Bearbeitung: Christoph Barth
Teilname-ID: : 65809

**Bundeswettbewerb Informatik**
**Runde 2**

# Aufgabe 2 - Alles Käse

## Inhaltsverzeichnis

**1 Lösungsidee** **2**
1.1 Ablauf . . . 2
1.2 Datenstruktur . . . 3
1.3 Theoretische Komplexität . . . 4

**2 Umsetzung** **5**
2.1 `run` Methode . . . 6
2.2 `rebuild_cuboid_a` Methode . . . 6
2.3 Tatsächliche Komplexität . . . 7

**3 Beispiele** **8**

**4 Optimierung** **8**
4.1 Ineffizientes Zählen . . . 8
4.2 Vermeiden von Kopien . . . 8
4.3 Bestimmung der Startscheiben . . . 8

**5 Aufgabenteil B** **11**
5.1 Fehlende Scheiben . . . 12
5.2 Mehrere Dimensionen . . . 14
5.3 Käsepolyeder . . . 15
5.4 Nicht-uniforme Scheiben . . . 15
5.5 Willkürliches Schneiden . . . 15

**6 Fazit** **15**

**7 Anhang** **16**
7.1 Generation der Daten . . . 16
7.1.1 Generation von Beispielproblemen . . . 16
7.1.2 Erstellen von Probleminstanzen . . . 16
7.1.3 Lösen der Probleminstanzen . . . 17
7.2 Analyse der Daten . . . 17

**8 Quellcode** **19**

# Abstract

Die folgende Arbeit entwickelt in Kapiteln 1 und 2 einen effizienten Algorithmus, der die Aufgabenteil A in linearer Zeit löst. In Kapitel 4 wird der Algorithmus optimiert und da die eindeutige Wahl der Startscheibe nicht möglich ist, eine Reihe von heuristischen Verfahren entwickelt, die die optimale Lösung des Startscheibenproblems versuchen anzunähern. Mit dem kmeans*second-Verfahren wird eine effiziente Metaheuristik entworfen, das Startscheibenproblem Problem in den meisten Fällen nahezu optimal löst.

In Aufgabenteil Kapitel 16 werden verschiedene verwandte Probleme entwickelt. In Kapitel 5.1 wird eine Lösung teilweise implementiert. In den anderen Kapiteln wird eine Lösungsidee skizziert.

Aus sprachlichen Gründen werden im Folgenden oft die Wörter Quader statt Käsequader/-stück und Rechteck/Scheibe statt Käsescheibe verwendet. Da für die Bearbeitung der Aufgabe immer nur die drei unterschiedlichen Flächen des Quaders relevant sind, werden deren identische Gegenflächen vernachlässigt und oft nur von drei Flächen gesprochen.


# 1 Lösungsidee

In dieser Aufgabe soll ein vollständig in Scheiben zerschnittener Käsequader wieder zusammengesetzt werden.
Das Detail, welches zur Lösung dieser Aufgabe führt, liegt darin, dass die Quaderform des Quaders nach jedem Schneiden einer Scheibe erhalten bleibt. Das bedeutet umgekehrt, dass auch nach jedem Anfügen einer Scheibe an den Quader die Quaderform erhalten bleiben muss. Hieraus kann gefolgert werden, dass jede Scheibe, die dem Quader angefügt wird, exakt einer seiner Flächen entsprechen muss (siehe Abb. 1).

## 1.1 Ablauf

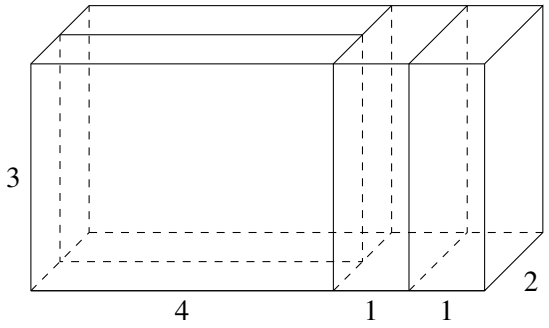



**Abbildung 1:** Quaderzusammensetzung aus $(3,4)(3,4)(3,2)(3,2)$

Bevor es zum Anfügen von Scheiben an den Quader kommen kann, muss ein initialer Quader aus zwei Startscheiben, dem Startscheibenpaar, gebildet werden. Auch hier gilt die oben aufgestellte Bedingung. Daraus folgt, dass die zwei Scheiben des Startscheibenpaars identisch sein müssen.
Im Fall, dass mehr als ein mögliches Startscheibenpaar existiert, kommt man per Intuition zu der Vermutung, das korrekte Startscheibenpaar müsse stets das kleinste sein. Abbildung 1 widerlegt dies allerdings.
Die Gleichheit ist somit die einzige konsistente Eigenschaft des Startscheibenpaars, weshalb alle Scheiben, die mindestens zweimal vorhanden sind, als potentielles Startscheibenpaar betrachtet und somit ausprobiert werden müssen.

Auf die Reihenfolge, in der diese ausprobiert werden, wird im Kapitel 4.3 eingegangen.

Der weitere Ablauf besteht daraus, so lange die drei Seiten des aktuellen Quaders zu bestimmen und diesem passend Folegescheiben anzufügen, bis keine weiteren Folgescheiben mehr gefunden werden können. Ist dann die Zahl der verbleibenden Scheiben 0, wurden alle Scheiben verwendet und der Quader erfolgreich zusammengesetzt. Ist dies nicht der Fall, muss der geschilderte Ablauf für ein anderes Startscheibenpaar wiederholt werden. Existiert irgenwann kein weiteres Startscheibenpaar mehr, so lässt sich der Quader nicht zusammensetzen.

## 1.2 Datenstruktur

Da ein großer Teil des Algorithmus auf dem Finden von passenden Folgescheiben in den verbleibenden Scheiben beruht, hat die Effizienz der Suche einen großen Einfluss auf die Komplexität des Algorithmus. Das ungeordnete Speichern der verbleibenden Scheiben in einer Liste ist somit ungeeignet, da das Suchen nach Elementen dann nur linear in $\mathcal{O}(n)$ Zeit möglich wäre. Um dies effizienter zu gestalten, muss eine Datenstruktur entworfen werden, die das Suchen effizienter macht. Drei Ansätze, die hierfür in Betracht gezogen wurden, werden am folgenden Beispiel erläutert.

$$[(4,3),(3,2),(2,6)]$$

Beim Entwerfen dieser Datenstruktur muss beachtet werden, dass der Käsequader beim Schneiden beliebig gedreht werden kann. Die Ausrichtung der Rechtecke, also die Koordinatenreihenfolge, verliert somit seine Relevanz. Das bedeutet, dass das Rechteck $(2,3)$ auch zur Suche nach $(3,2)$ passen muss.

1. Die verbleibenden Scheiben können als Tupel in einer sortierten Liste gespeichert werden. Als Sortierschlüssel eignet sich der Flächeninhalt $A = x * y$ der Rechtecke, da dieser aufgrund der Kommutativität der Multiplikation die Suche nicht auf eine Koordinatenreihenfolge beschränkt. In der sortierten Liste kann mit einer binären Suche in $\mathcal{O}(log\,n)$ Zeit nach Elementen anhand ihres Flächeninhaltes gesucht werden. Problem hierbei ist, dass verschiedene Elemente, denselben haben können. Unter diesen muss wieder mit einer linearen Suche in $\mathcal{O}(n)$ Zeit gesucht werden. Im schlechtesten Fall bietet diese Implementierung also keine oder nur wenig Verbesserung zur linearen Suche in einer unsortierten Liste.

$$[(3,2),(4,3),(2,6)]$$

2. Eine weitere Möglichkeit ist das Nutzen eines Dictionarys. Dies hat den Vorteil, dass in $\mathcal{O}(1)$ nach $k,v$ Paaren gesucht werden kann. Hier kann die jeweils kleinere Koordinate eines Rechtecks als Schlüssel $k$ genutzt werden, dessen assoziierter Wert $v$ eine Liste der zugehörigen größeren Koordinaten ist. Dies hat allerdings den Nachteil, dass innerhalb der Werte-Listen $v$ wieder nach dem größeren Wert gesucht werden muss. Sind die größeren Werte nicht sortiert, bietet diese Lösung im schlimmsten Fall also auch keine oder wenig Verbesserung gegenüber der unsortierten Liste. Würde man die Listen sortieren, könnte wieder binär in $\mathcal{O}(log\,n)$ Zeit gesucht werden. Die vielen Sortieroperationen beim Erstellen der Listen oder Einfügen von Elementen würden diese Effizienz aber wieder zunichte machen.

$$\{3:[4],2:[3,6]\}$$

3. Als dritte Möglichkeit können die x- und y-Koordinaten in einem Tupel als Schlüssel in einem Dictionary verwendet werden. Um die Suche nicht auf eine Koordinatenreihenfolge zu beschränken, kann die kleinere der Koordinaten im Schlüsseltupel vorangestellt werden. So gibt es eine einheitliche Vorschrift für die Koordinatenreihenfolge, weshalb es für die Suche nach einer Scheibe auch nur noch eine Möglichkeit gibt. Der Wert, der diesem Schlüssel zugeordnet wird, ist die eine Zählvariable, die zählt, wie oft die jeweilige Scheibe vorhanden ist. So kann jedes Element eindeutig und in $\mathcal{O}(1)$ Zeit gefunden werden. Auch das Einfügen und Entfernen von Scheiben ist effizient über das Manipulieren der Zählvariable möglich.

{(3,4): 1, (2,3): 1, (2,5):1}

Zusammenfassend ermöglicht die dritte Datenstruktur die effizienteste Suche und wird daher im Folgenden verwendet.

## 1.3 Theoretische Komplexität

Bei der aktuellen Lösungsidee muss beachtet werden, dass die Suche nach passenden Folgescheiben für einen Quader $Q_0$ bis zu drei Scheiben $S_1$, $S_2$ und $S_3$ zurückgeben kann. Würde man versuchen, den Quader mit jeder dieser Folgescheiben zusammenzusetzen, erhält man drei mögliche Folgequader $Q_{1;1}$, $Q1;2$ und $Q_{1;3}$. Im nächsten Schritt können für die Folgequader $Q_{1;1}$, $Q_{1;2}$ und $Q_{1;3}$ erneut jeweils bis zu drei Folgescheiben gefunden werden.

Die Komplexität für das Zusammensetzen des Quaders mit einer gegebenen Startscheibe beträgt somit im absolut schlimmsten Fall bis zu $\mathcal{O}(\frac{n*3^n}{2})$ (siehe Abb. 2a)[1] . Auch wenn dieser Worst-Case praktisch nicht zu erreichen ist, da der Faktor $3^n$ eigentlich durch $n$ limitert wird und die Basis in der Praxis sehr viel näher an 1 liegt, da im Schnitt weniger als drei Folgescheiben gefunden werden, sorgt das Ausprobieren aller Folgescheiben für große Einbußen an Effizienz.

Dies kann allerdings durch das eindeutige Bestimmen einer Folgescheibe vermieden werden. Um zu zeigen, dass dies möglich ist, wird der Quader mit den Seitenlängen $(x,y,z)$ betrachtet. Es existieren die Folgescheiben $(x,y)$ und $(y,z)$ mit $x > z$. Fügt man die Scheibe $(x,y)$ dem Quader an, erhält man einen neuen Quader mit den Seitenlängen $(x,y,z+1)$. Da $z+1 \neq z$ gilt, ist $(y,z)$ nun keine Folgescheibe mehr. Da die Seitenlängen des Quaders durch das Anfügen von Scheiben monoton steigen und $z < x$, existiert keine Möglichkeit, die Scheibe $(y,z)$ zu einem späteren Zeitpunkt anzufügen.
Fügt man hingegen die Scheibe $(y,z)$ an, erhält man einen neuen Quader $(x+1,y,z)$. Auch in diesem Fall ist $(x,y)$ keine Folgescheibe mehr, da $x+1 \neq x$. Es gilt allerdings $z < x$, weshalb $z$ durch weiteres Anfügen der Scheiben $(x+1,y)$ weiter steigen kann bis gilt $z = x$. Jetzt passt die Scheibe $(x,y)$ an die Fläche $(z,y)$ des Quaders.
Es folgt also, dass, stets die kleinste mögliche Folgescheibe verwendet werden muss. Hierdurch lässt sich die Komplexität auf $\mathcal{O}(n*1^n) = \mathcal{O}(n)$
verbessern (siehe Abb. 2b).
Ein weiterer Vorteil dieser Vorteil dieser Lösung ist, dass keine Technik wie Backtracking implementiert werden, die fehlerhafte Entscheidungen des Algorithmus rückgängig macht. Dies bringt zwar keine asym-

[1] Approximation der Fläche, die in Abbildung 2a, durch die skizzierten Entscheidungen eingenommen wird durch ein Dreieck mit Seitenlänge Höhe $n$ und Seitenlänge $3^n$

ptotische Verbesserung der Laufzeit, in der Praxis aber einen nützlichen konstanten Faktor und eine Optimierung der Raumkomplexität.

Da die Startscheibe noch zufällig gewählt wird, ist es möglich, dass der oben analysierte Ablauf für bis zu $\frac{n}{2}$ Startscheiben ausgeführt werden muss, weshalb die theoretische Gesamtkomplexität im Worst-Case bei $\mathcal{O}(n^2)$ liegt.

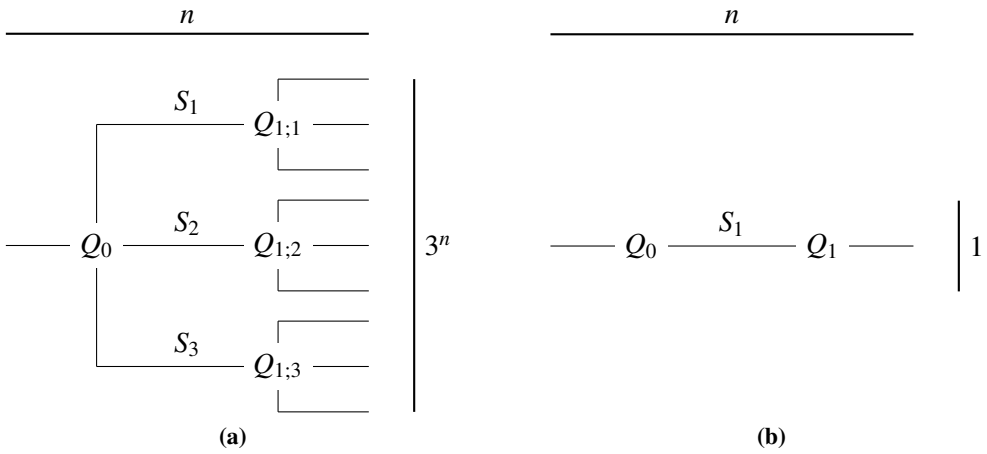



**Abbildung 2:** Menge an Entscheidungen im Worst-Case mit (a): zufälliger Wahl der Folgescheibe, (b): der kleinsten Folgescheibe

## 2 Umsetzung

Bevor der entworfene Algorithmus gestartet werden kann, müssen die Rechtecke eingelesen und in die entworfene Datenstruktur übersetzt werden. Die `get_data(n)` liest die Rechtecke als Tupel in eine Liste ein. Die `parse_data(data)` Methode nimmt diese Liste und wandelt sie in die beschriebene Datenstruktur um.

Hierfür wird die Subklasse `defaultdict` des regulären `dictionary` verwendet, da diese bei der Suche nach einem nicht vorhandenen Schlüssel einen Standardwert anstatt eines `KeyErrors` zurückgibt. Dieser Standardwert wird als `0` definiert, da ein Schlüssel/Scheibe, die nicht im `dictionary` enthalten ist, 0-mal existiert. Abb. 3 zeigt die `parse_data(data)` Methode.

```
1 def parse_data(data):
2     rects = defaultdict(lambda: 0)
3     for x, y in data:
4         rects[(x, y) if x < y else (y, x)] += 1
5
6     return rects
```

**Abbildung 3:** Code zur Erzeugung des Dictionarys

Die `parse_data(data)` Methode iteriert einmal über jedes Element der Eingabeliste und erhöht die An-

zahl jedes Elements um 1 im Dictionary `rects`. Da die Suche im Dicitonary in $\mathcal{O}(1)$ möglich ist, beträgt die Zeit- und Raumkomplexität dieser Methode $\mathcal{O}(n)$.

## 2.1 `run` Methode

Diese Daten werden nun als Eingabe für den Algorithmus verwendet. Dieser ist in der `run(rects)` Methode in main.py implementiert (Abhang Abb. 13). Zuerst ermittelt die Methode `get_start_rects(rects): list<'tuple'>` alle potentiellen Startscheiben, also alle Scheiben, die mindestens 2-mal vorhanden sind, und gibt sie in einer Liste zurück. Diese Methode ist in der Datei utils.py implementiert und wird wie die restlichen Methoden dieser Datei hier nicht näher erläutert.
Als Nächstes wird über jedes Element dieser Liste iteriert und mit der `rebuild_cuboid_a` Methode versucht, den Quader mit der ausgewählten Startscheibe zusammenzusetzen. Hierzu wird zunächst die Anzahl der aktuellen Startscheiben um 2 verringert und der aktuelle Quader als Tripel definiert. Der aktuelle Quader wird zusammen mit einer Kopie von `rects` als Parameter für die `rebuild_cuboid_a` Methode verwendet. Das Kopieren von `rects` ist notwendig, da die `rebuild_cuboid_a` Methode ihre Instanz von `rects` verändert.
Ist die Ausgabe von `rebuild_cuboid_a` nicht `None`, wurde der Quader erfolgreich zusammengesetzt. Der Algorithmus bricht ab und gibt den zusammengesetzten Quader zurück. Andernfalls fügt er das Startscheibenpaar wieder zu den Rechtecken hinzu und wiederholt diesen Ablauf in der nächsten Iteration mit einem anderen Startscheibenpaar.
Ist die Iteration über die Startscheiben beendet, konnte mit keiner der Startscheiben der Quader zusammengesetzt werden. Der Algorithmus schlägt fehl und gibt `False` zurück.

Die Methode `run(rects)` definiert die Variable `start_rects`, die in $\mathcal{O}(n)$ berechnet werden kann und maximal $\mathcal{O}(\frac{n}{2})$ Raum einnimmt. Zusätzlich ruft sie bis zu $\frac{n}{2}$ mal die `rebuild_cuboid_a` Methode auf. Hierbei wird eine Kopie aller Scheiben als Parameter verwendet, die logischerweise $\mathcal{O}(n)$ Raum beansprucht. Alle anderen Variablen haben eine konstante Raum- und Zeitkomplexität, weshalb sie vernachlässigt werden können.
Insgesamt liegt die Raumkomplexität hier also ungefähr bei $\mathcal{O}(n+n+x) \approx \mathcal{O}(n+x)$ [2] und die Zeitkomplexität bei $\mathcal{O}(n*x)$ mit $x$ als Komplexität der `rebuild_cuboid_a` Methode.

## 2.2 `rebuild_cuboid_a` Methode

Die `rebuild_cuboid_a` Methode ist für das Zusammensetzen des Quaders mit gegebenem Startquader zuständig. Sie nimmt den aktuellen Quader als Tripel in `current_cuboid` und die verbleibenden Rechtecke in `remaining_rects` als Parameter.
Bevor mit dem eigentlichem Zusammensetzen des Quaders begonnen wird, wird eine leere Liste `cuboid_list` definiert, in der die angefügten Rechtecke in Reihenfolge gespeichert werden.
Jetzt beginnt der Algorithmus in einer Endlosschleife mit der `get_next_rect` Methode Folgescheiben zu bestimmen, diese aus `remaining_rects` zu entfernen, dem Quader mit der `alter_cuboid` Methode anzufügen und der `cuboid_list` anzuhängen.
Die `get_next_rect` Methode bestimmt die drei unterschiedlichen Seiten von `current_cuboid` und gibt

[2] Die Raumkomplexität beträgt hier $\mathcal{O}(n+n+x)$ und nicht $\mathcal{O}(n*n*x)$, da der Speicherplatz $x$, der für der `rebuild_cuboid_a`-Methode und die Kopie der Scheiben reserviert wird, nach dem Ende der Methode wieder frei wird. So kann er von der nächsten Instanz wiederverwendet werden und der Speicherplatzbedarf addiert sich nicht.

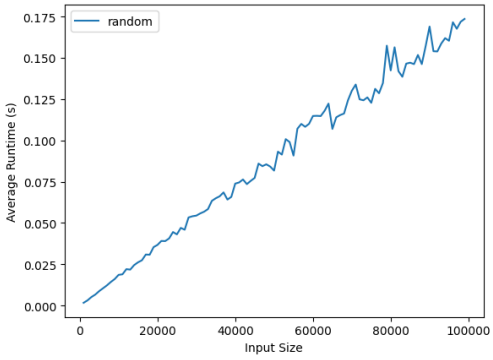
die kleinste, mit einer dieser Seite übereinstimmende Folgescheibe, die in `remaining_rects` enthalten ist, zurück. Sie ist zusammen mit der `alter_cuboid` Methode in utils.py implementiert.
Der Algorithmus wird beendet, wenn keine Folgescheibe mehr gefunden werden kann, die Rückgabe von `get_next_rect` also `None` ist. Ist dann die Zahl der verbleibenden Scheiben in `remaining_rects`, die mit `sum(remaining_rects.values())` bestimmt wird, gleich 0, so wurde der Quader erfolgreich zusammengesetzt. `cuboid_list` und `current_cuboid` werden zurückgegeben. Andernfalls wird für beide Werte `False` zurückgegeben.

Die Nutzung einer Endlosschleife ist hier unproblematisch, da maximal $n$ Folgescheiben gefunden werden können, weshalb die Schleife auch nur $n$ mal laufen kann, bevor sie durch ein `return` Statement beendet wird. Sie wird hier verwendet, da sie flexibler ist und sich Veränderungen in Hinsicht auf Aufgabenteil B leichter implementieren lassen.

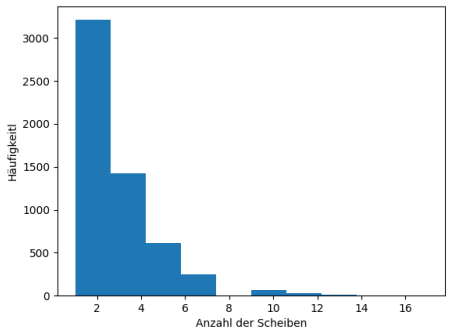
Die `rebuild_cuboid_a` Methode definiert eine neue Liste `cuboid_list`. Diese wird mit bis zu $n$ Elementen gefüllt. Die Raumkomplexität liegt also bei $\mathcal{O}(n)$. Das Füllen erfolgt in einer Schleife, die maximal $n$ mal läuft, weshalb die Zeitkomplexität hier ebenfalls $\mathcal{O}(n)$ beträgt. Diese Werte können für $x$ in die Komplexitäten des vorherigen Kapitels eingesetzt werden.

## 2.3 Tatsächliche Komplexität

Setzt man die für $x$ bestimmten Werte in die Komplexität aus Kapitel 2.1 ergibt sich eine Zeitkomplexität von $\mathcal{O}(n^2)$. Die in Kapitel 1.3 bestimmte theoretische Zeitkomplexität konnte also durch die Implementierung bestätigt werden. Messungen, die in Abb. 4a abgebildet sind, zeigen allerdings einen linearen Anstieg. Die Daten, die Abb. 4 zugrunde liegen, wurden mit dem Verfahren aus Kapitel 7.1 erstellt.




**(a)** Eingabegröße vs Laufzeit




**(b)** Anzahl der Folgescheiben für falsche Startscheibenpaare

**Abbildung 4:** Lineare Laufzeit des Algorithmus

Der Grund für das lineare Verhalten der Laufzeit ist, dass in der Regel nur wenige Folgescheiben existieren, die einem falschen Startscheinpaar angefügt werden können (Abb. 4b).
Eine quadratische Laufzeit liegt nur dann vor, wenn sich der Quader für fast jedes der bis zu $\frac{n}{2}$ Startscheiben nahezu vollständig zusammensetzen lässt. Da sich in den allermeisten Fällen nur maximal $3-4$ Folgescheiben für ein falsches Startscheibenpaar finden lassen, ist die Laufzeit der `rebuild_cuboid_a` Methode in diesen Fällen konstant. Die durchschnittliche Laufzeit beträgt also eher $\mathcal{O}((\frac{n}{2}-1)*1+1*n) \approx \mathcal{O}(n)$.

Die Raumkomplexität beläuft sich nach dem Einsetzen von $x$ auf $\mathcal{O}(n+n) \approx \mathcal{O}(n)$.

# 3 Beispiele

Der implementierte Algorithmus kann nun auf die Beispiele der BWInf-Seite angewandt werden. Es ergeben sich folgende Lösungen (die Lösungen zu Dateien $4-7$ liegen aufgrund ihrer Größe in den nach ihnen benannten Textdateien bei):

| | |
|---|---|
| kaese1.txt | [(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 | [(999, 998), (999, 998), (998, 2), (1000, 2), (1000, 2)] |
| kaese3.txt | [(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)] |
| kaese4.txt | Quader: $(210, 210, 210)$ |
| kaese5.txt | Quader: $(2310, 2730, 3570)$ |
| kaese6.txt | Quader: $(30030, 39270, 510510)$ |
| kaese7.txt | Quader: $(510510, 510510, 510510)$ |

# 4 Optimierung

Mit einer Laufzeit von $1:51$ Minuten für die Eingabe kaese7.txt läuft der Algorithmus noch inakzeptabel langsam. Um die Laufzeit des Programms zu verbessern, werden folgende Optimierungen vorgenommen.

## 4.1 Ineffizientes Zählen

Das wiederholte Zählen der verbleibenden Scheiben durch den Ausdruck `sum(remaining_rects.values())` ist sehr ineffizient, da jedes Mal in $\mathcal{O}(n)$ über alle Werte aus `remaining_rects` iteriert werden muss.
Eine Zählvariable `n` zu verwenden, die bei jedem Anfügen einer Scheibe um 1 verringert wird, ist hier ein effizienterer Ansatz.
Durch diese Verbesserung konnte die Laufzeit um 20% auf $1:29 min$ verbessert werden.

## 4.2 Vermeiden von Kopien

Eine Laufzeitanalyse ergibt, dass der Algorithmus 89% seiner Laufzeit damit verbringt, Kopien von `<collections.def` anzufertigen. Dies geschieht in der `run` Methode, da `rebuild_cuboid_a` Veränderungen an ihrer Instanz von `rects` vornimmt.
Das Kopieren von `rects` kann vermieden werden, wenn die `rebuild_cuboid_a` Methode ihre Instanz von `rects` im Falle eines Scheiterns wieder zurück in ihren Ausgangszustand bringt. Hierfür werden alle Elemente aus der `cuboid_list` entfernt und wieder `remaining_rects` hinzugefügt.
Durch diese Veränderung ist das Kopieren von `rects` obsolet geworden. Die Laufzeit konnte so um etwa 2000% auf $4.5 s$ gesenkt werden.

## 4.3 Bestimmung der Startscheiben

Weitere Analysen ergeben, dass der Algorithmus versucht, den Quader mit sehr vielen inkorrekten Startscheibenpaaren zusammenzusetzen, bevor die richtige Startscheibe gefunden wird. Das liegt daran, dass das Startscheibenpaar zufällig aus allen möglichen Startscheibenpaaren ausgewählt wird.

Eine Optimierung des Auswahlverfahrens des Startscheibenpaars würde die Laufzeit also weiter verbessern. Da es, wie in Kapitel 1.1 bereits gezeigt, nicht möglich ist, Scheiben kategorisch zum Beispiel aufgrund ih-

rer Größe etc. als Startscheibe auszuschließen, muss das neue Verfahren erneut auf dem Ausprobieren aller möglichen Startscheiben basieren. Was hingegen geändert werden kann, ist die Reihenfolge, in der die Startscheiben ausprobiert werden.
Um eine Eigenschaften des Startscheibenpaars zu finden, die sich für eine solche Priorisierung eignen, wurden viele mögliche Eingabedaten mit dem Verfahren, welches in Kapitel 7.1 beschrieben ist, erstellt und ausgewertet.
Abb. 5 präsentiert einige Ergebnisse der Analyse. Plots $1-4$ sind Histogramme, die die jeweils oben aufgeführte Eigenschaft eines möglichen Startscheibenpaares darstellen. Mit “First‘ bzw. “Second Tuple Param“ ist für ein mögliches Startscheibenpaar $(x,y)$ jeweils die $x$- bzw. $y$-Koordinate gemeint. Nach demselben Schema ist “Tuple Product“ das Produkt $x*y$, also die Fläche einer Startscheibe und “Tuple Difference“ die Differenz $y-x$. Rot eingezeichnet ist das korrekte Startscheibenpaar. Plots 5 und 6 werden später erläutert.

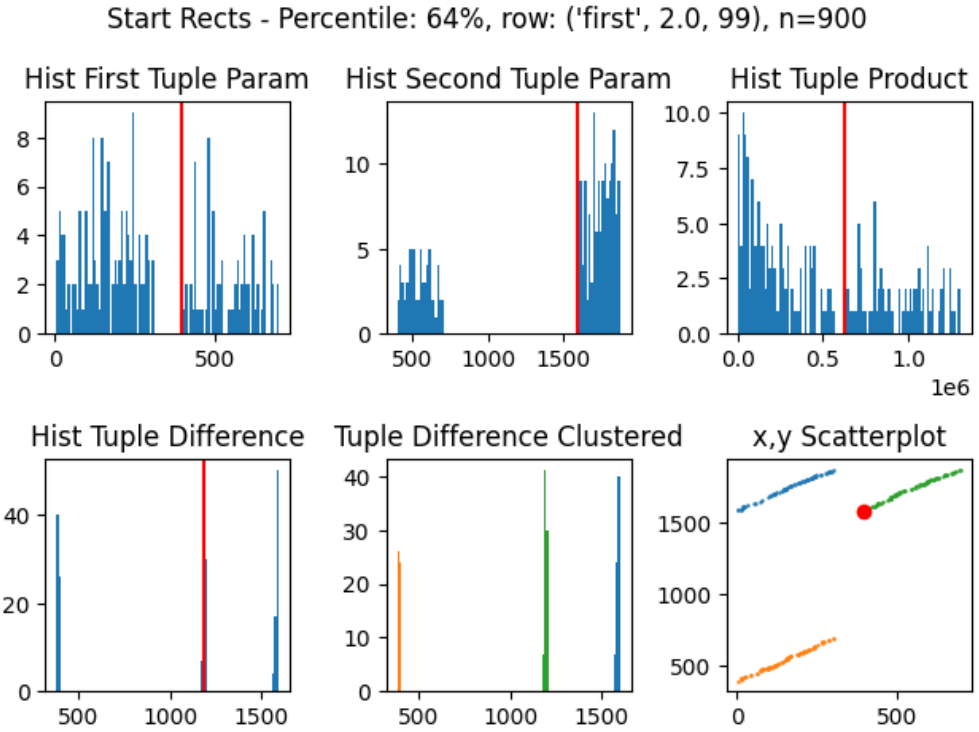



**Abbildung 5:** Analyse der Startscheiben

Aus Plots $1-3$ lässt sich keine direkte Korrelation zwischen den dargestellten Eigenschaften und der Position des korrekten Startscheibenpaars erkennen. Bei der Differenz der Koordinaten in Plot 4 fällt jedoch die Aufteilung der Startscheibenpaare in drei klar definierte Bereiche auf.

Plot 6 stellt mögliche Startscheiben als Punkte in einem $x,y$ Scatterplot dar. Es fällt auf, dass alle Punkte auf drei wohl definierten Geraden liegen. Hier befindet sich das korrekte Startscheibenpaar immer am Beginn einer dieser Geraden. Weiterhin fällt auf, dass sich die Peaks aus Plot 4 auf den y-Achsenabschnitt der drei Geraden übertragen lässt. Dies ist interessant, weil es so möglich wird, die Startscheibenpaare anhand ihrer Differenz in drei Gruppen zu Clustern (Plot 5) und diese so den Geraden zuzuordnen. Innerhalb dieser Cluster können die Startscheiben anhand ihres $x$-, $y$- oder $x*y$-Wertes sortiert werden, um so das erste Element zu bestimmen. In einer dieser drei Geraden ist das erste Element das korrekte Startscheibenpaar und kann

so schnell bestimmt werden.

Bevor dies implementiert und genutzt wird, ist es sinnvoll und interessant, zu untersuchen, warum diese Methode funktioniert und immer funktionieren muss:

Die Entstehung der drei Peaks in Plot 4 lässt sich auf die drei Seiten $S_1, S_2, S_3$ des initialen Quaders $Q_0$ am Beginn des Zusammensetzens zurückführen. Außerdem lassen sich die Seiten $S_1, S_2, S_3$ am Beginn jeder der drei Geraden in Plot 6 wiederfinden. Der Grund dafür, dass alle folgenden Scheiben, die $Q_0$ hinzugefügt werden eine Gerade mit den Startpunkten bilden, liegt im **zufälligen** Zerschneiden des Käsequaders.

Das zufällige Zerschneiden des Quaders bedeutet in umgekehrter Reihenfolge auch ein zufälliges Zusammensetzen. Die Wahrscheinlichkeit, dass eine Scheibe an eine Seite des Quaders angefügt wird, kann also jeweils als $\frac{1}{3}$ angenommen werden.

Idealisiert kann also angenommen werden, dass Scheiben in der Reihenfolge $S_1, S_2, S_3, S_1, S_2, S_3$ an die Quaderseiten angefügt werden. Nachdem an $S_1$ die Scheibe $(x,y)$ angefügt wurde, wird also im Schnitt einmal an $S_2$ und $S_3$ angefügt, bis wieder an $S_1$ angefügt wird. Dann gilt $S_1 = (x+1, y+1)$ und da $(y+1)-(x+1) = y-x$ hat die Scheibe, die an $S_1$ gesetzt wird, dieselbe Differenz, wie die vorherige. Dies gilt für alle Folgescheiben der Seite $S_1$, sowie an $S_2$ und $S_3$, wodurch sich die Peaks in Plot 4 erklären lassen.

Die Anordnung der Punkte in drei Geraden folgt daraus, dass die Scheiben, die einer Seite $(x,y)$ angefügt werden dem Muster $(x,y)(x+1,y+1)(x+2,y+2) = (x+i,y+i)$ folgen. Für jede Erhöhung der $x$-Koordinate eines Punktes wird also auch die $y$-Koordinate um 1 erhöht. Es entstehen Geraden mit der Steigung 1.

Natürlich handelt es sich bei diesen Annahmen um stochastische Spekulationen, die in der Regel niemals genau erreicht werden, für große $n$ aber angenähert werden.

So kann gefolgert werden, dass alle Scheiben den drei Geraden zugeordnet werden können, wobei die Position einer Scheibe in der Geraden Auskunft darüber gibt, zu welchem Zeitpunkt sie geschnitten wurde. Da das Startscheibenpaar zuletzt geschnitten wurde, muss es in einer der Geraden an erster Stelle stehen.

Für die Implementierung des Clusterns der Scheiben nach ihrer Differenz wird KMeans-Algorithmus aus scikit-learn genutzt, wobei $k = 3$ hier die Anzahl der Cluster angibt. Die Cluster werden dann nach $x$, $y$ oder $x * y$ Komponente sortiert, um die Position einer Scheibe in der Geraden zu ermitteln. Dann wird zunächst das erste Element jeder Geraden, dann das zweite Element und so weiter als Startscheibenpaar verwendet. Der so entstandene Sortieralgorithmus wird im Folgenden kmeans**schlüssel* genannt.

Neben diesem Algorithmus werden die Startscheiben der Form $(x,y)$ auch mit den Sortierschlüsseln $x :=$ first, $y :=$ second und $x * y :=$ area sortiert. Abbildung 6 stellt die Güte dieser Verfahren anhand der relativen Position des korrekten Startscheibenpaars in der sortierten Liste aller möglicher Startscheiben als Quantile in Violinplots dar.

Es lässt sich erkennen, dass die kmeans**schlüssel*-Verfahren das Startscheibenpaar deutlich schneller findet, als die anderen Verfahren. In sehr seltenen Fällen kommt es aber auch vor, dass diese Algorithmen das korrekte Startscheibenpaar nur im 20. Perzentil einordnen. Der Grund hierfür ist, dass sich in manchen Fällen zwei der drei Seiten des Quaders so stark ähneln, dass die Peaks von deren Seitenlängendifferenzen verschmelzen. In diesen Fällen kann der Algorithmus Scheiben keinem Peak mehr eindeutig zuordnen und

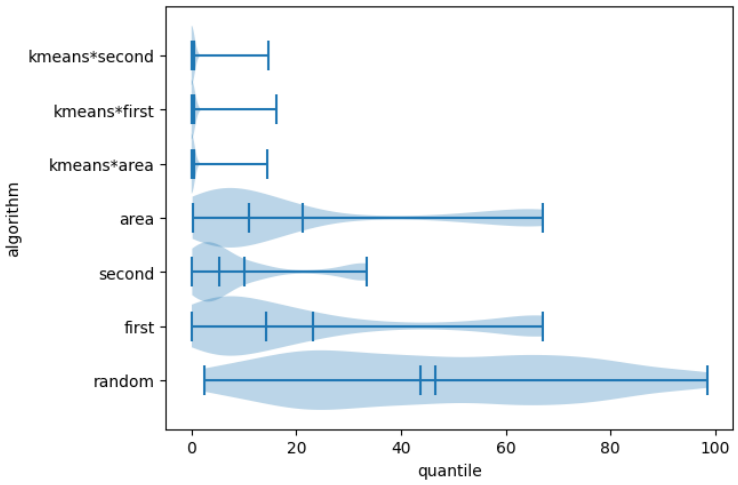



**Abbildung 6:** Quantil der korrekten Startscheibe mit verschiedenen Sortieralgorithmen

erzeugt nur mäßige Ergebnisse. Dieses Phänomen lässt sich zum Beispiel in kaese7.txt beobachten (Abb. 7).

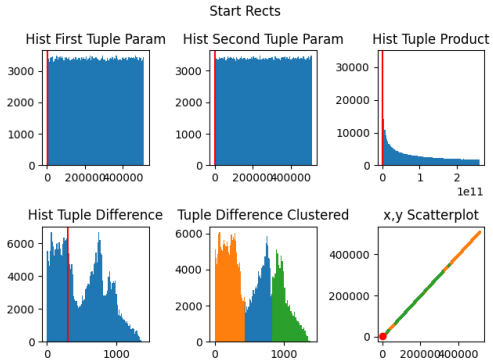
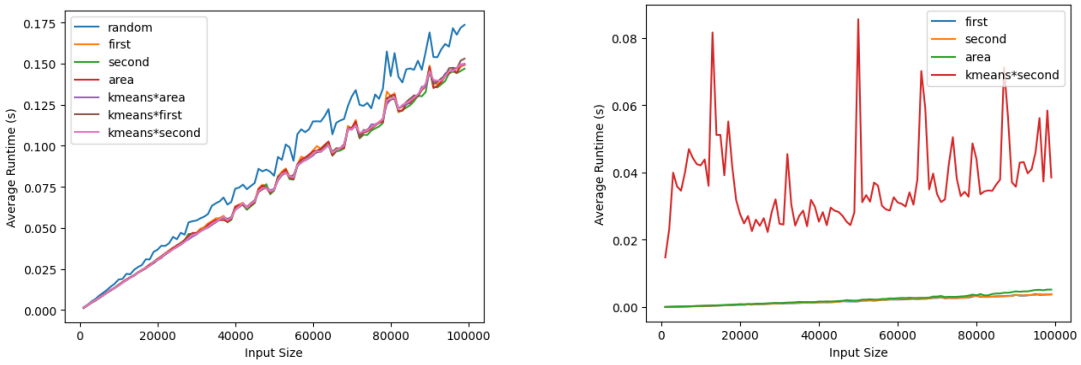



**Abbildung 7:** Analyse der Startscheiben von kaese7.txt

Bevor die kmeans**schlüssel* Algorithmen verwendet werden, sollte jedoch noch überprüft werden, ob das aufwendige Sortieren der Startscheiben die gewonnene Effizienz wert ist. Abb. 8a zeigt die Gesamtlaufzeit des Algorithmus mit den verschiedenen Sortierverfahren der Startscheiben und Abb. 8b die Zeit, die von ausgewählten Sortierverfahren benötigt wird. Es scheint, als würden alle entwickelten Sortierverfahren die Laufzeit, verglichen mit dem unsortierten Ansatz, um einen ähnlichen konstanten Faktor verbessern. Die Laufzeiten der zwei besten Algorithmen (second und kmeans*second) sind zum besseren Vergleich seperat in Abbildung 11 dargestellt. Hier ist praktisch kein Unterschied zu erkennen. Die Verbesserung, die das kmeans*second Verfahren über den Sortierschlüssel second bringt, scheint hier durch die Kosten des Verfahrens marginalisiert zu werden.
Für diesen Teil der Aufgabe ist der Sortierschlüssel second also die beste Wahl.

# 5 Aufgabenteil B

In Aufgabenteil B sollen allgmeinere Fragestellungen gefunden und gelöst werden. Hierfür wurde ein Beispiel gemacht, in dem Antje eine Käsescheibe gegessen hat und diese beim Zusammensetzen fehlt.




**(a)** **(b)**

**Abbildung 8:** Input-Größe vs (a): Laufzeit (b): Zeit zum Sortieren der Startscheiben

## 5.1 Fehlende Scheiben

Das Fehlen von Scheiben unter den Rechtecken kann zu verschiedenen Szenarien führen:

1. Der Quader lässt sich trotzdem zusammensetzen. In diesem Fall lässt sich der bereits erläuterte Lösungsweg anwenden.
2. Der Quader lässt sich ohne die fehlenden Scheiben nicht zusammensetzen

Das zuletzt beschriebene Szenario tritt ein, wenn keine weiteren Folgescheiben für einen Quader mehr gefunden werden können, obwohl noch Scheiben in `remaining_rects` verbleiben. In diesem Fall müssen fehlende Scheiben an geeigneter Stelle ergänzt werden. Auch für diese Scheiben gilt, dass sie mit einer Seite des Quaders übereinstimmen müssen und es keine kleinere Scheibe geben darf, die ebenfalls an den Quader passt. Wenn beispielsweise für den Quader $Q = (2,3,4)$ die Folgescheibe $(2,4)$ existiert, der Quader aber dennoch nicht zusammengesetzt werden kann, kann hier die Scheibe $(2,3)$ ergänzt werden.

Durch diesen Ansatz wird es möglich, dass für einen Quader mehrere Folgescheiben existieren, unter denen die eindeutige Auswahl nicht mehr möglich ist. Wenn es mit einer dieser Folgescheiben nicht möglich ist, den Quader zusammenszusetzen, muss die andere ausprobiert werden. Hierfür ist eine Technik wie Backtracking nötig. Für das Backtracking müssen Informationen über frühere Entscheidungen des Algorithmus gespeichert werden, um einen früheren Zustand wiederherzustellen und eine andere Entscheidung treffen zu können.

Der einfachste Weg, dies zu tun, wäre Kopien `remaining_rects`, `cuboid_list`, `current_cuboid` und eine Information über die Entscheidung auf einem Stack zu speichern. Jedes Mal, wenn der Algorithmus eine Scheibe anfügt, können diese Informationen auf den Stack gepusht werden. Durch das LIFO-Prinzip des Stacks kann durch `stack.pop()` immer der Zustand des Algorithmus bei der letzten Entscheidung wiederhergestellt werden. So ist es möglich das gewünschte Verhalten zu implementieren.

Die in Kapitel 4.2 herausgestellten Nachteile des Kopierens treffen hier immer noch zu. Deshalb wird auch hier das Kopieren dieser Elemente vermieden und erneut auf das Entfernen der zuletzt hinzugefügten Scheibe gesetzt, um einen früheren Zustand des Algorithmus wiederherzustellen.
Damit sich der Algorithmus nach dem backtracken nicht erneut für Folgescheiben entscheidet, die er bereits

```python
if start_idx > 0 or next_rect == None:
    if additions < max_additions and start_idx < 3:
        new_choices = generate_choices(next_rect, current_cuboid)
        if len(new_choices) > start_idx - 1:
            next_rect = new_choices[start_idx - 1]
            additions += 1
    else:
        next_rect = None
```

**Abbildung 9:** Code zur Generation fehlender Scheiben

ausprobiert hat, muss immernoch eine Information über die früheren Entscheidungen des Algorithmus auf dem Stack gespeichert werden. Hierfür bietet sich eine Art Index an: Wenn der Algorithmus die Scheibe nutzt, die von der `get_next_rect` Methode zurückgegeben wird, nutzt er die erste Scheibe. Dies wird mit einem `start_idx=1` kodiert. Wird an einem Punkt zu dieser Entscheidung "backgetrackt", wird diese `1` vom Stack `backtracking_list` in den `start_idx` geschrieben. So weiß der Algorithmus, dass die Folgescheibe, die von `get_next_rect` geliefert wird, bereits ausprobiert wurde und somit nicht mehr betrachtet werden darf. Jetzt werden alle möglichen Folgescheiben mit der Methode `generate_choices` generiert und die erste, noch nicht ausprobierte Scheibe als Folgescheibe verwendet. Auch diese Entscheidung wird als `2` in der `backtracking_list` gespeichert und zu einem späteren Zeitpunkt eventuell wieder in den `start_idx` geladen.
Hierbei ist es wichtig, Überblick über die Zahl der bereits generierten Scheiben zu halten, damit nicht unendlich viele Folgescheiben generiert werden. Dies geschieht mit der Variable `adiitions` und der Obergrenze für die generierten Scheiben `max_additions`. Im Sinne der vorgeschlagenen Verallgemeinerung ist `max_additions` auf 1 zu setzen, der Algorithmus funktioniert aber auch für jede andere Zahl.
Für jede generierte Scheibe wird `additions` um 1 erhöht. Wird beim Backtracking eine generierte Scheibe wieder entfernt, wird `additions` wieder um 1 verringert. Eine generierte Scheibe kann an einem negativen Wert in `remaining_rects` erkannt werden.

Da die Zahl der Folgescheiben, die für falsche Startscheibenpaare gefunden werden, hier durch die Generation von nicht-vorhandenen Scheiben massiv erhöht wird, gewinnt das schnelle Bestimmen der richtigen Startscheibe an Wichtigkeit. Deshalb ist es wichtig, hier das Auswahlverfahren kmeans*second zu verwenden.

Abbildung 9 zeigt den Teil des Codes, der für die Generation der neuen Scheiben verantwortlich ist. Der gesamte Algorithmus ist in Abb. 16 zu sehen.

Auch, wenn der Code theoretisch für beliebig große `max_additions` funktioniert, sollte Antje doch darauf achten, dass sie nicht zu viele Käsescheiben isst, da die Komplexität kubisch mit der Zahl der gegessenen Scheiben wächst. Wie es hierzu kommt, wurde bereits in Abb. 2a abgebildet und in Kapitel 1.3 erläutert. Der einzige Unterschied ist, dass es hier keine absolute Ausnahme ist, dass drei Folgescheiben gefunden werden, sondern die Regel ist. Die Komplexität beträgt somit $\mathcal{O}(n * k^3)$ mit $k =$ `max_additions`.

In den allermeisten Fällen funktioniert dieser Algorithmus und ist in der Lage, einen Quader selbst mit gegessenen Scheiben wieder zusammenzusetzen. Der Algorithmus ist hingegen nicht in der Lage den Quader zusammenzusetzen, wenn die gegessene Scheibe Teil des Startscheibenpaars ist.

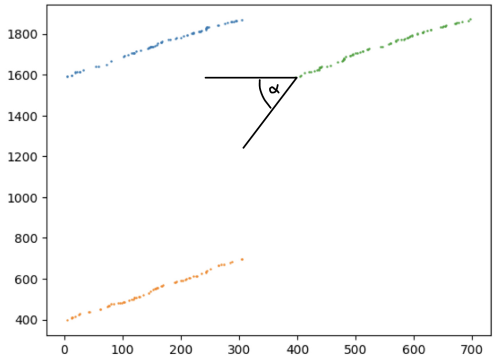



**Abbildung 10:** Bereich in dem mögliche Startscheiben liegen könnten

Um den Quader auch in diesem Sonderfall zusamensetzen zu können, müssen alle Scheiben als mögliches Startscheibenpaar angesehen werden. Da die Zahl der Startscheiben so vervielfacht wird, kommt es mit dieser Erweiterung noch mehr auf das Schnelle Finden der Startscheibe an, wodurch die Wichtigkeit des kmeans*second-Verfahrens hier noch einmal wächst.

Für den Fall, dass beide Startscheiben gegessen wurden, müsste eine gänzlich neue Scheibe Scheibe mit freien Dimensionen generiert werden. Hierfür wird nur noch die Lösungsidee beschrieben.
Ein Ansatz für das Finden einer solchen passenden Startscheibe ist der in Kapitel 4.3 erläuterte Fakt, dass die Startscheiben auf den drei Geraden aus Abb. 5 liegen. Diese haben eine Steigung von $\approx 1$. Punkte, die noch vor Beginn der Gerade liegen sind gute Kandidaten für eine Startscheibe und können so über eine Extrapolation der Geraden gefunden werden.
Da die Steigung von 1, wie bereits erläutert nur durch stochastische Vorgänge entsteht, kann die Position des korrekten Startscheibenpaars auch von den Geraden abweichen. Um hiermit umgehen zu können, müssen Punkte in einem größer werdenden Bereich um die Gerade betrachtet werden. Der Bereich ist in Abb. 10 skizziert. Da nie ausgeschlossen werden kann, dass sich ein Punkt auch außerhalb des eingezeichneten Bereichs befindet, kann der Winkel $\alpha$ verwendet werden, um die Wahrscheinlichkeit, das Startscheibenpaar zu finden, zu verändern. Bei $\alpha = 180°$ ist die Wahrscheinlichkeit 100%, eine Startscheibe zu finden, falls sie existiert.

## 5.2 Mehrere Dimensionen

Sollte Antje einmal in ein Paralleluniversum ziehen, in dem es mehr als drei Raumdimensionen gibt, steigen auch die Dimensionen der Käsescheiben an. Eine Scheibe kann dann nicht mehr durch ein $(x_1, x_2)$ Tupel gegeben sein, sondern durch ein $(x_1, x_2, \ldots, x_n)$ Tupel. Da das entwickelte Programm recht modular ist, wäre es relativ einfach auch dieses Szenario zu unterstützten.
Nur die Hilfsmethoden aus `utils.py` interagieren direkt mit den Scheiben und sind dafür programmiert, für zweidimensionale Scheiben effizient zu sein. Das bedeutet, dass anstatt der `sort()` -Methode zwei `if` -Abfragen verwendet wurden, um eine Scheibe nach der Größe ihrer Koordinaten zu sortieren, um eine bessere Effizienz zu erreichen.
Durch das Ändern solcher Details könnten auch höherdimensionale Quader zusammengesetzt werden.

### 5.3 Käsepolyeder

In einem anderen Szenario könnte Antje einmal einen ausgefalleneren Käse kaufen, der die Form eines 5 regulären-Polyeders hat. Ein regulärer Polyeder sind spezielle Polyeder, bei denen jede Seite dasselbe regelmäßiges Polygon ist. Dann kann wieder von jeder Seite geschnitten werden, unter der Bedingung, dass eine reguläre-polyederähnliche Form nach jedem Schnitt erhalten bleibt. Mit polyeder-ähnliche Form ist gemeint, dass die Zahl der Seiten des Polyeders nach jedem Schnitt bestehen bleiben muss, da es möglich ist, durch wiederholtes Schneiden an einer Seite den Polyeder komplett zu verändern. Das Zerschneiden wäre hier aber nicht beendet, wenn der Käsepolyeder vollständig in Scheiben zerschnitten ist, sondern wenn es nicht mehr möglich ist, eine Scheibe zu schneiden ohne eine Seite des Polyeders zu entfernen.
Ein Algorithmus für dieses Problem könnte sich hier stark an den Prinzipien aus Aufgabenteil A orientieren. Eine Umformung der bereits entwickelten Algorithmen ist allerdings schwierig, das sich auf die gänzlich neue Geometrie eingestellt werden muss, die ein anderes Vorgehen fordert.

### 5.4 Nicht-uniforme Scheiben

Sollte Antje neben Ihrem Gespräch noch an einem Glas Wein nippen, kann es sein, dass ihre Schnitte an Genauigkeit verlieren, und die Dicke der Scheiben variiert. In diesem Fall, müssten die Scheiben als Quader repräsentiert werden, da 1 nicht mehr als Tiefe angenommen werden kann. Um dies zu unterstützen, muss der Algorithmus die Seiten jeder Quaderscheibe bestimmen und prüfen, ob eine dieser Seiten an den Quader passt, der zusammengesetzt wird. Die algorithmischen Prinzipien beim Zusammensetzen werden nicht verändert. Die Wahl eines Startscheibenpaars ändert sich aber in die Wahl eines Startquaders.
Auch diese Änderung könnte durch Änderungen an den Hilfsmethoden und einige kleinere Änderungen am Grundablauf implementiert werden.

### 5.5 Willkürliches Schneiden

Ein Szenario, welches von dem entwickelten Algorithmus und keiner seiner leicht veränderten Formen gelöst werden kann ist, wenn Antje die Scheiben willkürlich schneidet. Das heißt, schräg, diagonal, quer oder den Käseklumpen ohne jegliche Beschränkungen um beliebige Winkel dreht. So ist nicht mehr garantiert, dass ein Schnitt ein Rechteck erzeugt. Durch zufällige vorhergegangene Schnitte könnten alle möglichen Polygone geschnitten werden. Auch das Käsestück würde seine Quaderform verlieren und eine Kugel annähern.

Nach längerem Überlegen, fällt mir kein Ansatz ein, dieses Problem vernünftig modellieren, geschweigedenn lösen zu können.

## 6 Fazit

Die vorhergegangene Bearbeitung fokussiert sich auf Aufgabenteil A. Ich habe viel Zeit in die Entwicklung der kmeans-Verfahren investiert und hierfür effiziente Methoden zum Testen der verschiedenen Algorithmen entwickelt. Dies umfasst, ist jedoch nicht beschränkt auf die Generation von vielen Probleminstanzen bei deren Lösung verschiedene Metriken gemessen und in großen Datensammlungen gespeichert wurden. Diese Daten wurden mithilfe von `pandas` organisiert, bereinigt und ausgewertet, um Laufzeit und Güte der verschiedenen Algorithmenvarianten zu beurteilen. Dieses Vorgehen weicht teils stark von der originalen Aufgabenstellung ab, wurde aufgrund von persönlichem Interesse in Data-Science aber dennoch weiter verfolgt und deshalb näher im Anhang beschrieben.

Aufgabenteil B umfasst die Lösung des Problems des Problems für fehlende Scheiben und skizziert kurz

andere Varianten des Problems.

Es gibt jedoch eine Methode zum wiederherstellen des Käsequaders, die auf jede der in B beschrieben Problemvariationen in konstanter bis linearer Zeit anwendbar ist: Antje könnte alle Käsescheiben in einem Topf einschmelzen, diesen optimal auskratzen und den flüssigen Käse in eine Kastenform füllen. Nach dem Abkühlen hat sie einen Käsequader zusammengesetzt und ihr Problem so unkompliziert gelöst.

# 7 Anhang

## 7.1 Generation der Daten

Für die Generation der Daten mussten zunächst Probleminstanzen erschaffen werden.

### 7.1.1 Generation von Beispielproblemen

Die Methode `get_sample_data(n, max_start_dim, missing_rects)` generiert die Probleminstanzen. Hierfür nimmt sie `n`, `max_start_dim` und `missing_rects` als Parameter. `n` meint hier die Zahl von Scheiben, die das Problem enthalten soll, `max_start_dim` gibt einen Wert für die maximale Größe der Startscheiben an und `missing_rects` entfernt zufällig Scheiben, sodass die Probleminstanz für Aufgabenteil B genutzt werden kann.
Die Methode generiert ein zufälliges Startscheibenpaar und setzt daraus einen initialen Quader zusammen. Dann wird $n-2$ mal eine Scheibe generiert, die an eine zufällige Seite des Quaders passt. Die verwendeten Scheiben werden in einer Liste gespeichert, die dann zusammen mit einer Information über eventuell fehlende Scheiben zurückgegeben wird.
Die Rückgabe hat dasselbe Format, wie wenn die Scheiben normal aus einer der Beispieldateien eingelesen worden wären und kann deshalb genauso behandelt werden.

### 7.1.2 Erstellen von Probleminstanzen

Als nächstes müssen viele Probleminstanzen mit variierender Problemgröße $n$ generiert werden. Hierfür wird eine maximale Problemgröße `max_input_size` und eine Variable `step` festgelegt. `step` gibt hierbei an in welchen Schritten die Problemgröße ansteigen soll.

`max_input_size = 10` und `step = 2` steht also für Eingabegrößen `[2,4,6,8,10]`. Um stochastischen Schwankungen entgegenzuwirken, werden mehrere Probleminstanzen für jede Eingabegröße generiert. Die Information hierüber ist in `runs` gespeichert. Außerdem kann die Größe der Startscheibe relativ zur Eingabegröße variieren. Hierzu wird eine Werteliste `dim_vs_n` angelegt. Außerdem soll jede Eingabe mit verschiedenen Verfahren zur Sortierung der Startscheiben gelöst werden. Hierzu wird eine Liste `algos` verwendet, die die Namen der verschiedenen Verfahren hält.
Die Liste `metrics` hält die Metriken, die für jedes Ausführen des Algorithmus gemessen werden sollen. Die folgende Tabelle stellt die verfügbaren Metriken kurz vor:

| | |
|---|---|
| `runtime` | Misst Gesamtlaufzeit des Algorithmus |
| `start_rect` | Gibt Korrekte Startscheibe zurück |
| `start_rect_count` | Gibt Zahl aller möglichen Startscheiben zurück |
| `start_rect_position` | Gibt Position der korrekten Startscheiben in der nach dem ausgewählten Algorithmus sortierten Liste von Startscheiben zurück |
| `start_rects` | Gibt Liste aller Startscheiben zurück |
| `return_cuboid` | Gibt zusammengesetzten Quader zurück |
| `return_solution` | Gibt Liste mit Reihenfolge der Scheiben für die Quderzusammensetzung zurück |
| `sorting_time` | Gibt Zeit zurück, die zum Sortieren der Startscheiben benötigt wurde |

### 7.1.3 Lösen der Probleminstanzen

Die vorgestellten Parameter werden in einem `dictionary` `params` gespeichert. `params` wird dann als Paramter an die `batch_process` Methode übergeben. Diese generiert mit der zuvor vorgestellten Methode die benötigten Probleminstanzen. Da es sich hier schnell um mehrere hunderttausende Probleminstanzen handeln kann, werden die Probleminstanzen in `batches` aufgeteilt und von mehreren Prozessen gleichzeitig gelöst. Hierbei wird die Größe der Probleminstanzen berücksichtigt, sodass jeder "batch" ungefähr gleich lange benötigt und nicht ein "batch" 100 Probleminstanzen mit 100 Scheiben enthält und ein anderer 100 mit 10000.

Jede Probleminstanz gibt die in `metrics` spezifizierten Messwerte zurück, die dann in ein `pandas` Dataframe mit den Indizes `input_size`, `dim_vs_n`, `algorithm`, `run` zurückgibt.

Große Datensätze sollten als Datei gespeichert werden und zur späteren Benutzung mit der `pd.read_csv()` Methode wieder geöffnet werden.

## 7.2 Analyse der Daten

Die Analyse der Daten findet im Jupyter-Notebook performance_evaluation.ipynb statt. Hierfür wird zunächst das gewünschte `pandas` `dataframe` aus dem Speicher geladen, korrekt indexiert und aus der Position der korrekten Startscheibe und der Anzahl von Startscheiben das Quantil berechnet, an dem sich die korrekte Startscheibe letztendlich befunden hat.

Die nächste Zelle entfernt alle Werte der Laufzeit und Sortierzeit, die mehr als 3 Standardabweichungen vom Mittelwert ihrer jeweiligen Probleminstanzklasse abhängen. Zu einer Probleminstanzklasse gehören alle Datensätze, deren Index sich nur um den `run` unterscheidet. Diese Unterteilung ist sinnvoll, da innerhalb der Problemistanzklasse alle Datensätze die gleiche Problemgröße, denselben Algorithmus und denselben `dim_vs_n` Wert haben. Nur innerhalb einer Probleminstanzklasse kann von einer gleichen oder ähnlichen Laufzeit ausgegangen werden.
So können Ausreißer, die teilweise merhere Minuten zum Lösen gebraucht haben, während der Rest ihrer Probleminstanzgruppe in Sekundenbruchteilen gelöst wurden gefiltert werden. Die Ausreißer entstehen durch Schwankungen der Prozessorleistung. Da ich in meiner Simulation nahezu alle Kerne und Threads des Prozessors gleichzeitig genutzt habe, musste das Betriebssystem zwischenzeitlich Threads, die eigentlich die Simulation berechnet haben, für andere Prozesse allokieren. Dadurch stieg die Laufzeit auf ein

vielfaches an, was eine Verfälschung der Messdaten mit sich bringt, die hier gefiltert wird.[3]

Die nächste Zelle stellt die Gesamtlaufzeit mit verschiedenen Algorithmen gegen die Eingabegröße dar. `blacklist` und `whitelist` können genutzt werden, um anzupassen, welche Algorithmen angezeigt werden.

Die nächste Zelle generiert einen Violinplot, in dem die relative Position der Startscheibe für verschiedene Algorithmen dargestellt wird. `blacklist` und `whitelist` können genutzt werden, um anzupassen, welche Algorithmen angezeigt werden.

Die nächste Zelle generiert den bereits verwendeten Plot, in dem die Startscheiben analysiert werden. Dies funktioniert immer nur mit einem einzelnen Datensatz, der in Zeile 2 ausgewählt wird.

Die nächste Zelle trägt die Zeit, die zum Sortieren benötigt wurde, gegen die Eingabegröße auf. `blacklist` und `whitelist` können genutzt werden, um anzupassen, welche Algorithmen angezeigt werden.

Die nächste Zelle erstellt einen $x,y$-Scatterplot von allen Startscheiben, der schon im vorletzten Plot verwendet wird. Dies funktioniert immer nur mit einem einzelnen Datensatz, der in Zeile 2 ausgewählt wird.

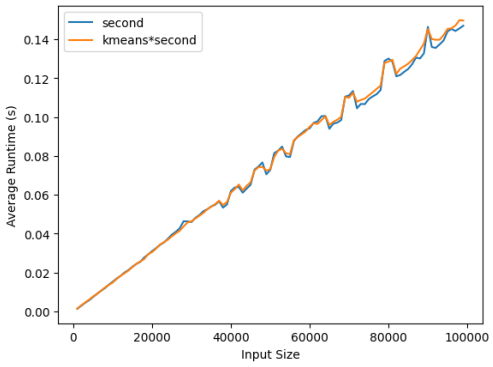



**Abbildung 11:** Laufzeiten der zwei besten Algorithmen

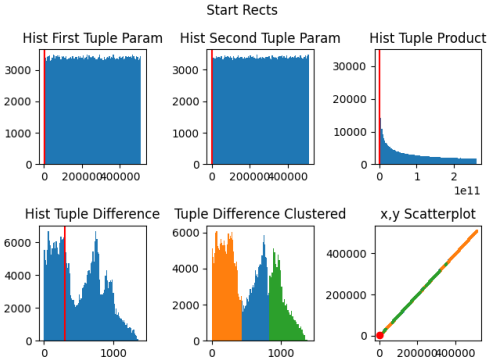



**Abbildung 12:** Startscheibenanalyse von kaese7.txt

[3] Eine Stichprobe der Ausreißer wurde im Nachhinein als normale, einzelne Eingabe für den Algorithmus verwendet und brachte wieder normale Ergebnisse.

# 8 Quellcode

Da das Latex-Package, welches ich zur Darstellung des Quellcodes nutze, nicht gut mit Kommentaren umgehen kann, ist der Code der Folgenden Abbildungen unkommentiert. Kommentierte Versionen sind in den zugehörigen Python-Dateien zu finden.

```
def run(rects):
    start_rects = sorted(get_start_rects(rects), key=lambda x:x[1])
    n = sum(rects.values())-2

    for start_rect in start_rects:

        rects[start_rect] -= 2
        current_cuboid = start_rect + (2,)
        cuboid_list, cuboid = rebuild_cuboid_a(current_cuboid, rects, n)

        if not cuboid_list == (False):
            cuboid_list = [start_rect, start_rect] + cuboid_list
            print('rebuilt cuboid!')
            return cuboid_list
        else:
            rects[start_rect] += 2
    print("Quader laesst sich nicht zusammensetzen")
    return False
```

**Abbildung 13:** `run` Methode

```python
def rebuild_cuboid_a(current_cuboid, remaining_rects, n):
    cuboid_list = []
    current_cuboid = sorted(current_cuboid)

    while True:

        next_rect = get_next_rect(remaining_rects, current_cuboid)

        if next_rect is not None:
            remaining_rects[next_rect] -= 1
            n -= 1

            current_cuboid = alter_cuboid(current_cuboid, next_rect, 1)
            cuboid_list.append(next_rect)
        else:
            if n == 0:
                return cuboid_list, current_cuboid
            else:
                while len(cuboid_list) > 0:
                    last_rect = cuboid_list.pop()
                    current_cuboid = alter_cuboid(current_cuboid, last_rect, -1)
                    remaining_rects[last_rect] += 1
                    n += 1
                return False, False
```

**Abbildung 14:** `rebuild_cuboid_a`-Methode nach den Optimierungen aus Kapitel 4

```python
def alter_cuboid(cuboid, rect, n):
    cuboid = list(cuboid)
    if rect == (cuboid[0], cuboid[1]) or rect == (cuboid[1], cuboid[0]):
        cuboid[2] += n
    elif rect == (cuboid[1], cuboid[2]) or rect == (cuboid[2], cuboid[1]):
        cuboid[0] += n
    elif rect == (cuboid[0], cuboid[2]) or rect == (cuboid[2], cuboid[0]):
        cuboid[1] += n
    else:
        print("bug found:", cuboid, "+", rect)

    return sorted(tuple(cuboid))

def get_cuboid_faces(cuboid):
    return [(rect[1], rect[0]) if rect[1]<rect[0] else rect for rect in list(
        itertools.combinations(cuboid, 2))]

def get_next_rect(rects, cuboid):

    out = None

    for rect in get_cuboid_faces(cuboid):
        rect = (rect[1], rect[0]) if rect[1]<rect[0] else rect
        if rects[rect] >= 1  and (out == None or rect[0]*rect[1]<out[0]*out[1]):
            out = rect

    return out
```

**Abbildung 15:** Auszug aus utils.py

```python
def rebuild_cuboid_b(current_cuboid, remaining_rects, n, max_additions = 0):
    cuboid_list = []
    current_cuboid = sorted(current_cuboid)

    backtracking_list = []
    start_idx = 0
    additions = 0

    while True:

        next_rect = get_next_rect(remaining_rects, current_cuboid)

        if start_idx > 0 or next_rect == None:
            if additions < max_additions and start_idx < 3:
                new_choices = generate_choices(next_rect, current_cuboid)
                if len(new_choices) > start_idx - 1:
                    next_rect = new_choices[start_idx - 1]
                    additions += 1
            else:
                next_rect = None

        if next_rect is not None:
            remaining_rects[next_rect] -= 1
            n -= 1

            current_cuboid = alter_cuboid(current_cuboid, next_rect, 1)
            cuboid_list.append(next_rect)

            backtracking_list.append(start_idx + 1)
            start_idx = 0
        elif n + additions == 0:
            return cuboid_list, additions
        elif len(cuboid_list) > 0:
            last_rect = cuboid_list.pop()
            if remaining_rects[last_rect] < 0:
                additions -= 1
            start_idx = backtracking_list.pop()
            current_cuboid = alter_cuboid(current_cuboid, last_rect, -1)
            remaining_rects[last_rect] += 1
            n += 1
        else:
            return False, False
```

**Abbildung 16:** `rebuild_cuboid`-Methode für Aufgabenteil b