Aufgabe 2: Alles Käse

Teilnahme-ID: 66692

Bearbeiter dieser Aufgabe:
Sven Zimmermann

March 2023

Inhalt

1. Lösungsidee

Bei der Aufgabe „Alles Käse“ sollte ein, in Scheiben zerschnittener, Käsequader durch ein Programm wieder zusammengefügt werden. Hierzu sind folgende Beobachtungen relevant:

- Da die Käsescheiben jeweils nur ganzzahlige Seitenlängen (relative Einheiten) besitzen, kann der gesamte Käsequader nur ganzzahlige Seitenlängen besitzen.
- Da jede Scheibe eine Dicke von 1 hat, wird zu Anfang beim Abschneiden einer Scheibe vom Gesamtquader eine der drei Dimensionen des Käsequaders um 1 kleiner. Außerdem ist die Trivialität der Information, das jede Scheibe eine Dicke von 1 hat, zu beachten, sodass nachfolgend die Scheiben lediglich auf ihre zwei ‚nicht trivialen‘ Dimensionen reduziert werden.
- Beim Schneiden des Käsequaders ist es aufgrund der Symmetrie im dreidimensionalen irrelevant, ob in der x/y/z-Dimension von „links“ oder von „rechts“ abgeschnitten wird, da die Änderung in der Größe des Quaders für beide Fälle identisch wäre. Dies gilt ebenso für das Zusammensetzen. Der Quader kann zu jedem Zeitpunkt eindeutig durch seine drei Kantenlängen definiert werden.
- Die kleinste Scheibe muss nicht notwendigerweise die innerste sein (siehe Abb. 1.1). Gleiches gilt für die Oberfläche (mit ‚Oberfläche‘ ist das Produkt der beiden ‚relevanten‘ Kantenlängen gemeint) bei der äußersten Scheibe. Vielmehr ist die Kantenlänge relevant, da jene die Größe des Quaders begrenzt (mit den Kanten sind jene, die Scheibe definierende, Kanten gemeint, also nicht die Kanten der dicke, da jene bei jeder Scheibe eine Länge von 1 haben).
- Die innerste Scheibe des Quaders muss doppelt vorkommen, da der letzte Schnitt des Käsequaders jenen halbiert. Es ist jedoch möglich, dass, neben der innersten, auch andere Scheiben doppelt vorkommen.

Es ergeben sich zum Vorgehen des Wiederaufbaus zwei Möglichkeiten: Zum einen könnte man mit den kleinsten Scheiben starten und versuchen von ihnen ausgehend den Quader von innen heraus aufzubauen. Ebenso möglich wäre der Aufbau von der größten Scheibe ausgehend nach innen (siehe Abb. 1.2). Aus, später noch genauer definierten, Gründen verwende ich die zweite Variante.

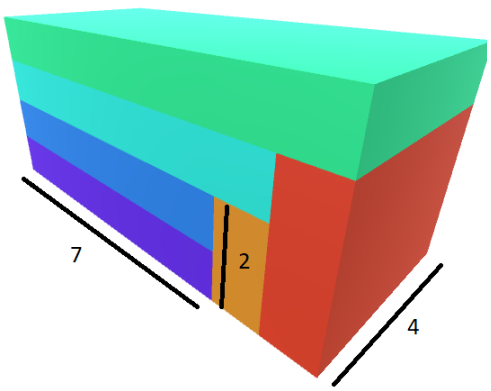

Abbildung 1.1:
Die orangene Scheibe ist kleiner als die blaue & violette (sowohl die Kantenlängen, als auch die Oberfläche). Jedoch kann die orangene Scheibe nicht als 'innerste' genutzt werden.

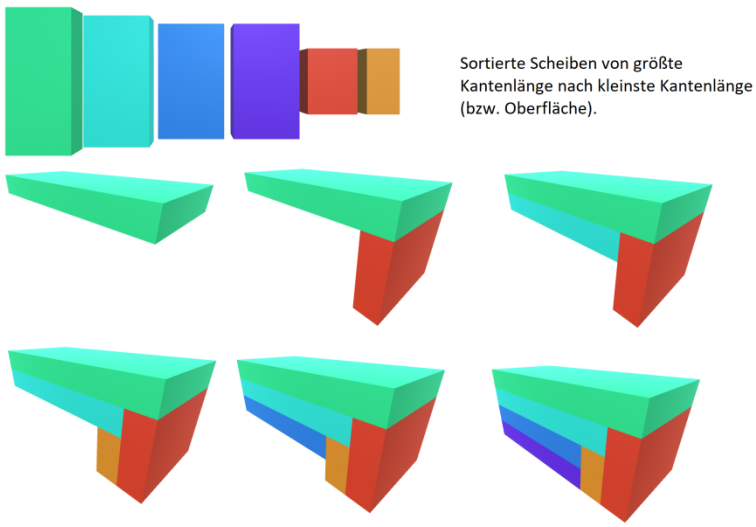

Abbildung 1.2:
Exemplarischer Ablauf des Wiederaufbaus eines Käsequaders von außen nach innen.

1.1 Bestimmung der Startscheibe

Beide beschriebenen Verfahren benötigen eine Startscheibe auf welche sie aufbauen können. Das Problem liegt in der Ermittlung ebenjener Startscheibe, da die Oberfläche einer Scheibe nicht ausschlaggebend für ihre Position im Würfel ist. Vielmehr ist die Kantenlänge der kürzesten beziehungsweise der längsten Kante einer Scheibe relevant. Da sich zwei, in der Zusammenfüge-Sequenz aufeinanderfolgende, Scheiben mindestens eine Kantenlänge Teilen müssen (vgl. Abb. 1.2), gibt es die kürzeste und die längste Kante notwendigerweise mindestens doppelt. Zur Bestimmung der Startscheibe gibt es nun mehrere Möglichkeiten. Bei dem Verfahren des Wiederaufbaus des Käsequaders von innen nach außen, wäre es möglich all jene Scheiben, die die kürzeste Kantenlänge besitzen, herauszusuchen und dann das Verfahren mit jedem von ihnen durchzuprobieren. Zusätzlich dazu könnte man auch noch diejenigen Scheiben ausschließen, die nur einfach und nicht doppelt vorkommen. Für die Bestimmung der Startscheibe bei dem Verfahren des Wiederaufbaus des Käsequaders von außen nach innen ergibt sich noch eine weitere Möglichkeit:

Da alle Scheiben des Käsequaders vorliegen und deren Maße eindeutig bekannt sind, lässt sich sehr einfach das Volumen des Käsequaders aus der Summe der Volumina der einzelnen Scheiben berechnen:

$$V_{Käsewürfel} = \sum_{k=1}^{n} V_k \quad n: Menge\ der\ Scheiben, V_k: Volumen\ der\ k^{ten} - Scheibe$$

Da nun der Käsewürfel ganzzahlige Kantenlängen haben muss, lassen sich alle möglichen Kantenlängen für den Käsewürfel aus den Teilern des Gesamtvolumens berechnen:

$$T_V = \{d \in \mathbb{N}: \mathrm{d}|\mathrm{V}\}$$

Jedes Triplett G_K aus T_V für das gilt $T_{V_1} * T_{V_2} * T_{V_3} = V$ ist eine mögliche Größe für den gesuchten Käsequader.

$$G_K = \{(a, b, c)|a, b, c \in T_V, a * b * c = V\}$$

Für die äußerste Käsescheibe muss gelten, dass ihre beiden (relevanten) Kanten zwei Längen eines Tripletts entsprechen, da die äußerste Scheibe zwei Kantenlängen mit dem Käsequader gemeinsam hat. Ein Programm kann somit für alle Tripletts aus G_K prüfen, ob es eine Scheibe gibt, welche zwei der drei Längen mit dem Triplett teilt. Es ergeben sich also mögliche Startscheiben mit einer, jeweils ihr zugehörigen, Quadergröße. Der Vorteil dieser Methode ist, dass, mit einer gut gewählten Speichermethode für die Käsescheiben, eine Sortierung der Scheiben nicht nötig ist.

1.2 Zusammensetzen des Käsequaders

Das Zusammensetzen des Käsequaders ist, wenn die Größe des Quaders bekannt ist, rekursiv möglich. Das heißt, jede, der Zusammensetz-Sequenz hinzugefügte, Scheibe füllt eine gewisse Menge des nötigen, durch die Größe des Quaders gegebenes, Volumen indem sie an eine der drei möglichen Seiten angefügt wird (vgl. Abb. 1.3).

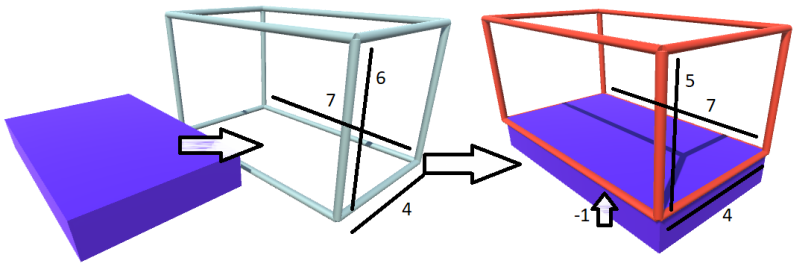

Abbildung 1.3:
Beispielhaftes einfügen einer Scheibe in ein Größen-Gitter

Durch das Einfügen entsteht nun wieder ein quaderförmiges Größen-Gitter, welches sich vom vorherigen Gitter nur in einer Dimension um 1 unterscheidet. In dieses kann nun erneut, falls eine passende Scheibe existiert, eine weitere Scheibe eingefügt werden (vgl. Abb. 1.3). Man beachte, dass durch das Reduzieren einer Dimension um 1 im Größen-Gitter zwei neue mögliche Scheibengrößen entstehen, die eingefügt werden können. In diesem Beispiel wäre zu Anfang nur das Einfügen der Scheiben (6x4), (7x4) und (7x6) möglich. Durch das Einfügen der Scheibe (6x4) (die violette) entstehen nun die beiden Scheiben (5x4) und (7x5) als neue Möglichkeiten. Dafür ist das Einfügen der Scheiben (6x4) und (7x6) nicht mehr möglich. Da durch das Einfügen die möglichen Scheiben immer kleiner werden, ist es sinnvoll, die größten Scheiben möglichst früh einzufügen. Ist die längere der beiden Kanten einer Scheibe am Ende größer, als die größte der drei Dimensionen des Gitters, so gehört die Scheibe entweder nicht zu dem Käsewürfel, die vorausgesetzte Größe des Käsequaders war falsch, oder in der Einfüge-Sequenz wurde eine falsche Entscheidung getroffen. Es ist nämlich denkbar, dass an einer Stelle des Zusammensetzens mehrere Käsescheiben als Möglichkeit zum Einfügen zur Verfügung stehen (z.B. könnte sowohl die Scheibe (6x4), als auch die Scheibe (7x6) im Beispiel verfügbar sein). In diesem Falle ist jedoch die größte Scheibe der Optionen zu wählen, da durch das Einfügen einer kleineren Scheibe die Dimension, in welche die größere eingefügt würde, verkleinert wird und somit die größere Scheibe eventuell nicht mehr in den Quader eingefügt werden kann (vgl. Abb. 1.4).

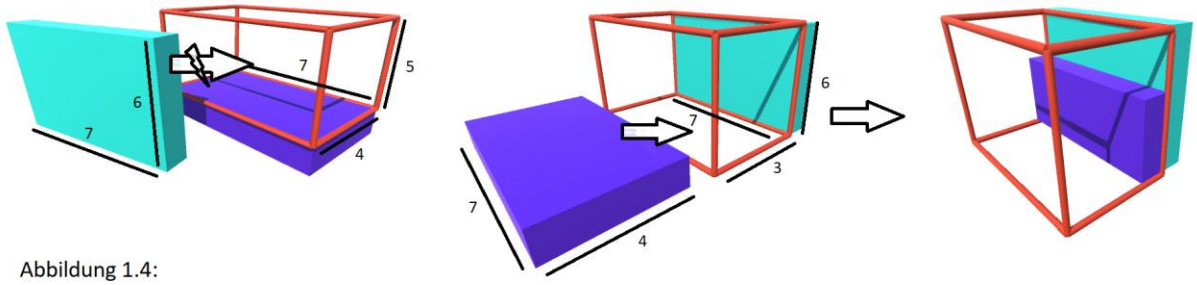

Abbildung 1.4:

Fügt man in das 4x6x7-Gitter zunächst die Violette 4x7-Scheibe ein, so gibt es keine Möglichkeit die Türkise 6x7-Scheibe zu integrieren, da, egal wie viele Scheiben man noch einfügt, niemals wieder zwei Seiten des Gitters die Größe 6x7 haben werden.

Fügt man hingegen zunächst die Türkise 6x7-Scheibe ein, so ist es sehr wohl möglich im späteren Verlauf der Rekonstruktion die Violette 7x4-Scheibe in den Käsewürfel zu integrieren.

Allgemein kann man sagen, dass es immer ratsam ist, zunächst die größere Scheibe einzufügen.

Ist am Ende das ganze Gitter gefüllt, so ist die Reihenfolge der eingefügten Scheiben eine mögliche Lösung für die Zusammenfüge-Sequenz des Käsewürfels.

2 Umsetzung

Der Algorithmus lässt sich gemäß der Lösungsidee in zwei Teile aufteilen. Zum einen die Ermittlung der potentiellen Größe des Käsequaders durch die Volumina der einzelnen Käsescheiben und zum anderen das Zusammenpuzzeln der Scheiben ausgehend von der zuvor ermittelten Größe.

2.1 Implementierung der Bestimmung der potentiellen Größe des Käsequaders

Zunächst gilt es die potentielle Größe des ursprünglichen Käsequaders zu ermitteln. Wie in der Lösungsidee (1.1) beschrieben, lässt sich die Größe aus dem Volumen des Käsequaders bestimmen, da die Kantenlängen des Endquaders ganzzahlig sein müssen. Hierfür wird zunächst die Summe der Produkte der Kantenlängen aller Scheiben gebildet. Beispielsweise (für den ersten Testfall der BWInf-Seite):

$$Scheiben = [[2,4],[4,6],[6,6],[3,4],[4,6],[3,6],[2,4],[2,4],[4,6],[3,3],[3,3],[6,6]]$$
$$Produkte = [8,24,36,12,24,18,8,8,24,9,9,36]$$
$$Volumen = 216$$

Anschließend wird das Volumen in seine Primfaktoren zerlegt:

$$216 => [2,2,2,3,3,3], \quad (2*2*2*3*3*3 = 216)$$

Hierfür zählt ein Algorithmus iterativ von 2 beginnend hoch und fügt, falls das Volumen durch die Zahl teilbar ist, die Zahl sooft einer Liste hinzu, wie das Volumen durch die Zahl teilbar ist:

```
1  Algorithmus: Generiere_Primfaktoren_von_Volumen:
2  Eingabe: Volumen
3  a <- Volumen
4  p <- 2
5  Primfaktoren <- leere Liste
6  solange p^2 kleiner ist als a, tue
7          falls p kein Teiler von a ist, tue
8                  p <- p+1
9          sonst tue
10                 a <- a/p
11                 füge p der Liste Primfaktoren hinzu
12 falls a größer ist als 1, tue
13         füge a der Liste Primfaktoren hinzu
14 Gebe die Liste der Primfaktoren zurück
```

Hierdurch entsteht eine Liste der Primfaktoren einer Zahl.

Aus den Primfaktoren kann nun ein weiterer Algorithmus alle möglichen Teiler des Volumens bestimmen, indem er alle möglichen Produkte aus den Primfaktoren bildet:

```
1  Algorithmus: Kombiniere_Primfaktoren_zu_Teilern:
2  Eingabe: Liste der Primfaktoren des Volumens
3  gebe die Liste Kombiniere_Primfaktoren_zu_Teilern_intern mit der Liste Primfaktoren des Volumens und 0 ohne Duplikate zurück

4  Algorithmus: Kombiniere_Primfaktoren_zu_Teilern_intern:
5  Eingabe: Liste der Primfaktoren des Volumens, Zähler i
6  falls i kleiner ist als die Länge der Liste der Primfaktoren, tue
7          rekursiv_Kombinationen <- führe Algorithmus , Kombiniere_Primfaktoren_zu_Teilern_intern‘ mit der Liste der Primfaktoren des Volumens und i+1
8  aus
9          Kombinationen <- kopiere rekursiv_Kombinationen
10         für jede Kombination k in rekursiv_Kombinationen, tue
11                 füge Kombinationen k*Liste der Primfaktoren an der Stelle i hinzu
12         gebe die Liste Kombinationen zurück
13 gebe eine Liste mit einer 1 zurück
```

Der Algorithmus generiert rekursiv alle Kombinationsmöglichkeiten aus den Primfaktoren des Volumens, indem er jeden Primfaktor einmal mit allen Kombinationsmöglichkeiten der, in der Liste nachfolgenden, Primfaktoren multipliziert und einmal nicht und beide Möglichkeiten dann kombiniert an die höhere Ebene, beziehungsweise den Nutzer zurückgibt:
Beispiel mit Primfaktoren [2, 2, 2, 3, 3, 3]:

[2, 2, 2, 3, 3, 3]	Multipliziert mit Primfaktor	Untere Schicht
Index 0	[216,118,118,54,118,54,54,27, 72,36,36,18,36,18,18,9,72,36,36, 18,36,18,18,9,24,12,12,6,12,6,6,3]	[72,36,36,18,36,18,18,9,24,12,12,6,12,6, 6,3, 24,12,12,6,12,6,6,3,8,4,4,2,4,2,2,1]
Index 1	[72,36,36,18,36,18,18,9,24,12,12,6,12,6, 6,3]	[24,12,12,6,12,6,6,3,8,4,4,2,4,2,2,1]
Index 2	[24,12,12,6,12,6,6,3]	[8,4,4,2,4,2,2,1]
Index 3	[8,4,4,2]	[4,2,2,1]
Index 4	[4,2]	[2,1]
Index 5	[2]	[1]
Index 6 (außerhalb der Liste)	[1]	/

Zur Übersicht ohne Duplikate:
Output: [1, 2, 3, 4, 6, 8, 9, 12, 18, 24, 27, 36, 54, 72, 108, 216]

[2, 2, 2, 3, 3, 3]	Multipliziert mit Primfaktor	Untere Schicht
Index 0	[216,118,72,54,36,27,24,18,12,9,6,3]	[72,36,24,18,12,9,6,4,3,2,1]
Index 1	[72,36,24,18,12,9,6,3]	[24,12,8,6,4,2,1]
Index 2	[24,12,6,3]	[8,4,2,1]
Index 3	[8,4,2]	[4,2,1]
Index 4	[4,2]	[2,1]
Index 5	[2]	[1]
Index 6	[1]	/

Zur Bestimmung der möglichen Volumina gilt es nunmehr noch die Kombinationsmöglichkeiten mit den Scheiben abzugleichen. Hierfür wird in der Liste der Scheiben nach einer Scheibe mit einer Oberfläche gesucht, welche einem Teiler des Volumens entspricht. Dann lassen sich die Maße des Käsequaders aus den Kantenlängen jener Scheibe ableiten und als dritte Kantenlänge der Quotient aus dem Volumen des Käsequaders und dem Teiler.

2.2 Implementierung des Zusammensetzens

Wie in der Lösungsidee beschrieben, muss der Algorithmus jede potentielle Größe des Käsequaders prüfen. Der Ablauf des Zusammensetzens ist immer derselbe:

1. Suche nach allen, auf die Dimensionen des, zu füllenden, Größengitters passenden, Scheibe.
2. Füge die größte, passende Scheibe ein, entferne jene als verfügbare Scheibe und aktualisiere das Größengitter entsprechend. Falls keine Scheibe passt, verwerfe die Größe als potentielle Größe des Käsequaders.
3. Wiederhole Schritte 1&2 bis das Gitter vollständig aufgefüllt ist und, falls bis dahin nicht abgebrochen wurde, gebe die Einfüge-Sequenz als Lösung zurück.

Die Zusammengesetzte Implementierung des Algorithmus in Pseudocode sähe ungefähr so aus:

```
Algorithmus: Setzte_Käsequader_aus_Scheiben_zusammen:
Eingabe: Liste der Scheiben
S <- Liste der Scheiben
V <- 0
für jedes Element aus S tue
        V <- V + Produkt der Kanten des Elements aus S
T <- leere Liste
für jeden Teiler t von V tue
        füge den Teiler t in T ein
G <- leere Liste
für jedes Element e aus T tue
        für jede Scheibe s aus S tue
                falls das Produkt der Kanten von s gleich e ist, tue
                        a <- erste Kante von s
                        b <- zweite Kante von s
                        c <- V/a/b
                        füge der Liste G die mögliche Größe (a,b,c) hinzu
für jede Größe g aus G tue
        Lösungssequenz <- leere Liste
        pS <- Kopie von S
        solange keine Dimension d aus g gleich 0 ist, tue
                n <- leere Liste
                m <- leere Liste
                für jedes Paar aus g tue
                        füge das Produkt des Paares der Liste m hinzu
                für jede Scheibe s in pS tue
                        falls das Produkt der Kanten von s in m ist und eine Kante von s in g ist, tue
                                füge die Scheibe s der Liste n hinzu
                falls n leer ist tue
                        breche die Schleife ab
                größtesN <- leere Liste
                für jede Scheibe aus n tue
                        falls das Produkt der Kanten aus n größer ist als das Produkt der Kanten aus größtesN, tue
                                größtesN <- n
                füge größtesN der Liste Lösungssequenz hinzu
                für jede Dimension d aus g tue
                        falls d nicht in größtesN, tue
                                d <- d-1
                lösche größtesN aus pS
        falls eine Dimension aus d gleich 0 ist, tue
                gebe die umgekehrte Reihenfolge von Lösungssequenz zurück
gebe nichts zurück
```

2.3 Optimierung

Die größte Schwierigkeit stellt die Speicherung der Scheiben zur optimalen Nutzung jener im Algorithmus dar. Denn, da mit einer Startscheibe kein Erfolg des Zusammenfügens gewährleistet ist, muss der Algorithmus eventuell mehrmals auf die Scheiben zugreifen. Daher ist es nicht möglich beispielsweise bereits verwendete Scheiben zu löschen ohne vorher eine Kopie jener zu erstellen. Das Problem hierbei ist, dass das Kopieren, besonders bei großen Datenmengen, in dem Fall viele Käsescheiben, sehr Zeit- und Ressourcenaufwendig ist. Auch das permanente Durchsuchen oder Filtern der Scheiben nach einer passenden, wäre sehr Zeitaufwendig.

2.3.1 Optimierung durch alternative Datenstrukturen

Wie dem Pseudocode zu entnehmen ist, ist bei dem Durchsuchen der Liste an Scheiben besonders das Produkt der Kantenlängen relevant. Es ist daher naheliegend anstelle einer Liste eine assoziative Liste (alias Wörterbuch) zu verwenden und als Schlüssel das Produkt der Kantenlängen:
Aus…

$$Scheiben = \big[[2,4],[4,6],[6,6],[3,4],[4,6],[3,6],[2,4],[2,4],[4,6],[3,3],[3,3],[6,6]\big]$$

Wird…

$$Scheiben =$$
$$\{8\text{: }[(4,2),(4,2),(4,2)], 24\text{: }[(6,4),(6,4),(6,4)], 36\text{: }[(6,6),(6,6)], 12\text{: }[(4,3)], 18\text{: }[(6,3)], 9\text{: }[(3,3),(3,3)]\}$$

Hierdurch vereinfacht sich sowohl das Filtern der Größen (vgl. Pseudocode Z.12f.), als auch das Zusammensetzen der Scheiben (vgl. Pseudocode Z. 26f.), da das lineare Durchsuchen der Scheibenliste wegfällt. Um das Durchsuchen der Werte der assoziativen Liste weiter zu vereinfachen, lässt sich desweiteren erkennen, dass durch die Speicherung der Scheiben im Wörterbuch redundant Information gespeichert wird. Es reicht nämlich lediglich eine der zwei Kanten jeder Scheibe zu speichern, da sich die andere aus dem Schlüssel (Key), welcher das Produkt der Kantenlängen ist, und der bekannten Kante berechnen lässt.

Beispielsweise lässt sich die zweite Kante in der unterstrichenen Scheibe aus dem Quotient des Produkts der Kantenlängen (8) und der ersten Kante (4) berechnen:

$$K_2 = \frac{P}{K_1} = \frac{8}{4} = 2, \qquad da\ P = K_1 * K_2 = 4 * 2 = 8$$

Somit lässt sich die assoziative Liste auch schreiben als:

$$Scheiben = \{8\text{: }[4,4,4], 24\text{: }[6,6,6], 36\text{: }[6,6], 12\text{: }[4], 18\text{: }[6], 9\text{: }[3,3]\}$$

Wobei es für die spätere Erweiterung des Algorithmus sinnvoll ist, jeweils die längere der beiden Kanten als Eintrag zu nutzen.

2.3.2 Optimierung durch Filter

Trotz der Verbesserung des Algorithmus durch ein Wörterbuch ist jedoch immer noch die Kopierung des gesamten Datensatzes für jede mögliche Größe des Käsequaders nötig. Da dies bei großen Mengen an Scheiben sehr Zeit- und Ressourcenaufwändig ist, empfiehlt es sich stattdessen einen Filter zu nutzen, welcher bei jeder Referenz des Datensatzes über jenen gelegt und dabei die bereits verwendeten Scheiben heraus filtert. Dies funktioniert indem der Filter dieselbe ‚Form' (shape), wie das Scheiben-Wörterbuch, und dieselben Schlüssel hat, nur mit dem Unterschied, dass statt den Kantenlängen der Scheiben, eine 1 für nichtbenutzt oder eine 0 für benutzt als Eintrag im Wörterbuch steht. Dies ermöglicht das Filtern durch einfache punktuelle Multiplikation der Liste der Scheiben mit der Liste des Filters an dem entsprechenden Schlüssel:
Beispielsweise:

$$Scheiben = \{8\text{: }[4,4,4], 24\text{: }[6,6,6], 36\text{: }[6,6], 12\text{: }[4], 18\text{: }[6], 9\text{: }[3,3]\}$$

$$Filter = \{8\text{: }[1,1,1], 24\text{: }[0,1,1], 36\text{: }[0,0], 12\text{: }[1], 18\text{: }[1], 9\text{: }[1,1]\}$$

Sucht man nun nach einer Scheibe mit den Maßen 6x4 (also Schlüssel 24) so erhält man beim punktuellen Multiplizieren:

$$Scheiben[24] = [6,6,6] \qquad Filter[24] = [0,1,1]$$

$$=> \quad [6*0, 6*1, 6*1] = [0,6,6]$$

Was bedeutet, dass noch zwei Scheiben mit den Maßen $\left(6, \frac{24}{6}\right)$ und $\left(6, \frac{24}{6}\right)$ verfügbar sind.

Nun scheint das Erstellen des Filters ebenso speicherintensiv, wie das Kopieren der Scheiben, jedoch ist es nicht nötig bei dem Filter, wie bei den Scheiben, direkt zu Anfang den gesamten Datensatz zu

kopieren. Viele mögliche Käsequader-Größen können schon innerhalb der ersten paar Iterationen ausgeschlossen werden, sodass gar nicht alle Schlüssel und Werte des Scheiben-Wörterbuchs abgerufen werden müssen. Im Gegensatz zu der Implementierung ohne den Filter ist es jedoch mit Filter hierbei nicht nötig den gesamten Datensatz zu kopieren, da zwar auch ungewiss ist, welche Schlüssel noch vor dem Ausschließen der Größe relevant sind, jedoch der Filter nicht zwangsweise für alle Schlüssel Einträge haben muss. So lässt sich zum Beispiel im Programm definieren, dass, wenn es keinen Eintrag für einen Schlüssel im Filter gibt, auch noch keine Scheibe mit dieser Größe verwendet wurde. Erst nämlich, wenn eine Scheibe verwendet wird und diese noch keine Liste im Filter besitzt, wird auch ein Eintrag für diese im Filter erstellt. Hierdurch wächst die Größe des Filters erst mit den Iterationen beim Verwenden von Käsescheiben während des Zusammensetzens des Käsequaders und es muss nicht sofort der gesamte Filter erstellt werden. Der Filter im oberen Beispiel kann daher auch so geschrieben werden:

$$Filter = \{24\colon [0, 1, 1], 36\colon [0, 0]\}$$

Und würde trotzdem denselben Zweck erfüllen.

3 Beispiele

Ausgaben meines Programms zu den Beispieleingaben der BWInfo-Webside (https://bwinf.de/bundeswettbewerb/41/2/):

kaese1.txt:
Output:
[[2, 4], [2, 4], [4, 2], [4, 3], [3, 3], [3, 3], [6, 3], [4, 6], [4, 6], [6, 4], [6, 6], [6, 6]]

kaese2.txt:
Output:
[[998, 999], [999, 998], [998, 2], [2, 1000], [1000, 2]]

kaese3.txt:
Output:
[[992, 995], [995, 992], [995, 2], [993, 2], [996, 2], [994, 2], [997, 2], [995, 997], [997, 995], [995, 4], [4, 998], [998, 4], [997, 4], [999, 4], [999, 998], [998, 5], [1000, 998], [1000, 6], [1000, 999], [1000, 7], [1000, 1000], [1000, 1000], [1000, 1000]]

kaese4.txt:
Output:
[[29, 51], [29, 51], [51, 29], [29, 3], [52, 3], [3, 30], [3, 30], [30, 3], [3, 55], [55, 3], [...][1], [207, 207], [207, 206], [208, 207], [207, 207], [209, 207], [208, 209], [209, 208], [209, 209], [209, 209], [210, 209], [210, 210]]

kaese5.txt:
Output:
[[437, 1325], [437, 1325], [1325, 437], [1325, 3], [438, 3], [1326, 438], [4, 1326], [1326, 4], [1326, 440], [5, 1326], [...], [3567, 2726], [2726, 2308], [2308, 3568], [3568, 2308], [2728, 2308], [2308, 3569], [3569, 2308], [3569, 2730], [2730, 2309], [3570, 2730]]

[1] Jeweils nur die ersten und letzten 10 Scheiben der Sequenz

kaese6.txt:
Output:
[[9255, 480255], [480255, 9255], [480255, 2], [9256, 480255], [480255, 9256], [9256, 4], [480256, 4], [480256, 9257], [9257, 5], [480257, 5], [...], [510506, 30027], [510506, 39268], [39268, 30028], [510507, 30028], [30028, 39269], [39269, 30028], [510509, 39269], [510509, 30029], [39270, 30029], [510510, 39270]]

kaese7.txt:
Output:
[[665, 962], [962, 665], [2, 962], [962, 2], [667, 962], [962, 667], [962, 4], [4, 668], [4, 668], [668, 4], [...], [510505, 510509], [510509, 510505], [510509, 510508], [510506, 510509], [510509, 510506], [510509, 510510], [510510, 510509], [510510, 510508], [510510, 510510], [510510, 510510]]

4 Quellcode: Implementierung in Python

Nachfolgend ist das Hauptprogramm meiner Implementierung in Python 3.10:

```
Import numpy as np
def recreate_cheese_cuboid(slices):
    """
    recreate_cheese_cuboid baut aus einer Liste an Scheiben (slices), welche durch das gerade Zerschneiden
    eines Käsequaders in gleichmäßige 1 LE-Dicke Scheiben entstehen,
    den Käsequader wieder zusammen und gibt die Sequenz des Zusammenfügens, falls eine existiert, aus.
    """
    #Schritt 1: Bestimmung der potentiellen Größen des Käsequaders
    #Schritt 1.1: Berechnung des Volumens des Käsequaders aus der Summe der Volumina der Scheiben
    total_volume = 0
    for key in slices:
        total_volume += key*len(slices[key])

    #Schritt 1.2: Bestimmung der Teiler des Volumens
    primes_of_volume = generate_primefactors(total_volume)
    combinations_volume = prod_combinations(primes_of_volume)

    #Schritt 1.3: Überprüfung der Teiler auf Übereinstimmung mit Maßen der Scheiben
    possible_cubes = []
    for combination in combinations_volume:
        if slices.get(combination) != None:
            new_available_faces = []
            for edge in slices.get(combination):
                if edge in combinations_volume:
                    new_available_faces.append(edge)
            new_available_faces = list(dict.fromkeys(new_available_faces))
            for n in new_available_faces:
                possible_cubes.append(([n, int(combination/n)], (n, int(combination/n),
int(np.ceil(total_volume/combination)))))

    #Phase 2: Testen der Zusammensetzbarkeit für jede ermittelten Größe
    for first_face, cube_size in possible_cubes:
        #Phase 2.1: Initiierung des Filters und der Lösungssequenz
        slices_filter = {}
        slices_filter[prod(first_face)] = [1 for x in range(len(slices[prod(first_face)]))]
        slices_filter[prod(first_face)][slices[prod(first_face)].index(first_face[0])] = 0
        solution = [first_face]
        current_size = [cube_size[0], cube_size[1], cube_size[2]-1]
        added_face = True
```

```
        #Phase 2.2: Wiederholendes Einfügen von Scheiben in das 'Größengitter', bis keine Scheibe mehr eingefügt werden kann
        while True:
            #Phase 2.2.1: Durchsuche die Scheiben auf eine Passende (Beachtung des Filters)
            pot_face = []
            for face_size in [(current_size[0], current_size[1]),(current_size[0], current_size[2]),(current_size[1],current_size[2])]:
                if slices.get(prod(face_size)) != None:
                    if slices_filter.get(prod(face_size)) == None:
                        slices_filter[prod(face_size)] = [1 for x in range(len(slices.get(prod(face_size))))]
                    if sum(map(lambda x,y:x*y, slices.get(prod(face_size)), slices_filter.get(prod(face_size)))) > 0: #Punkweise Multiplikation der Scheiben mit dem Filter
                        for n in face_size:
                            if n in map(lambda x,y:x*y, slices.get(prod(face_size)), slices_filter.get(prod(face_size))) and n in current_size:
                                pot_face.append((n, int(prod(face_size)/n)))
            #Phase 2.2.2: Falls es mehrere Mögliche Scheiben zum einfügen gibt, wähle die Größte
            face = []
            if len(pot_face) == 0:
                #Phase 2.2.2.1: Falls keine Scheibe eingefügt werden kann, breche den Wiederaufbau ab
                break
            elif len(pot_face) > 1:
                for pface in pot_face:
                    if prod(face) < prod(pface):
                        face = list(pface)
            else:
                face = list(pot_face[0])
            #Phase 2.2.3: Füge die Scheibe ein
            solution.append(face)
            slices_filter.get(prod(face))[list(map(lambda x,y:x*y,slices.get(prod(face)),slices_filter.get(prod(face)))).index(face[0])] = 0
            current_size[not_match_index(current_size, face)] -= 1
        #Phase 2.3: Überprüfung, ob das Käse-Größengitter ausgefüllt ist und, falls ja, mögliche Lösung invertiert zurückgeben
        if one_equals(current_size, [0,0,0]):
            return solution[::-1]
```

5 Erweiterungen

Dank der repetierenden Eigenschaft des Zusammensetzens des Käsequaders und der Abbruchbedingung des ausgefüllten Käse-Größengitters anstelle der Nutzung aller Scheiben, ist es möglich den Algorithmus zu erweitern, sodass er die Scheiben mehrerer Käsequader einlesen und dann separiert zusammensetzen kann. Außerdem ist es möglich, aus der Scheibenliste entfernte, Scheiben zu erkennen und zu ersetzen, da der Algorithmus auch rekursiv implementierbar ist:

5.1 Mischen der Scheiben mehrerer Käsequader

Das Problem bei einem Input, welcher die Scheiben mehrerer Käsequader enthält, gegenüber einem Input, welcher immer nur die Scheiben eines Käsequaders enthält, ist das, dass die Größe der einzelnen Käsequader nicht mehr so einfach bestimmt werden können. So ist die Summe der Volumina aller Scheiben nicht mehr notwendigerweise das Volumen eines Käsequaders, sondern kann die Summer der Volumina mehrerer Käsequader sein:

$$V_K \leq \sum_{i=1}^{p} V_{K\,i}, \qquad V_{k_i}: Volumen\ des\ i^{ten} - K\ddot{a}sequaders, p: Menge\ der\ gemischten\ K\ddot{a}sequader$$

$$V_K = V_{K_z}, z \in \mathbb{N}^{\leq p} \;\Leftrightarrow\; p = 1 \;\; v. \;\; \sum_{i=1}^{p} V_{K_i} = V_{K_z}, \;\; z \in \mathbb{N}^{\leq p} \;\Rightarrow V_{K_i} = 0 \;\; \forall \;\; i \in \mathbb{N}^{\leq p} \backslash \{z\}$$

Das Volumen der Summe aller Volumina der gemischten Käsequader entspricht nur dann dem Volumen eines ‚echten' Käsequaders, wenn nur die Summe von einem Käsequader zu dem Gesamtvolumen beiträgt.

$$\sum_{i=1}^{p} V_{K_i} = \sum_{k=1}^{n} V_k \,, \qquad n: Menge\ der\ Scheiben, V_k: Volumen\ der\ k^{ten} - Scheibe$$

Das Problem ist auch, dass die Menge der Scheiben (n) unabhängig von der Anzahl der durchmischten Käsequader (p) ist. Dadurch lässt sich nicht, allein aus der Menge der Scheiben, vor der Anwendung des Algorithmus sagen, wie viele Käsequader vermischt wurden.

Wie bereits erwähnt, ist der Trick, zur Dispersion der Käsescheiben und zur Gruppierung der Scheiben zu einem Käsequader, die rekursive (selbstreferenzierende) Implementierung des Algorithmus. Das heißt in der Praxis, dass der Algorithmus versuchen soll, wenn noch Scheiben nach der Rekonstruktion eines Quaders übrig sind, aus diesen ebenfalls einen Quader zusammenzusetzen.

5.1.1 Bestimmung der Startscheibe

Da sich die Startscheibe bei mehreren vermischten Käsequadern nicht mehr durch die möglichen Volumina des Käsequaders bestimmen lässt und nichteimal bestimmbar ist welche Käsescheibe welchem Käsequader zugehört, gilt es eine neue Methode für das Bestimmen der Volumina und daraus die Startscheiben der einzelnen Käsequader zu finden. Die Idee ist, sich immer nur auf einen Käsequader auf einmal zu konzentrieren und dabei zu ignorieren, dass es überhaupt Scheiben anderer Käsequader gibt.

Am einfachsten ist es nun, mit dem ‚Größten' Quader zu beginnen, wobei alle Scheiben nach der Scheibe durchsucht werden, welche die längste Kante hat. Wie nämlich schon zuvor bemerkt, muss diese Scheibe die äußerste sein, da durch das Einfügen von Scheiben in das Scheibengitter (vgl. Kapitel 1.2) die Kanten für das Einfügen weiterer Scheiben nur kleiner werden. Falls die längste Kante doppelt vorkommen sollte, ist wieder die größere Scheibe zu bevorzugen. Nutzt man hierbei ebenfalls die assoziative Liste (vgl. Kapitel 2.3.1), so sähe ein möglicher Algorithmus zur Bestimmung der Startscheibe zum Beispiel so aus (Python 3.10):

```
Import numpy as np
Import sidefunc as sf #sidefunc enthält einige 'triviale' Funktionen
def generate_sizes(slices, filt):
        """
        generate_sizes nimmt ein Wörterbuch mit Käsescheiben, welche aus dem Zerschneiden eines oder mehrerer Käsequader entstanden sind,
        als Input und gibt mögliche Größen für den größten (ausgehend von der Scheibe mit der längsten Kante) Käsequader zurück,
        sowie die dazugehörigen 'Randscheiben' des Käsequaders.
        """
        #1. Konvertiere das Scheiben-Wörterbuch (slices) in ein Kanten-Wörterbuch, wobei statt dem Produkt der Kanten die jeweils längere Kante der
Schlüssel ist
        #und alle Kanten, die mit dieser längeren Kante in Kombination in einer der Scheiben auftreten, die Werte sind.
        edges = sf.isolate_edges(slices, filt)
        print(edges)
        #2. Suche nach der längsten Kante
        largest_face, num_duplicates = [0, 0], 0
        for edge, value in edges.items():
                if edge > largest_face[0]:
                        largest_face[0] = edge
                        largest_face[1] = 0
                        num_duplicates = 0
```

```
                for v in value:
                    #2.1 Füge nur die längsten 'Kombinationskanten' (Kanten, die mit der längsten Kante vorkommen) als mögliche Startscheibe hinzu.
                    if v > largest_face[1]:
                        largest_face[1] = v
                        num_duplicates = 1
                    #2.2 Falls die 'größte' Scheibe mehrfach vorkommt, zähle wie oft.
                    elif v == largest_face[1]:
                        num_duplicates += 1
        #3. Generiere aus den größten gefundenen Scheiben mögliche Größen für den 'größten' Käsequader
        possible_cubes = []
        #3.1 Nutze nicht notwendigerweise alle größten Scheiben, da jene möglicherweise von unterschiedlichen Quadern kommen
        for d in range(1, num_duplicates+1):
            all_slices_list = sf.edges_to_list(edges) #konvertiert Kanten-Wörterbuch wieder zu einer Liste an Scheiben
            #all_slices_list sieht z.B. so aus: [[4, 2], [4, 2], [4, 2], [4, 3], [6, 4], [6, 4], [6, 4], [6, 6], [6, 6], [6, 3], [3, 3], [3, 3]]
            cut_slices_list = sf.remove_usedANDduplicates(all_slices_list, [largest_face for x in range(d)]) #Entfernt Redundante Scheiben und die benutzten größten Scheiben aus der Scheibenliste
            #cut_slices_list sieht z.B. so aus: [[4, 2], [4, 3], [6, 4], [6, 6], [6, 3], [3, 3]] (es wurde nur 1x [6, 6] genutzt)
            split_edges_list = sf.separate(cut_slices_list) #teilt die Scheiben in einzelne Kanten auf:
            #split_edges_list sieht z.B. so aus: [4, 2, 4, 3, 6, 4, 6, 6, 6, 3, 3, 3]
            possibility = []
            for e in range(len(split_edges_list)):
                if split_edges_list[e] == largest_face[1]:
                    #Die Kante mit dem Index e in split_edges_list gehört zur Scheibe mit dem Index ⌊e/2⌋ in cut_slices_list
                    possibility.append(int(np.floor(e/2)))
            for p in possibility:
                used_slices = [largest_face for x in range(d)]
                used_slices.append(cut_slices_list[p])
                #3.2 Füge die, sich ergebende, Größe den möglichen Größen hinzu, wobei die Verwendeten Randscheiben abzuziehen sind
                potential_remaining_size = [largest_face[0]-1, largest_face[1], cut_slices_list[p][sf.not_match_index(cut_slices_list[p], largest_face)]]
                possible_cubes.append([used_slices, potential_remaining_size])
        return possible_cubes
```

5.1.2 Zusammensetzen der Käsequader

Bei den vermischten Scheiben mehrerer Quader kann das Zusammensetzten trotzdem genauso verlaufen, wie bei den Scheiben nur eines Quaders. Der Unterschied und das Problem dabei sind, dass dabei nicht notwendigerweise der richtige Quader entsteht. So können Scheiben eines anderen Quaders verwendet werden, welche dafür sorgen, dass jener nicht mehr zusammenfügebar ist. Das heißt, dass anders als es vorher war, das hinzufügen einer Scheibe nicht unbedingt richtig sein muss, nur weil es die größtmögliche Scheibe ist.

Das führt dazu, dass der (in Kapitel 2.2) beschriebene Algorithmus rekursiv implementiert werden muss, wobei, wenn der Algorithmus an einer Stelle mehrere Optionen zum Einfügen einer Scheibe in das Größen-Gitter hat, er immer ‚alle Optionen gleichzeitig durchprobiert'. Der neue Algorithmus besteht also aus folgenden Schritten:

1. Falls das erhaltene Größengitter bereits gefüllt ist, gebe nichts zurück (Rekursionsanker).
2. Suche nach, in das Größen-Gitter passenden, Scheiben.
    a. Erstelle die möglichen neuen Gittergrößen und aktualisiere die Scheibenliste entsprechend.
    b. Rufe den Algorithmus für jede neue Gittergröße mit den aktualisierten Scheibenlisten erneut auf (versuche mit den, noch vorhandenen, Scheiben das Gitter zu füllen).
    c. Erweitere die, aus den ‚tieferen Schichten' erhaltene Lösung um die eingefügte Scheibe und durchsuche die resultierenden Lösungen nach der besten, anhand des Kriteriums, wie viele Scheiben unbenutzt bleiben (je weniger desto besser).

3. Gebe die beste Lösung und die ungenutzten Scheiben (an die höhere Schicht beziehungsweise den Nutzer) zurück.

Bei der Implementierung dieses Algorithmus mit der Optimierung des Filters (vgl. Kapitel 2.3.2) ergibt sich bei der Rekursion jedoch ein Problem. Denn, da der Filter, als eine assoziative Liste, kein primitiver Datentyp ist, wird er nicht kopiert und Änderungen an ihm in ‚tieferen Schichten' der Rekursion hätten einen Einfluss auf die ‚höheren Schichten'. Daher muss der Filter jedes Mal kopiert werden, sodass Änderungen keinen Einfluss auf andere Iterationen haben.

5.2 Entfernen von Scheiben aus der Scheibenliste

Die Erweiterung des rekursiven Algorithmus (aus Kapitel 5.1), sodass dieser auch Fehlende Scheiben erkennen kann, ist dadurch möglich, dass man beim Rekursionsschritt, also dem Aufrufen des Algorithmus innerhalb des Algorithmus (vgl. Kapitel 5.1.2: Schritt 2) die Verfügbarkeit der Scheibe gar nicht erst überprüft. Sollte es dann, durch das Einfügen der Scheibe, zu einer Lösung kommen, wird die Scheibe, sollte sie nicht in der Scheibenliste existieren, als ‚fehlend' hinzugefügt.

Hierdraus ergibt sich ein Problem, da nun der Algorithmus theoretisch alle Scheiben, die nötig wären eine Lösung zu generieren, als ‚fehlend' deklarieren und hinzufügen könnte und somit immer zu einer Lösung kommen würde, die alle Käsescheiben nutzt. Um dem entgegen zu wirken, kann das Kriterium ‚Menge an hinzugefügten Scheiben' neben ‚Menge an ungenutzten Scheiben' zur Bewertung der Qualität einer Lösung genutzt werden. Dadurch sucht der Algorithmus rekursiv nach einer Lösung, welche möglichst alle Käsescheiben, mit möglichst wenig hinzugefügten Scheiben, zur Rekonstruktion von Käsequadern nutzt.

5.3 Probleme des erweiterten Algorithmus und Optimierung

Das größte Problem ist der deutliche Verlust in Geschwindigkeit und die starke Zunahme in Rechenintensität des Algorithmus durch die Implementierung der beiden Erweiterungen. Besonders durch die Rekursivität kann der Algorithmus, durch immer ‚tiefere' Rekursion hinsichtlich des Hinzufügens der Scheiben und zusätzlichen Quadern, sehr lange für das Lösen, selbst nicht-modifizierter Scheibenlisten, benötigen. Daher ist es ratsam, eine Rekursionstiefenregulierung zu implementieren, die die Anzahl hinzuzufügender Scheiben und die Menge an Quadern begrenzt.

5.4 Quellcode: Implementierung in Python

Nachfolgend ist das Hauptprogramm meiner Implementierung der Erweiterungen in Python 3.10:

```
import numpy as np
import sidefunc as sf

def cubeit(slices, filt={}, p_adds=2, remaining_cubes=2):
        possible_solutions = []
        #1. Initiierung des Filters
        for key, value in slices.items():
                if filt.get(key) == None:
                        filt[key] = [1 for x in range(len(value))]
        #2. Generieren der möglichen Größen des 'größten' Käsequaders
        sizes = generate_sizes(slices, filt)

        #3. Durchprobieren aller Größen
        for first_faces, cube_size in sizes:
                slices_filter = sf.copy_dict(filt)
                for sl in first_faces:
                        slices_filter[sf.prod(sl)][list(map(lambda x,y:x*y, slices.get(sf.prod(sl)), slices_filter[sf.prod(sl)])).index(sl[0])] = 0
                sol = recreate_cheese_cuboid([[first_faces[-1]], cube_size], slices, slices_filter, p_adds, remaining_cubes)
                if sol != None:
                        s = [first_faces[:-1]]
```

```
                                        s[0].extend(sol[0][0])
                                        if len(sol[0]) > 1:
                                                s.extend(sol[0][1:])
                                        possible_solutions.append([s, sol[1], sol[2]])
        #4. Filtern der Lösungen und Invertieren der Lösungssequenz
        if len(possible_solutions) == 0:
                return None
        best_sol = possible_solutions[0]
        if len(possible_solutions) > 1:
                for sol in possible_solutions[1:]:
                        #Die beste Lösung ist die, die so wenig Scheiben wie möglich einfügen muss und so viele Scheiben wie möglich nutzt.
                        if len(sol[1]) + len(sol[2]) < len(best_sol[1]) + len(best_sol[2]):
                                best_sol = sol
        invert_sol = [best_sol[0][0][::-1]]
        if len(best_sol[0]) > 1:
                for s in best_sol[0][1:]:
                        invert_sol.append(s)
        return [invert_sol, best_sol[1], best_sol[2]]

def recreate_cheese_cuboid(sizes, slices, filt, remaining_adds=0, remaining_cubes=0):
        """
        recreate_cheese_cuboid generiert rekursiv die optimale Sequenz zum Zusammenfügen einer Liste an Scheiben zu mehreren Käsequadern.
        sizes: Liste an möglichen Größen für den 'größten' Quader mit zugehörigen 'Randscheiben'
        slices: assoziative List/Wörterbuch mit den Scheiben (key=Produkt der Kanten, value=Liste der Kanten der Scheiben dessen Kanten das Produkt des zugehörigen keys ergeben)
        filt: Filter für die slices (sodass slices nicht kopiert werden muss)
        remaining_adds: Integer der zur Begrenzung der Rekursionstiefe dient, indem er die Menge der Hinzufügbaren Scheiben begrenzt
        remaining_cubes: Integer der zur Begrenzung der Rekursionstiefe dient, indem er die Menge der gesuchten, vermischen Quader begrenzt
        """
        possible_solutions = []
        #1. Rekursionsanker: Falls das Größengitter gefüllt ist:
        if sf.one_equals(sizes[1], [0,0,0]):
                #1.1 Bestimme ob es übriggebliebene Scheiben gibt.
                remaining = []
                for key, value in slices.items():
                        for v in list(map(lambda x,y:x*y, slices.get(key), filt.get(key))):
                                if v != 0:
                                        remaining.append([v, int(key/v)])
                #1.2 Falls ja, versuche aus ihnen ebenfalls einen Quader zusammenzubauen
                if len(remaining) > 0 and remaining_cubes>0:
                        s1 = [sizes[0]]
                        s2 = cubeit(slices, filt, remaining_adds, remaining_cubes-1)
                        #1.2.1 Falls das möglich ist, füge die Lösungen zusammen
                        if s2 == None:
                                return [s1, [], remaining]
                        s1.extend(s2[0])
                        return [s1, s2[1], s2[2]]
                return [[sizes[0]], [], remaining]
        #2. Bestimme, in das Größengitter einfügbaren, Scheiben
        solution = sizes[0]
        for dim in range(3):
                p_face = sizes[1].copy()
                p_face.pop(dim)
                p_new_size = sizes[1].copy()
                p_new_size[dim] -= 1
                #2.1 Überprüfe ob es die Scheibe in der Scheibenliste gibt
                if sf.does_contain(slices, filt, p_face):
                        #2.1.1 falls ja, führe für sie den Algorithmus mit dem aktualisierten Größengitter rekursiv aus
                        new_slices_filter = sf.copy_dict(filt)
                        new_slices_filter[sf.prod(p_face)][list(map(lambda x,y:x*y, slices.get(sf.prod(p_face)), filt[sf.prod(p_face)])).index(p_face[0])] = 0
                        new_solution = recreate_cheese_cuboid([[p_face], p_new_size], slices, new_slices_filter, remaining_adds, remaining_cubes)
                        #2.1.2 Falls es für die eingefügte Scheibe eine Lösung gibt, füge der Lösung die eingefügte Scheibe hinzu und speichere die Lösung als mögliche Gesamtlösung
                        if new_solution != None:
                                p_solution = solution.copy()
                                p_solution.extend(new_solution[0][0])
                                total_sol = [p_solution]
                                if len(new_solution[0]) > 1:
                                        total_sol.extend(new_solution[0][1:])
                                possible_solutions.append([total_sol, new_solution[1], new_solution[2]])
```

```
                #2.2 Falls es die Scheibe nicht gibt und noch nicht bereits einige Scheiben hinzugefügt wurden,
                elif remaining_adds > 0:
                        #2.2.1 'tue so' als ob es die Scheibe gäbe und führe den Algorithmus trotzdem rekursiv mit dem aktualisierten
Größengitter aus
                        new_slices_filter = sf.copy_dict(filt)
                        new_solution = recreate_cheese_cuboid([[p_face], p_new_size], slices, new_slices_filter, remaining_adds-1,
remaining_cubes)
                        #2.2.2 Falls es für die eingefügte Scheibe eine Lösung gibt, füge der Lösung und der Sammlung hinzugefügter Scheiben
                        #die eingefügte Scheibe hinzu und speichere die Lösung als mögliche Gesamtlösung
                        if new_solution != None:
                                p_solution = solution.copy()
                                p_solution.extend(new_solution[0][0])
                                p_added = new_solution[1]
                                p_added.append(p_face)
                                total_sol = [p_solution]
                                if len(new_solution[0]) > 1:
                                        total_sol.extend(new_solution[0][1:])
                                possible_solutions.append([total_sol, p_added, new_solution[2]])
        #3. Filtere die Lösungen nach der besten und gebe diese an die nächsthöhere Schicht/den Benutzer zurück:
        if len(possible_solutions) == 0:
                return None
        best_sol = possible_solutions[0]
        if len(possible_solutions) > 1:
                for sol in possible_solutions[1:]:
                        #Die beste Lösung ist die, die so wenig Scheiben wie möglich einfügen muss und so viele Scheiben wie möglich nutzt.
                        if len(sol[1]) + len(sol[2]) < len(best_sol[1]) + len(best_sol[2]):
                                best_sol = sol
        return best_sol
```

Ausgabeformat: [Lösungssequenzen (je Quader), hinzugefügte Scheiben, übriggebliebene Scheiben]

5.5 Beispiele

Ausgaben meines Programms (Kapitel 5.4) zu Kombinationen und Modifikationen der Testfälle der BWInf-Seite (https://bwinf.de/bundeswettbewerb/41/2/):

5.5.1 nicht-modifizierte Scheibenlisten

kaese1.txt:

Output:

Würfel 1: [[4, 2], [4, 2], [4, 2], [4, 3], [3, 3], [3, 3], [6, 3], [6, 4], [6, 4], [6, 4], [6, 6], [6, 6]]

kaese2.txt:

Output:

Würfel 1: [[999, 998], [999, 998], [998, 2], [1000, 2], [1000, 2]]

kaese3.txt:

Output:

Würfel 1: [[995, 992], [995, 992], [995, 2], [993, 2], [996, 2], [994, 2], [997, 2], [997, 995], [997, 995], [995, 4], [998, 4], [998, 4], [997, 4], [999, 4], [999, 998], [998, 5], [1000, 998], [1000, 6], [1000, 999], [1000, 7], [1000, 1000], [1000, 1000], [1000, 1000]]

5.5.2 Vermischte Scheibenlisten

kaese1.txt + kaese2.txt:

Input:

[[4, 6], [6, 6], [998, 999], [3, 3], [3, 6], [2, 4], [3, 4], [4, 6], [6, 6], [2, 4], [2, 1000], [2, 1000], [2, 998], [4, 6], [2, 4], [998, 999], [3, 3]]

Output:

Würfel 1: [[999, 998], [999, 998], [998, 2], [1000, 2], [1000, 2]]

Würfel 2: [[4, 2], [4, 2], [4, 2], [4, 3], [3, 3], [3, 3], [6, 3], [6, 4], [6, 4], [6, 4], [6, 6], [6, 6]]

kaese2.txt + kaese3.txt:
Input:
[[998, 1000], [998, 999], [4, 998], [1000, 1000], [1000, 1000], [2, 998], [998, 999], [992, 995], [2, 1000], [2, 997], [999, 1000], [2, 996], [995, 997], [7, 1000], [992, 995], [2, 1000], [998, 999], [5, 998], [4, 999], [995, 997], [4, 997], [1000, 1000], [4, 995], [2, 994], [2, 993], [2, 995], [6, 1000], [4, 998]]
Output:
Würfel 1: [[999, 998], [999, 998], [998, 2], [1000, 2], [1000, 2], [1000, 1000]]
Würfel 2: [[995, 992], [995, 992], [995, 2], [993, 2], [996, 2], [994, 2], [997, 2], [997, 995], [997, 995], [995, 4], [998, 4], [998, 4], [997, 4], [999, 4], [999, 998], [998, 5], [1000, 998], [1000, 6], [1000, 999], [1000, 7], [1000, 1000], [1000, 1000]]

kaese1.txt + kaese2.txt + kaese3.txt:
Input:
[[2, 995], [5, 998], [992, 995], [2, 994], [4, 998], [1000, 1000], [992, 995], [6, 1000], [4, 997], [7, 1000], [3, 4], [2, 1000], [3, 6], [3, 3], [3, 3], [998, 999], [4, 6], [4, 6], [998, 999], [2, 1000], [6, 6], [1000, 1000], [4, 999], [2, 998], [999, 1000], [998, 1000], [4, 995], [2, 4], [2, 996], [2, 4], [995, 997], [995, 997], [2, 993], [2, 997], [4, 6], [1000, 1000], [998, 999], [6, 6], [4, 998], [2, 4]]
Output:
Würfel 1: [[995, 992], [995, 992], [995, 2], [993, 2], [996, 2], [994, 2], [997, 2], [997, 995], [997, 995], [995, 4], [998, 4], [998, 4], [997, 4], [999, 4], [999, 998], [998, 5], [1000, 998], [1000, 6], [1000, 999], [1000, 7], [1000, 1000]]
Würfel 2: [[999, 998], [999, 998], [998, 2], [1000, 2], [1000, 2], [1000, 1000], [1000, 1000]]
Würfel 3: [[4, 2], [4, 2], [4, 2], [4, 3], [3, 3], [3, 3], [6, 3], [6, 4], [6, 4], [6, 4], [6, 6], [6, 6]]

5.5.3 Unvollständige Scheibenlisten

kaese1.txt:
Input:
[[6, 6], [4, 6], [4, 6], [3, 3], [4, 6], [3, 6], [6, 6], [2, 4], [3, 3], [3, 4], [2, 4]]
Removed: [[2, 4]]
Output:
Würfel 1: [[4, 2], [4, 2], [4, 2], [4, 3], [3, 3], [3, 3], [6, 3], [6, 4], [6, 4], [6, 4], [6, 6], [6, 6]]]

Hinzugefügte Scheiben: [[4, 2]]

kaese2.txt:
Input:
[[998, 999], [2, 998], [2, 1000]]
Removed: [[998, 999], [2, 1000]]
Output:
Würfel 1: [[999, 998], [999, 998], [998, 2], [1000, 2]]]

Hinzugefügte Scheiben: [[999, 998]]

=> Zusammensetzten zu einem alternativen Würfel ohne die Scheibe [2, 1000] möglich.

kaese3.txt:
Input:
[[2, 993], [1000, 1000], [992, 995], [995, 997], [2, 997], [6, 1000], [999, 1000], [4, 997], [992, 995],

[1000, 1000], [5, 998], [995, 997], [4, 999], [2, 995], [1000, 1000], [998, 1000], [2, 996], [2, 994], [4, 998], [7, 1000]]
Removed: [[4, 995], [998, 999], [4, 998]]
Output:
Würfel 1: [[995, 992], [995, 992], [995, 2], [993, 2], [996, 2], [994, 2], [997, 2], [997, 995], [997, 995], [995, 4], [998, 4], [998, 4], [997, 4], [999, 4], [999, 998], [998, 5], [1000, 998], [1000, 6], [1000, 999], [1000, 7], [1000, 1000], [1000, 1000], [1000, 1000]]

Hinzugefügte Scheiben: [[995, 4], [998, 4], [999, 998]]

5.4.4 Unvollständige & gemischte Scheibenlisten

kaese1.txt + kaese2.txt:
Input:
[[2, 4], [3, 4], [4, 6], [998, 999], [4, 6], [6, 6], [2, 1000], [2, 998], [3, 6], [3, 3], [2, 1000], [6, 6], [4, 6], [3, 3]]
Removed: [[998, 999], [2, 4], [2, 4]]
Output:
Würfel 1: [[999, 998], [999, 998], [998, 2], [1000, 2], [1000, 2]]

Würfel 2: [[4, 2], [4, 1], [4, 3], [4, 3], [3, 3], [3, 3], [6, 3], [6, 4], [6, 4], [6, 4], [6, 6], [6, 6]]

Hinzugefügte Scheiben: [[4, 1], [4, 3], [999, 998]]

=> Zusammensetzten mit andern Scheiben zu einem Quader möglich (Scheiben [4, 1], [4, 3] statt [2, 4], [2, 4] für den Quader der Größe 6x6x6)

kaese2.txt + kaese3.txt:
Input:
[[995, 997], [2, 997], [6, 1000], [998, 1000], [998, 999], [992, 995], [4, 997], [1000, 1000], [4, 999], [4, 995], [2, 996], [999, 1000], [1000, 1000], [2, 1000], [2, 993], [4, 998], [2, 1000], [998, 999], [2, 994], [7, 1000]]
Removed: [[4, 998], [998, 999], [5, 998], [1000, 1000], [992, 995], [2, 995], [995, 997], [2, 998]]
Output:
Würfel 1: [[1000, 998], [1000, 998], [1000, 2], [1000, 2], [1000, 1000]]
Würfel 2: [[995, 992], [995, 992], [995, 2], [993, 2], [996, 2], [994, 2], [997, 2], [997, 995], [997, 995], [995, 4], [998, 4], [998, 4], [997, 4], [999, 4], [999, 998], [999, 998], [998, 6], [1000, 6], [1000, 999], [1000, 7], [1000, 1000]]
Hinzugefügte Scheiben: [[995, 992], [995, 2], [997, 995], [998, 4], [998, 6], [1000, 998]]

=> Kombiniert beide Käsequader für die optimalste Lösung (Zusammengesetzte Quader entsprechen nicht den eigentlichen Käsequadern aus kaese2 & 3)

kaese1.txt + kaese2.txt + kaese3.txt:
Input:
[[4, 6], [6, 6], [2, 995], [998, 999], [998, 999], [992, 995], [3, 4], [4, 999], [7, 1000], [2, 1000], [992, 995], [995, 997], [4, 998], [6, 6], [4, 6], [998, 999], [1000, 1000], [2, 1000], [2, 996], [2, 993], [4, 995], [995, 997], [2, 997], [2, 4], [4, 997], [6, 1000], [2, 998], [999, 1000], [4, 6], [1000, 1000]]
Removed: [[3, 3], [2, 4], [2, 994], [5, 998], [4, 998], [3, 6], [2, 4], [3, 3], [998, 1000], [1000, 1000]]

Output:
Würfel 1: [[995, 992], [995, 992], [995, 2], [993, 2], [996, 2], [994, 2], [997, 2], [997, 995], [997, 995], [995, 4], [998, 4], [998, 4], [997, 4], [999, 4], [999, 998], [999, 998], [998, 6], [1000, 6], [1000, 999], [1000, 7], [1000, 1000]]
Würfel 2: [[999, 998], [999, 998], [998, 2], [1000, 2], [1000, 2], [1000, 1000]]
Würfel 3: [[4, 4], [4, 4], [4, 2], [5, 4], [4, 3], [6, 4], [6, 4], [6, 4], [6, 6], [6, 6]]

Hinzugefügte Scheiben: [[4, 4], [4, 4], [5, 4], [999, 998], [994, 2], [998, 4], [998, 6]]