Aufgabe 2: Die goldene Mitte

Teilnahme-ID: 69408

Bearbeiter dieser Aufgabe:
Tim Krome

31. Oktober 2023

Lösungsidee

Gegeben sind eine leere Kiste mit den Seitenlängen (x, y, z) und mehrere Quader. Es soll eine mögliche Anordnung der gegebenen Quader in der Kiste gefunden werden, sodass die Quader die Kiste vollständig ausfüllen. Es dürfen dabei keine Plätze in der Kiste freibleiben und die Quader dürfen sich nicht überlappen. Zusätzlich zu den gegebenen Quadern soll ein goldener Würfel mit den Kantenlängen (1,1,1) genau in der Mitte der Kiste platziert werden. Es wird davon ausgegangen, dass die Kiste immer ungerade Kantenlängen hat, da nur dann ein goldener Würfel genau in der Mitte der Kiste platziert werden kann.

Eine Anordnung kann grundsätzlich gefunden werden, indem die Quader nacheinander in der Kiste platziert werden.
Vorüberlegung: In jedem Befüllungszustand[1] (außer im vollständig befüllten Zustand) hat die Kiste innere Ecken. Diese Ecken können als Plätze, an denen Quader platziert werden müssen, verstanden werden: Im vollständig gefüllten Zustand der Kiste gibt es keine inneren Ecken mehr, da kein freier

[1] Mit „Befüllungszustand“ ist ein Zustand der Kiste gemeint, in dem mindestens einer der gegebenen Quader in der Kiste platziert ist. Sind alle gegebenen Quader in der Kiste platziert, dann wird vom „vollständig befüllten Zustand“ gesprochen.

Platz mehr verfügbar ist. Es müssen also beim Befüllen der Kiste bzw. beim Ermitteln einer möglichen Quaderanordnung alle inneren Ecken eliminiert werden.
Eine innere Ecke kann nur eliminiert werden, indem in ihr ein Quader platziert wird. Anfangs beispielsweise existieren in der Kiste acht innere Ecken. Wenn ein Quader in einer dieser inneren Ecken platziert wird, verschwindet die entsprechende innere Ecke, und gleichzeitig entstehen drei neue innere Ecken. Wird ein Quader so platziert, dass er in mehreren inneren Ecken liegt, dann werden dadurch mehrere innere Ecken gleichzeitig eliminiert, die Gesamtanzahl an inneren Ecken kann durch das Platzieren eines Quaders also auch sinken. Ziel ist, dass alle inneren Ecken verschwinden.
Somit muss in einer korrekt teilweise gefüllten Kiste für jede beliebige innere Ecke mindestens ein noch nicht platzierter Quader existieren, der in dieser inneren Ecke platziert werden kann. Wenn es keinen solchen Quader gibt, gibt es keine Möglichkeit, die innere Ecke zu eliminieren, folglich wurde mindestens einer der bisher in der Kiste platzierten Quader an einer falschen Position platziert (oder die Kiste kann gar nicht vollständig gefüllt werden).

Zur systematischen Suche nach einer möglichen Anordnung der Quader kann so vorgegangen werden:

1. Zuerst wird überprüft, ob das Gesamtvolumen der gegebenen Quader (inkl. Goldener Würfel) dem Volumen der Kiste entspricht. Wenn dem nicht der Fall ist, kann es keine Lösung geben, da die Quader entweder nicht alle in die Kiste passen oder die Kiste nicht vollständig ausfüllen können.

2. Der goldene Würfel wird in der Mitte der leeren Kiste platziert.

3. Eine Liste namens *alle_kisten* wird initialisiert, die zum Speichern von teilweise gefüllten Kisten dienen wird. Die Liste enthält zunächst nur die aus Schritt 2 hervorgehende Kiste, die bis auf den goldenen Würfel leer ist.

4. Die folgenden Schritte werden wiederholt, bis die Liste *alle_kisten* eine vollständig gefüllte Kiste enthält oder die Liste *alle_kisten* leer ist:
    a) Aus *alle_kisten* wird das erste Element entnommen und entfernt. Die entnommene Kiste wird im Folgenden als *kiste* bezeichnet.
    b) Es wird **eine** beliebige innere Ecke von *kiste* ausgewählt.
    c) Es werden alle Möglichkeiten ermittelt, einen der noch nicht verbauten Quader in beliebiger Ausrichtung in der in b) ausgewählten inneren Ecke zu platzieren.
    Dies wird erreicht, indem alle verfügbaren Quader in allen möglichen Rotationen durchprobiert werden.
    Falls ein Quader an der ausgewählten Ecke in die Kiste passt, wird eine identische Kopie der Kiste erzeugt, in der der Quader in der inneren Ecke platziert ist. Diese erzeugte Kiste wird vorne in die Liste *alle_kisten* eingefügt.

Mit dem obigen Vorgehen werden alle Möglichkeiten durchprobiert, die inneren Ecken der anfangs leeren Kiste nacheinander durch Platzieren eines Quaders in der jeweiligen inneren Ecke zu eliminieren. Somit wird eine vollständige Befüllung der Kiste auf jeden Fall gefunden, wenn sie existiert. Wenn die Kiste nicht vollständig befüllt werden kann, dann ist die Liste *alle_kisten* nach einer endlichen Anzahl von durchgeführten Schritten leer, da irgendwann alle Möglichkeiten

durchprobiert wurden. Das Vorgehen terminiert also auf jeden Fall nach einer endlichen Anzahl von Schritten.

Laufzeitanalyse

Im folgenden Abschnitt wird die Laufzeit des von mir verwendeten Verfahrens (siehe Vorkapitel) bestimmt und mit der eines alternativen Verfahrens verglichen. Das alternative Verfahren ist dabei ein nicht systematischer Brute-Force-Ansatz, der deutlich ineffizienter als das in meiner Lösung verwendete Verfahren ist.

	Verwendetes Verfahren, dass sich an den inneren Ecken der Kiste orientiert (siehe Vorkapitel)	**Alternative zum verwendeten Verfahren: Tiefensuche, die alle möglichen Anordnungen ausprobiert**
Funktionsweise	Alle Möglichkeiten durchprobieren, die inneren Ecke der Kiste nacheinander durch Platzieren eines Quaders zu eliminieren.	Für jeden Quader alle möglichen Positionen (x,y,z) durchprobieren, bis eine Anordnung gefunden wird, die die Kiste vollständig füllt.
Laufzeit (Worst-Case-Szenario)	Siehe *Lösungsidee, 4.*: In jedem Schritt wird **eine** beliebige innere Ecke ausgewählt. Im ersten Schritt gibt es n Optionen, an der gewählten inneren Ecke einen Quader zu platzieren (Zur Vereinfachung der Laufzeitberechnung wurden hier die möglichen Rotationen der Quader nicht berücksichtigt), im zweiten Schritt gibt es n-1 Optionen usw. (*n* = Anzahl der Quader) Daher ist die Laufzeit n! Durch Optimierungen kann die Laufzeit allerdings sogar noch verbessert werden.	Es werden für jede Position alle Möglichkeiten probiert, einen Quader an dieser Position zu platzieren. Beim Platzieren des ersten Quaders gibt es *a*b*c* mögliche Positionen, an denen der Quader platziert werden kann. Beim Platzieren des zweiten Quaders gibt es a*b*c – d Optionen, den Quader zu platzieren (d = Konstante, die von den Seitenlängen des platzierten Quaders abhängt). Die Laufzeit des Verfahrens ist also hauptsächlich vom Volumen des Quaders bzw. von a*b*c abhängig und lässt sich als (a*b*c)! abschätzen.

Fazit: Das verwendete Verfahren (Laufzeit: n!) ist effizienter als das alternative Verfahren (Laufzeit (a*b*c)!), da in der Regel gilt: n < a*b*c. (Dieser Zusammenhang gilt nur dann nicht, wenn alle Quader die Seitenlängen (1,1,1) haben; dann gilt nämlich: n = a*b*c)

Optimierung: Mehrfach vorkommende Quader

In der Menge an gegebenen Quadern können Quader mehrfach vorkommen bzw. identisch sein. Dies kann beim Ermitteln aller Möglichkeiten, einen der gegebenen Quader in einer festgelegten inneren Ecke zu platzieren (Schritt 4c), berücksichtigt werden. Kommt ein Quader mehrfach vor, dann muss nur für einen der untereinander identischen Quader probiert werden, ihn (in allen möglichen Rotationen) in der inneren Ecke zu platzieren. Auf die Art können schneller alle Möglichkeiten, die inneren Ecken zu eliminieren, durchprobiert werden, da identische Quader nicht unnötigerweise mehrfach betrachtet werden.

Optimierung: Punktsymmetrische Befüllung

Bei Betrachtung der Beispiele von der BWinf-Webseite stellt man fest, dass drei der fünf Beispiele[2] als Lösung eine Anordnung haben, die punktsymmetrisch zum goldenen Würfel bzw. zur Mitte der Kiste ist. Weiß man von einer Kiste, dass sie punktsymmetrisch befüllt werden muss, beschleunigt dies das im Vorkapitel beschriebene Verfahren enorm, da immer zwei Quader auf einmal platziert werden können (der eigentlich zu platzierende Quader und der dazu punktsymmetrisch gespiegelte Quader können gleichzeitig platziert werden, somit können mehrere innere Ecken auf einmal eliminiert werden). Es lohnt sich daher, bei einem gegebenen Rätsel immer zunächst eine punktsymmetrische Befüllung der Kiste zu versuchen[3] (auch wenn dadurch das Vorgehen für Kisten mit nicht punktsymmetrischer Befüllung etwas aufwendiger wird, siehe Fußnote 3). Sollte diese erfolglos bleiben, kann danach versucht werden, die Kiste ohne Beachtung von Symmetrien zu befüllen.

Umsetzung

Die Lösungsidee wird in Python 3.10 implementiert. Das Programm verwendet das PyPi-Library *numpy*, um eine Kiste als dreidimensionales Array darzustellen, dass als Dimensionen die Seitenlängen der Kiste hat. Nicht von Quadern belegte Plätze werden im Array als 0 dargestellt. Den einzelnen Quadern werden Indices >=1 zugewiesen, ein von einem Quader belegter Platz wird im Array mit dem Index des jeweiligen Quaders dargestellt. Der goldene Würfel wird im Array als Eintrag mit dem Wert -1 gespeichert .

Einlesen der Eingabedatei und Abgleich der Gesamtvolumina

Die Eingabedatei wird vom Nutzer als erstes Kommandozeilenargument angegeben und vom Programm eingelesen.

```
import numpy as np
import sys

# Textdatei einlesen
# sys.argv[1] ist das erste Kommandozeilenargument
with open(sys.argv[1]) as f:
    input_lines = f.read().split("\n")
```

Das Programm speichert anschließend die inneren Kantenlängen der Kiste als Tripel namens *kiste_dims* und berechnet das Gesamtvolumen der Kiste. Dann speichert es in einem Dictionary namens *quaders* für jeden Quader, wie oft er in der Menge an gegebenen Quadern vorkommt. Parallel dazu wird das Gesamtvolumen aller gegebenen Quader (inklusive des goldenen Würfels) berechnet. Die beiden Gesamtvolumina werden anschließend verglichen; wenn sie nicht identisch sind, wird ausgegeben, dass es keine Lösung geben kann, und das Programm beendet:

```
if kiste_gesamtvolumen != quaders_gesamtvolumen:
    print("Es kann keine Lösung geben, da des Gesamtvolumen der einzufüllenden
Quader (einschl. dem goldenen Würfel) nicht dem Volumen der Kiste entspricht.")
```

[2] Beispiel 1, Beispiel 2 und Beispiel 5 haben punktsymmetrische Lösungen

[3] Das Versuchen einer punktsymmetrischen Befüllung geht relativ schnell, da immer zwei Quader gleichzeitig platziert werden. Bei Kisten, die nicht punktsymmetrisch befüllt werden können, wird das Finden einer Lösung durch das Probieren einer punktsymmetrischen Befüllung also kaum verlangsamt.

```
exit()
```

Definieren einer Klasse zum Speichern einer Kiste

Zum Speichern einer teilweise befüllten Kiste wird eine Klasse namens *Kiste* definiert, die diese Attribute hat:

- *Kiste.dims : tuple* – Die Seitenlängen der Kiste als Tripel gespeichert
- *Kiste.inhalt : np.array* – Der Inhalt der Kiste als numpy Array gespeichert
- *Kiste.quaders : dict* – Ein dictionary mit allen Quadern, die noch in die Kiste gefüllt werden müssen
- *Kiste.num_placed_quaders : int* – Gibt an, wie viele Quader bereits in der Kiste platziert wurden
- *Kiste.symmetric : boolean* – Gibt an, ob die Kiste punktsymmetrisch zur Mitte befüllt werden soll

- Die Klasse hat eine *place_quader(self, quader_dims, position, mirror=False)* Funktion, die einen Quader mit den Kantenlängen *quader_dims* in der Kiste an der Position *position* platziert, das Argument *mirror* gibt an ob ein zweiter Quader punktsymmetrisch gespiegelt platziert werden soll. Der platzierte Quader wird von der Funktion aus *Kiste.quaders* entfernt.
- Die Klasse hat eine *get_corner_index()*-Funktion, die eine beliebige innere Ecke der Kiste ermittelt und zurückgibt.
- Die Klasse hat eine *print()*-Funktion, die die Kiste formatiert in der Konsole ausgibt.
- Die Klasse hat außerden eine *next_kisten()*-Funktion. Diese Funktion wählt zunächst mit der *get_corner_index()*-Funktion eine innere Ecke aus, die unter dem Namen *corner_position* gespeichert wird. Dann ermittelt sie alle Möglichkeiten, einen noch nicht verbauten Quader in beliebiger Ausrichtung an dieser Ecke zu platzieren. Hierfür iteriert sie zunächst über alle noch nicht platzierten Quader und bestimmt für jeden Quader alle möglichen Rotationen:

```
for quader in list(self.quaders.keys()):
    next_quaders = dict(self.quaders)
    next_quaders[quader] -= 1 + self.symmetric
    if next_quaders[quader] == -1:
        return [] # -> Kiste kann nicht punktsymmetrisch befüllt werden, da es
                    nur einen Quader mit diesen Kantenlängen gibt und der
                    Quader somit nicht punktsymmetrisch gespiegelt werden kann
    if next_quaders[quader] == 0:
        next_quaders.pop(quader)

    # Alle möglichen Rotationen des Quaders ermitteln:
    quader_rotations = []
    for i in range(2):
        for i in range(3):
            quader = (quader[1], quader[2], quader[0])
            quader_rotations.append(quader)8
        quader = quader[::-1]
```

Anschließend iteriert sie über alle möglichen Rotationen des jeweiligen Quaders und versucht in jeder Iteration, den rotierten Quader an der zuvor ausgewählten inneren Ecke *corner_position* einzufügen:

```
# Über alle möglichen Rotationen des Quaders iterieren:
for quader in set(quader_rotations): # quader_rotations zu Set
```

```
                                                konvertieren, um Duplikate zu entfernen
                new_inhalt = self.place_quader(quader, corner_position,
mirror=self.symmetric) # Versuchen, den rotierten Quader an der festgelegten
                            Eckposition hinzuzufügen. Wenn die Kiste symmetrisch befüllt
                   werden soll, dann wird ein zweiter gespiegelter Quader mitplatziert
                if new_inhalt is None:
                    # Platzieren des rotierten Quaders an der Eckposition nicht möglich
-> Nächste Rotation ausprobieren
                    continue
                else:
                    # Platzieren des rotierten Quaders erfolgreich -> Resultat
speichern
                    next_kisten.append(Kiste(self.dims, new_inhalt, next_quaders,
self.num_placed_quaders+self.symmetric+1, symmetric=self.symmetric))

                if self.num_placed_quaders == 0:
                    break
```

Alle neuen Kisten, die so erzeugt wurden, werden zurückgegeben:

```
return next_kisten
```

Durchprobieren aller Möglichkeiten, innere Ecken zu eliminieren

Eine Liste namens *alle_kisten* wird initialisiert. Sie enthält zunächst zwei *Kiste*-Objekte, die jeweils eine Kiste repräsentieren, in der lediglich der goldene Würfel platziert ist. Im ersten *Kiste*-Objekt ist *symmetric* auf True gesetzt, es dient als Startpunkt für den Versuch, die Kiste symmetrisch zu füllen. Im zweiten *Kiste*-Objekt ist *symmetric* auf False gesetzt.

```
inhalt = np.zeros(kiste_dims, dtype=int)
mitte = (int(kiste_dims[0]/2),int(kiste_dims[1]/2),int(kiste_dims[2]/2))
inhalt[mitte] = -1
alle_kisten = [Kiste(kiste_dims, inhalt, quaders, symmetric=True), Kiste(kiste_dims,
inhalt, quaders, symmetric=False)]
```

In einer while-Schleife wird nun das erste Element aus *alle_kisten* entnommen, alle aus der entnommenen Kiste hervorgehenden Kisten werden mit *kiste.next_kisten()* bestimmt und vorne in *alle_kisten* eingefügt:

```
while alle_kisten != []:

    kiste = alle_kisten.pop(0)
    if kiste.num_placed_quaders == num_quaders:

        # Lösung gefunden
        print("Lösung gefunden")
        kiste.print()
        break

    next_kisten = kiste.next_kisten()
    alle_kisten = next_kisten + alle_kisten
```

```
else:

    print("Es gibt keine Lösung")
```

Ausgabe

Wird eine mögliche Befüllung gefunden, dann wird diese in einer ebenenweisen Darstellung ausgegeben, so wie in der README.txt Datei vorgeschlagen. Hierfür wird die *Kiste.print*() Funktion verwendet, die die Kiste in einem solchen Format ausgibt (siehe Kapitel, in dem die *Kiste*-Klasse definiert wird). Ansonsten wird ausgegeben, dass es keine Lösung gibt.

Beispiele

Die in Beispiel 1 - 5 verwendeten Eingabedateien stammen von der BWinf-Webseite. Die in Beispiel 6 und 7 verwendeten Eingabedateien habe ich erstellt, um Besonderheiten zu demonstrieren. Alle Eingabedateien sind im Ordner *A2_DieGoldeneMitte/* zu finden. Zu Beispiel *x* gehört stets die Eingabedatei raetsel*x*.txt.

Beispiel 1

Eingabedatei:
raetsel1.txt

Inhalt der Eingabedatei:

```
3 3 3
8
1 1 3
1 1 3
1 1 2
1 1 2
1 2 2
1 2 2
1 2 2
1 2 2
```

Ausgabe:

```
Lösung gefunden

Ebene 1
1    1    1
3    3    5
7    7    5

Ebene 2
6    8    8
6    G    5
7    7    5

Ebene 3
6    8    8
6    4    4
2    2    2
```

Evalierungsdauer:
0,21 Sekunden

Beispiel 2

Eingabedatei:
raetsel2.txt

Inhalt der Eingabedatei:
```
3 3 3
6
3 3 1
3 3 1
1 1 1
1 1 1
1 1 3
1 1 3
```

Ausgabe:
```
Lösung gefunden

Ebene 1
1    1    1
3    3    3
2    2    2

Ebene 2
1    1    1
5    G    6
2    2    2

Ebene 3
1    1    1
4    4    4
2    2    2
```

Evalierungsdauer:
0,204 Sekunden

Beispiel 3

Eingabedatei:
raetsel3.txt

Inhalt der Eingabedatei:
```
3 3 3
6
1 1 1
1 1 1
3 2 1
2 2 2
3 2 1
```

```
2 2 1
```

Ausgabe:
```
Es gibt keine Lösung
```

Evalierungsdauer:
0,18 Sekunden

Anmerkung:
Das Programm gibt korrekterweise aus, dass es keine Lösung gibt. Der Quader mit den Maßen (2,2,2) kann nämlich nirgendwo platziert werden, da er immer mit dem goldenen Würfel kollidieren würde.

Beispiel 4

Eingabedatei:
raetsel4.txt

Inhalt der Eingabedatei:
siehe Datei *A2_DieGoldeneMitte/raetsel4.txt*

Ausgabe:
```
Lösung gefunden

Ebene 1
 1     2     2     3     3
 1     2     2     3     3
 1     4     4     5     5
 1     4     4     5     5
 6     7     8     9    10

Ebene 2
 1     2     2     3     3
 1     2     2     3     3
 1     4     4     5     5
 1     4     4     5     5
 6     7     8     9    10

Ebene 3
11    11    12    12    12
11    11    12    12    12
11    11     G    13    14
15    15    15    13    14
 6     7     8     9    10

Ebene 4
16    16    16    16    16
17    17    17    17    17
18    18    18    13    14
15    15    15    13    14
 6     7     8     9    10

Ebene 5
19    19    19    19    19
20    20    20    20    20
```

```
18    18    18    13    14
15    15    15    13    14
 6     7     8     9    10
```

Evalierungsdauer:
0,24 Sekunden

Anmerkung:
Das Programm findet auch bei größeren Kisten sehr schnell (in weniger als einer halben Sekunde) eine Lösung, wenn es eine solche gibt.

Beispiel 5

Eingabedatei:
raetsel5.txt

Inhalt der Eingabedatei:
Siehe Datei *A2_DieGoldeneMitte/raetsel5.txt*

Ausgabe:

```
Lösung gefunden

Ebene 1
 1     3     3     3     3
 5     5     7     7     7
 5     5     7     7     7
 5     5     9     9    11
 5     5     9     9    11

Ebene 2
12     3     3     3     3
12    13     7     7     7
15    15     7     7     7
15    15     9     9    11
15    15     9     9    11

Ebene 3
12    10    10    16    16
12    10    10    16    16
15    15     G    16    16
15    15     9     9    11
15    15     9     9    11

Ebene 4
12    10    10    16    16
12    10    10    16    16
 8     8     8    16    16
 8     8     8    14    11
 4     4     4     4    11

Ebene 5
12    10    10     6     6
12    10    10     6     6
 8     8     8     6     6
 8     8     8     6     6
```

```
4     4     4     4     2
```

Evalierungsdauer:
0,25 Sekunden

Anmerkung:
Bei diesem Beispiel ist die befüllte Kiste punktsymmetrisch zum goldenen Würfel bzw. zur Mitte. Dieses Beispiel kann dank der vorgenommenen Optimierung (Versuchen einer punktsymmetrischen Lösung) sehr schnell gelöst werden.

Beispiel 6

Eingabedatei:
raetsel6.txt

Inhalt der Eingabedatei:
siehe Datei *A2_DieGoldeneMitte/raetsel6.txt*

Ausgabe:
```
Es kann keine Lösung geben, da des Gesamtvolumen der einzufüllenden Quader (einschl.
dem goldenen Würfel) nicht dem Volumen der Kiste entspricht.
```

Evalierungsdauer:
0,172 Sekunden

Anmerkung:
Dieses Beispiel wurde von mir erstellt. Der Inhalt der Eingabedatei entsprecht weitgehend dem von Beispiel 5, allerdings wurde ein Quader durch einen Quader mit den Maßen (7,7,7) ersetzt. Das Programm erkennt direkt, dass das Gesamtvolumen der Quader das Volumen der Kiste übersteigt und bricht ab.

Beispiel 7

Eingabedatei:
raetsel7.txt

Inhalt der Eingabedatei:
siehe Datei *A2_DieGoldeneMitte/raetsel7.txt*

Ausgabe:
```
Keine Lösung gefunden
```

Evalierungsdauer:
62,42 Sekunden

Anmerkung:
Dieses Beispiel wurde von mir erstellt (Es handelt um keines der Beispiele von der BWinf-Webseite. Alle Beispiele von der BWinf Webseite werden von meinem Programm in weniger als einer Sekunde ausgeführt). Es zeigt, dass das Programm es auch bei größeren Kisten in vertretbarer Zeit (~ 1 Minute) erkennt, wenn es keine Lösung gibt.

Quellcode

```
import numpy as np
import sys

# Textdatei einlesen
# sys.argv[1] ist das erste Kommandozeilenargument
with open(sys.argv[1]) as f:
    input_lines = f.read().split("\n")

class Kiste:

    """
    Dient zum Speichern einer teilweise befüllten Kiste
    """

    def __init__(self, kiste_dims, inhalt, quaders, num_placed_quaders=0, *, symmetric):
        self.dims = kiste_dims # Die Seitenlängen der Kiste (tuple)
        self.inhalt = inhalt # Der Inhalt der Liste (np.array)
        self.quaders = dict(quaders) # Ein dictionary mit allen Quadern, die noch in die
Kiste gefüllt werden müssen
        self.num_placed_quaders = num_placed_quaders # Anzahl an Quadern, die bisher in die
Kiste gefüllt wurden
        self.symmetric = symmetric # Gibt an, ob die Befüllung der Kiste punktsymmetrisch
zur Mitte (zum goldenen Würfel) sein soll

    def place_quader(self, quader_dims, position, *, mirror=False):
        """
        Platziert einen Quader in der Kiste an der angebenen Position, sofern dies möglich
ist

        Args:
            quader_dims (tuple): Die Seitenlängen des Quaders, der platziert werden soll
            position (tuple): Die Koordinaten der Position, an der der Quader platziert
werden soll
            mirror (boolean): Gibt an, ob punktsymmetrisch zum goldenen Würfel ein weitere
Quader der gleichen Seitenlängen platziert werden soll
        """
        new_inhalt = np.array(self.inhalt)
        range0 = (position[0],position[0]+quader_dims[0])
        range1 = (position[1],position[1]+quader_dims[1])
        range2 = (position[2],position[2]+quader_dims[2])
        if range0[1] > self.dims[0] or range1[1] > self.dims[1] or range2[1] > self.dims[2]:
            return None
        if False in (new_inhalt[ range0[0]:range0[1], range1[0]:range1[1],
range2[0]:range2[1]] == 0):
            return None
        new_inhalt[ range0[0]:range0[1], range1[0]:range1[1], range2[0]:range2[1]] =
self.num_placed_quaders + 1
        if mirror:
            new_inhalt[ self.dims[0]-range0[1]:self.dims[0]-range0[0], self.dims[1]-
range1[1]:self.dims[1]-range1[0], self.dims[2]-range2[1]:self.dims[2]-range2[0]] =
self.num_placed_quaders + 2

        return new_inhalt

    def get_corner_index(self):
        """
        Eine beliebige innere Ecke der Kiste ermitteln
        """
        for index0 in range(self.dims[0]):
```

```
            for index1 in range(self.dims[1]):
                for index2 in range(self.dims[2]):
                    if self.inhalt[index0, index1, index2] != 0:
                        continue

                    if self.inhalt[index0-1, index1, index2] != 0 or index0-1 < 0:
                        if self.inhalt[index0, index1-1, index2] != 0 or index1-1 < 0:
                            if self.inhalt[index0, index1, index2-1] != 0 or index2-1 < 0:
                                return index0, index1, index2

    def next_kisten(self):
        """
        Ermittelt alle Kisten, die aus dieser Kiste hervorgehen können
        """
        corner_position = self.get_corner_index() # Eckposition festlegen, an der die zu
Verfüung stehenden Quader platziert werden sollen
        next_kisten = []
        # Über alle noch nicht platzierten Quader iterieren:
        for quader in list(self.quaders.keys()):
            next_quaders = dict(self.quaders)
            next_quaders[quader] -= 1 + self.symmetric
            if next_quaders[quader] == -1:
                return [] # -> Kiste kann nicht punktsymmetrisch befüllt werden
            if next_quaders[quader] == 0:
                next_quaders.pop(quader)

            # Alle möglichen Rotationen des Quaders ermitteln:
            quader_rotations = []
            for i in range(2):
                for i in range(3):
                    quader = (quader[1], quader[2], quader[0])
                    quader_rotations.append(quader)
                quader = quader[::-1]


            # Über alle möglichen Rotationen des Quaders iterieren:
            for quader in set(quader_rotations): # quader_rotations zu Set konvertieren, um
Duplikate zu entfernen
                new_inhalt = self.place_quader(quader, corner_position,
mirror=self.symmetric) # Versuchen, den rotierten Quader an der festgelegten Eckposition
hinzuzufügen
                if new_inhalt is None:
                    # Platzieren des rotierten Quaders an der Eckposition nicht möglich ->
Nächste Rotation ausprobieren
                    continue
                else:
                    # Platzieren des rotierten Quaders erfolgreich -> Resultat speichern
                    next_kisten.append(Kiste(self.dims, new_inhalt, next_quaders,
self.num_placed_quaders+self.symmetric+1, symmetric=self.symmetric))

                if self.num_placed_quaders == 0:
                    break

        return next_kisten

    def print(self):
        """
        Gibt den Inhalt der Kiste in der Konsole aus
        """
        for ebene in range(len(self.inhalt)):
            print("\nEbene", ebene+1)
```

```
            for row in self.inhalt[ebene]:
                output = ""
                for item in row:
                    item = str(item)
                    if item == "-1":
                        item = "G"
                    while len(item) < len(str(num_quaders)):
                        item = " "+item
                    output += str(item) + "    "
                print(output)


# Größe der Kiste einlesen:
data = input_lines.pop(0).split(" ")
kiste_dims = (int(data[0]), int(data[1]), int(data[2]))
kiste_gesamtvolumen = kiste_dims[0] * kiste_dims[1] * kiste_dims[2]

# Größen der Quader einlesen:
num_quaders = int(input_lines.pop(0))
quaders = {}
quaders_gesamtvolumen = 1
for i in range(num_quaders):
    data = input_lines.pop(0).split(" ")
    data = (int(data[0]), int(data[1]), int(data[2]))
    if data in quaders:
        quaders[data] += 1
    else:
        quaders[data] = 1
    quaders_gesamtvolumen += int(data[0]) * int(data[1]) * int(data[2])

# Überprüfen, ob Gesamtvolumen der Quader dem Volumen der Kiste entspricht (wenn nicht, dann
können die Quader gar nicht alle in die Kiste passen):
if kiste_gesamtvolumen != quaders_gesamtvolumen:
    print("Es kann keine Lösung geben, da des Gesamtvolumen der einzufüllenden Quader
(einschl. dem goldenen Würfel) nicht dem Volumen der Kiste entspricht.")
    exit()

# Startkiste erzeugen:
inhalt = np.zeros(kiste_dims, dtype=int)
mitte = (int(kiste_dims[0]/2),int(kiste_dims[1]/2),int(kiste_dims[2]/2))
inhalt[mitte] = -1
alle_kisten = [Kiste(kiste_dims, inhalt, quaders, symmetric=True), Kiste(kiste_dims, inhalt,
quaders, symmetric=False)]

# Alle Möglichkeiten, innere Ecken zu eliminieren, durchprobieren:

while alle_kisten != []:

    kiste = alle_kisten.pop(0)
    if kiste.num_placed_quaders == num_quaders:

        # Lösung gefunden
        print("Lösung gefunden")
        kiste.print()
        break

    next_kisten = kiste.next_kisten()
    alle_kisten = next_kisten + alle_kisten

else:
```

```
    print("Es gibt keine Lösung")
```