

Schließlich wird ein „IndirectResult“ mit „closest“ als naechtes Gewicht und den Listen des results („result“) als Linke und Rechte Liste zurueckgegeben.

2.4.2 Die „getIndirectResult“-Methode

```
1 private IndirectResult getIndirectResult(long w)
```

Diese Methode gibt ein „IndirectResult“ zurueck, indem sie zunaechst ueberprueft, ob die Liste „notPossibleWeights“ „w“ nicht enthaelt. Die Liste „notPossibleWeights“ speichert, alle Werte, die nicht durch die vorgegebenen Gewicht erreicht werden koennen (wann sie befuehllt wird, beschreibe ich gleich). Enthaelt notPossibleWeights w also nicht, wird die Methode „getDirectResult“ (siehe 2.3.1 Die „getDirectResult“-Methode) mit dem Uebergabewert „w“ benutzt und ihr Rueckgabewert als „result“ gespeichert. Ist „result“ nicht null, wird ein IndirectResult mit w als naechstem Gewicht, und der linken und rechten Liste aus „result“ zurueckgegeben. Wenn „result“ null ist, wird zur Liste „notPossibleWeights“ „w“ hinzugefuegt. Nun wird null zurueckgegeben. Dies passiert also, wenn „notPossibleWeights“ w enthaelt, oder „getDirectResult“ mit dem Uebergabewert „w“ null zurueckgibt.

3 Beispiele

Im Folgenden kann ich nicht komplett auf die Ergebnisse jeder Datei eingehen. Deshalb finden sie im Ordner Aufgabe5/Loesungen genauere Ergebnisse. Jedoch kann ich immer auf die Anzahl der genauen und der „naechsten“ Ergebnisse eingehen. Naechste Ergebnisse sind die, die das Warengewicht nicht genau treffen, sondern nur so nah, wie moeglich an das Warengewicht kommen.

3.1 Aufgabe des Aufgabenblatts/gewichtsstuecke0.txt

Die Loesung meines Programmes fuer diesen Schiebeparkplatz hat 993 Genaue Ergebnisse und 7 „Naechste Ergebnisse. Dies ergibt Sinn, da die Summe (und somit die hoechste kreierbare Zahl) 9930 ist und somit keine hoehere Zahl gebildet werden.
Die Loesung meines Programms fuer das Beispiel des Aufgabenblattes (440) ist dieselbe, wie die des Aufgabenblattes (links: 50g, 10g; rechts: 500g).

3.2 gewichtsstuecke1.txt

Genaue Ergebnisse: 300 / 1000 Naechste Ergebnisse: 700 / 1000

3.3 gewichtsstuecke2.txt

Genaue Ergebnisse: 1000 / 1000 Naechste Ergebnisse: 0 / 1000

3.4 gewichtsstuecke3.txt

Genaue Ergebnisse: 1000 / 1000 Naechste Ergebnisse: 0 / 1000

3.5 gewichtsstuecke4.txt

Genaue Ergebnisse: 610 / 1000 Naechste Ergebnisse: 390 / 1000

3.6 gewichtsstuecke5.txt

Genaue Ergebnisse: 84 / 1000 Naechste Ergebnisse: 916 / 1000

3.7 Eigenes Beispiel 1

Dieses Beispiel erhaelt als Input keine Gewichte
Mein Programm gibt fuer jedes Warengewicht das „naechtes Ergebnis“ 0g zurueck.

3.8 Eigenes Beispiel 2

Dieses Beispiel enthaelt als Input ausschließlich einmal das Gewicht 300.
Mein Programm gibt fuer diesen Input fuer die Warengewicht 10 bis 150 das „naechtes Ergebnis“ 0 zurueck. Ab 160 haben alle Warengewichte außer 300 das „naechtes Ergebnis“ 300. Das Ergebnis fuer 300 sieht wie folgt aus: „300g: links: /; rechts: 300g“.
Diese Ausgabe ist logisch, da

1. 300 offensichtlich als einziges darstellbar ist.
2. 0 naeher oder gleich nah an allen Zahlen von 0 bis 150 liegt, als 300.
3. Alle Warengewicht ab (einschließlich) 160 (bis 10.000) Naeher an 300 als 0 liegen.

3.9 Eigenes Beispiel 3

Dieses Beispiel enthaelt als Input die Zahlen 1337 und 337 je einmal.
Mein Programm gibt fuer diesen Input folgende Ergebnisse aus:

1. Fuer die Warengewicht 10 bis 160: „naechtes Ergebnis: 0g: links: /; rechts: /“
2. Fuer die Warengewicht 170 bis 660: „naechtes Ergebnis: 337g: links: /; rechts: 337g“
3. Fuer die Warengewicht 670 bis 990: „naechtes Ergebnis: 1000g: links: 337g; rechts: 1337g“
4. Fuer das Warengewicht 1000: „links: 337g; rechts: 1337g“
5. Fuer die Warengewicht 1010 bis 1160: „naechtes Ergebnis: 1000g: links: 337g; rechts: 1337g“
6. Fuer die Warengewicht 1170 bis 1510: „naechtes Ergebnis: 1337g: links: /; rechts: 1337g“
7. Fuer die Warengewicht 1520 bis 10.000: „naechtes Ergebnis: 1674g: links: /; rechts: 337g, 1337g “

Dies ergibt ebenfalls Sinn:
Alle positiven Zahlen, die man erhalten kann, sind 0, 337, 1000 (1337-337), 1337 und 1674 (337+1337).
Die Warengewichte an welchen sich das Ergebnis aendert sind stets naeher an ihrem Ergebnis als an dem vorherigen:

1. 170 ist naeher an 337 (Differenz 167) als an 0 (Differenz 170).
2. 670 ist naeher an 1000 (Differenz 330) als an 337 (Differenz 333).
3. 1170 ist naeher an 1337 (Differenz 167) als an 1000 (Differenz 170).
4. 1520 ist naeher an 1674 (Differenz 154) als an 1337 (Differenz 183).

4 Quellcode

4.1 Die „addables“-HashMap

4.1.1 Die „setAddables“-Methode

```
    // Sets addables to List of all possible Combinations of weights (weight to List)
    private void setAddables() {
        // Create List of all weights (including doubles)
        List<Long> totalWeights = new ArrayList<>();
        for (long key : this.weights.keySet())
            for (int i = 0; i < this.weights.get(key); i++)
                totalWeights.add(key);

        // Set maxWeights to size of created list
        this.maxWeights = totalWeights.size();

        // Set addables
        CombinationFinder combinationFinder = new CombinationFinder(totalWeights);
        this.addables = combinationFinder.getPossibleCombinations();
    }
```

4.1.2 Die „CombinationFinder“ Klasse

```
    private final long size;
    private final List<Long> input;
    private final Map<Long, List<Long>> possibleCombinations;

    public CombinationFinder(List<Long> input) {
        this.size = input.size();
        this.input = input;
        this.possibleCombinations = new HashMap<>();

```

```
        this.setPossibleAdditions();
    }

    // Start recursively adding every number
    private void setPossibleAdditions() {
        for (int i = 0; i < this.size; i++) {
            List<Integer> list = this.copy(Collections.singletonList(i));
            this.add(list);
            this.getAll(list);
        }
    }

    // Recursively adds possibleCombinations
    private void getAll(List<Integer> list) {
        int lastElement = list.get(list.size() - 1);
        if (lastElement + 1 >= this.size) return;

        for (int i = lastElement + 1; i < this.size; i++) {
            List<Integer> list1 = this.copy(list);
            list1.add(i);
            this.add(list1);
            this.getAll(list1);
        }
    }

    // Adds list of indexes in possibleCombinations
    private void add(List<Integer> list) {
        // Get actual list from indexes
        List<Long> actualList = list.stream().map(this.input::get).collect(Collectors.toList());
        // Get sum of actualList
        long sum = this.sum(actualList);

        // If sum is not a key in possibleCombinations add sum and list to it
        if (!this.possibleCombinations.containsKey(sum))
            this.possibleCombinations.put(sum, actualList);
    }
```

4.2 Die direkten Results

4.2.1 Die „getDirectResults“-Methode

```
    // Adds ListResult for all weights (10, 20, ..., 10_000) using methode
    // "getDirectResult". (ListResult: Left List and Right List) If there is no
    // result nothing gets added
    private void getDirectResults() {
        for (long weight = 10; weight <= 10_000; weight += 10) {
            ListResult result = this.getDirectResult(weight);

            if (result != null)
                this.directResults.put(weight, result);
        }
    }
```

4.2.2 Die „getDirectResult“-Methode

```
    // Return the "ListResult" of a weight:
    // (ListResult = Left List and Right List which form the weight)
    private ListResult getDirectResult(long weight) {
        // Check if you can form the weight from the input-weights
        if (this.isAddable(weight)) {
            return new ListResult(new ArrayList<>(), this.addables.get(weight));
        } else {
            // Gets ListResult (might be null)
            return this.getListResult(weight);
        }
```

```
    }
```

4.2.3 Die „getListResult“-Methode

```
    // Return ListResult for weight if existing
    private ListResult getListResult(long weight) {
        // Goes through all possible addable numbers
        for (long current : this.addablesKeys) {
            // Get difference from weight to current
            long dif = current - weight;

            if (dif < 0) continue; // Prevent negative differences

            // Check if the difference is also addable
            if (this.isAddable(dif)) {
                List<Long> left = this.addables.get(dif), right = this.addables.get(current);

                // Check if there are one or more weights in both lists, which are used to often
                if (this.hasNoDoubles(left, right))
                    return new ListResult(left, right);
            }
        }

        return null;
    }
```

4.2.4 Die „hasNoDoubles“ und die „noDoubles“ -Methoden

```
    // Returns if there are weights more often than possible
    private boolean hasNoDoubles(List<Long> left, List<Long> right) {
        if (left.size() + right.size() > this.maxWeights) return false;

        // Check if left and right are already marked as double in "doubles"
        if (this.doubles.stream().anyMatch(result -> result.getLeft().equals(left)
                && result.getRight().equals(right))) return false;

        // Actually checking if there are elements too often, return true if not so
        if (this.noDoubles(left, right)) return true;

        // Else putting lists in "doubles" and return false
        this.doubles.add(new ListResult(left, right));
        return false;
    }

    // Returns if there are weights more often than possible
    private boolean noDoubles(List<Long> list, List<Long> list1) {
        Map<Long, Integer> existingTimes = new HashMap<>();
        list.forEach(i -> existingTimes.put(i, existingTimes.getOrDefault(i, 0) + 1));
        list1.forEach(i -> existingTimes.put(i, existingTimes.getOrDefault(i, 0) + 1));

        for (long key : existingTimes.keySet()) {
            long actualTimes = this.getAmount(key), realTimes = existingTimes.get(key);
            if (realTimes > actualTimes) return false;
        }

        return true;
    }
```

4.3 Die indirekten Results

4.3.1 Die „getClosestsResults“-Methode

```
    // Adds the closest result for all weights (10, 20, ..., 10_000)
    // if there is no "best"/normal result in "directResults" using the
    // methode "getClosestResult".
    // (IndirectResult: Closest Weight, Left List, Right List)
    private void getClosestsResults() {
        if (this.directResults.size() == 1000) return;

        for (long weight = 10L; weight <= 10_000; weight += 10) {
            // Skip if there already is a ListResult for current weight
            if (this.directResults.containsKey(weight)) continue;

            IndirectResult result = this.getClosestResult(weight);
            long closestWeight = result.getClosestWeight();

            // Put new result in cache
            if (!this.cache.containsKey(closestWeight))
                this.cache.put(closestWeight, result);

            this.closestResults.put(weight, result);
        }
    }
```

4.3.2 Die „getClosestResult“-Methode

```
    // Returns the closest result possible to weight
    private IndirectResult getClosestResult(long weight) {
        // Get closest normal weight below weight
        long closestBelow = 0L, closestAhead = -1L;
        for (long l = weight - 10; l >= 10; l -= 10) {
            if (this.directResults.containsKey(l)) {
                closestBelow = l;
                break;
            }
        }

        // Get closest normal weight ahead of weight
        for (long l = weight + 10; l <= 10_000; l += 10) {
            if (this.directResults.containsKey(l)) {
                closestAhead = l;
                break;
            }
        }

        // Make sure closestAhead is not -1
        boolean check = false;
        if (closestAhead == -1) {
            closestAhead = this.getClosestAddableWeight(weight);
            check = true;
        }

        // Difference between weight and closestBelow
        long difBelow = weight - closestBelow;
        // Difference between weight and numAhead
        // (Make sure difAhead is not -1 because closestAhead might be smaller than weight)
        long difAhead = Math.abs(closestAhead - weight);

        // Check if closestAhead is equal to sum of all weights (biggestAddable)
        // and the difference to closestAhead is smaller (or equal) to difBelow
        if (closestAhead == this.biggestAddable && difAhead <= difBelow)
           return new IndirectResult(closestAhead, new ArrayList<>(),
                                     this.addables.get(closestAhead));

        // Gets smaller difference
        long smallerDif = Math.min(difBelow, difAhead);

        // Test all weight from (weight-smallerDif) to (weight+smallerDif), going away from weight
        for (long i = 1; i < smallerDif; i++) {
            // Skip weights dividable by 10 because it would be in directResults
            // (-> already checked)
            if (i % 10 == 0) continue;
```

```
47
            // Initialize weight i steps left and right from weight
49          long w1 = weight - i, w2 = weight + i;

51          // Ckeck already existing weights
            if (this.cache.containsKey(w1))
53              return new IndirectResult(w1, this.cache.get(w1));

55          if (this.cache.containsKey(w2))
                return new IndirectResult(w2, this.cache.get(w2));
57
            // Get IndirectResults usning "getIndirectResult" and return them if they are not null
59          IndirectResult indirectResult1 = this.getIndirectResult(w1);
            if (indirectResult1 != null)
61              return indirectResult1;

63          IndirectResult indirectResult2 = this.getIndirectResult(w2);
            if (indirectResult2 != null)
65              return indirectResult2;
        }
67
        // Set closest if there is nothing addable between closestBelow and closestAhead
69      long closest = difAhead < difBelow ? closestAhead : closestBelow;

71      // Set result
        ListResult result;
73      if (closest == 0) {
            result = ListResult.emptyListResult();
75      } else if (!check) {
            result = this.directResults.get(closest);
77      } else {
            result = this.directResults.containsKey(closest) ? this.directResults.get(closest)
79                  : new ListResult(new ArrayList<>(), this.addables.get(closest));
        }
81
        // Return map of closest and result
83      return new IndirectResult(closest, result);
    }
```

4.3.3 Die „getIndirectResult“-Methode

```
    // Getting result of w if notPossibleWeights does not contain w1
2   private IndirectResult getIndirectResult(long w) {
        if (!this.notPossibleWeights.contains(w)) {
4           // Get result of w
            ListResult result = this.getDirectResult(w);
6
            if (result != null) {
8               return new IndirectResult(w, result);
            } else {
10              this.notPossibleWeights.add(w);
            }
12      }

14      return null;
    }
```