Aufgabe 1: Kisten in Kisten in Kisten

1 Lösungsidee

1.1 Kisten

Allgemein gilt:

- eine Kiste wird durch das Tripel (L, B, H) beschrieben
- die inneren Maße der Kiste sind (L – 1, B – 1, H – 1)
- die kleinste mögliche Kiste ist (2, 2, 2)
- das Volumen einer Kiste beträgt L * B * H
- es sind keine Schrägpackungen erlaubt
- eine bereits verpackte Kiste kann nicht nochmal verpackt werden
- in einer Kiste dürfen nicht mehr als zwei Kisten verpackt werden

1.1.1 Eine Kiste verpacken

Aufgrund der Vorbedingungen kann eine Kiste nur sechs verschiedene Lagen im Raum annehmen:

- keinerlei Drehung
- 90º um die x-Achse gedreht
- 90º um die y-Achse gedreht
- 90º um die z-Achse gedreht
- 90º um die x-Achse, dann 90º um die y-Achse gedreht
- 90º um die x-Achse, dann 90º um die z-Achse gedreht

Eine 180º Drehung der Kiste um eine der Achsen hat dasselbe Ergebnis wie eine 0º bzw. 360º Drehung um diese Achse.

Die Rotationsachsen einer Kiste habe ich folgendermaßen definiert:

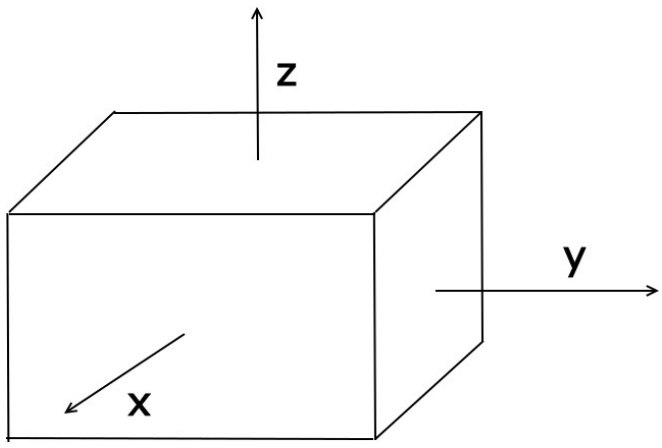

Abbildung 1: Rotationsachsen einer Kiste

Die Anzahl der Lagen im Raum einer Kiste ist zurückzuführen auf die Permutation der drei Werte der Kiste. Eine Kiste hat genau drei Werte, die permutiert werden können, deshalb kann eine Kiste genau 3! Lagen im Raum annehmen. Demzufolge gibt es weniger Möglichkeiten die Kiste im Raum anzuordnen, wenn zwei oder alle drei Werte gleich sind:

Bei zwei gleichen Werten (z. B. Länge = Breite) gibt es $\frac{3!}{2! \cdot 1!} = 3$ Möglichkeiten eine Kiste im Raum anzuordnen. Bei drei gleichen Werten (Würfel) gibt es nur noch eine Möglichkeit.

Wird eine Kiste in eine andere Kiste gepackt, so muss geprüft werden, ob die zu packende Kiste in mindestens einer dieser sechs Lagen in die andere Kiste passt. Dazu müssen lediglich die Werte der einen Kiste mit den Werten der anderen Kiste verglichen werden:

$$\begin{array}{lll} L_A < L_B \wedge B_A < B_B \wedge H_A < H_B & \rightarrow & \text{keine Drehung} \\ L_A < L_B \wedge B_A < H_B \wedge H_A < B_B & \rightarrow & 90^\circ \text{ um x-Achse gedreht} \\ L_A < H_B \wedge B_A < B_B \wedge H_A < L_B & \rightarrow & 90^\circ \text{ um y-Achse gedreht} \\ \ldots & & \ldots \end{array}$$

Abbildung 2: Prüfung, ob Kiste A in Kiste B gepackt werden kann

Wenn eine der Aussagen wahr ist, dann passt die Kiste A in die Kiste B.

1.1.2 Zwei Kisten verpacken

Die erste Kiste kann in sechs verschiedenen Lagen sowohl hinter, neben als auch unter der zweiten Kiste platziert werden. Die aneinander gepackten Kisten werden dann als eine Kiste angesehen, die wiederum in sechs verschiedenen Möglichkeiten in die „Ziel-Kiste“ gepackt werden kann. Dadurch ergeben sich 6 * 3 * 6 = 108 Möglichkeiten zwei Kisten in eine andere Kiste zu packen. Hat eine der beiden Kisten die Form eines Würfels, dann gibt es dementsprechend weniger Möglichkeiten.

Beim Prüfen, ob zwei Kisten in eine andere Kiste passen, wird zunächst eine der beiden Kisten hinter, neben und unter der zweiten Kiste in je sechs Lagen platziert. Nach jeder der (max.) 18 Kombination werden die zwei Kisten als eine Kiste angesehen und versucht in die Ziel-Kiste zu platzieren.

1.2 Methoden zur Suche nach einer (optimalen) Verschachtelung

Um die Verschachtelung zu erhalten, die das optimale Volumen der äußeren Kisten besitzt, kann man die Brute-Force-Methode anwenden. Um die optimale Verschachtelung zu erhalten müssen alle Möglichkeiten die Kisten zu verschachteln durchlaufen werden. Die Verschachtelung mit dem geringsten Gesamtvolumen der äußeren Kisten ist dann die optimale Lösung.

Die Brute-Force-Methode findet zwar die optimale Lösung, braucht jedoch exponentiell mehr Zeit, je mehr Kisten es zu packen gibt. Es müssen also andere Methoden in Betracht gezogen werden, die zwar nicht immer die optimale Lösung finden, dafür aber schneller als die Brute-Force-Methode sind.

Ich habe vier Methoden entworfen, die schneller als die Brute-Force-Methode sind und zudem gute Ergebnisse liefern. Die Methoden werden mit den Buchstaben A bis D bezeichnet.

Voraussetzung für jede Methoden ist, dass die Kisten aufsteigend nach ihrem Volumen sortiert sind. Die erste Kiste ist also die Kiste mit dem geringsten Volumen. Die letzte Kiste ist dementsprechend die größte Kiste.

Die Methoden A bis D werden nacheinander ausgeführt. Nachdem alle Methoden eine Verschachtelung gefunden haben, wird die beste Verschachtelung in Form einer „Pack-Anleitung“ auf dem Bildschirm ausgegeben. Die beste Verschachtelung ist die Verschachtelung mit dem geringsten Gesamtvolumen der äußeren Kisten.

Die Methoden A und B sind die schnellsten Methoden zur Suche nach einer Verschachtelung. Die Methoden C und D brauchen etwas mehr Zeit. Jede Methode hat ihre Vor- und Nachteile, weshalb es sinnvoll ist alle Methoden zu implementieren.

Um bei sehr vielen Kisten möglichst schnell ein Ergebnis zu bekommen, kann Frau Y. zusätzlich noch eine maximale Kompression angeben. Eine Kompression von 1 würde bedeuten, dass keine Kisten verpackt sind. Eine Kompression von 2 hieße, dass das Gesamtvolumen der äußeren Kisten nur halb so groß ist wie das Gesamtvolumen aller Kisten. Die Kompression lässt sich berechnen mit:

$$\text{Kompression} = \frac{\text{Gesamtvolumen aller Kisten}}{\text{Gesamtvolumen der äußeren Kisten}}$$

Sobald eine der Methoden die Kompression erreicht oder sogar überschritten hat muss kein weiterer Rechenaufwand betrieben werden und die „Pack-Anleitung“ kann sofort ausgegeben werden.

Eine geringere Kompression ist sinnvoll, wenn Frau Y. nicht so viel Zeit beim Ver- und Auspacken der Kisten aufwenden will.

1.2.1 Methode A: Kisten nur einzeln verpacken

Beginnend bei der kleinsten Kiste wird versucht jede Kiste in eine der größeren Kisten zu packen. Sobald eine Kiste in eine andere Kiste passt, wird diese sofort verpackt. Passt die kleinere Kiste mal nicht in eine der größeren Kisten, so wird versucht die kleinere Kiste in die nächste Kiste zu packen:

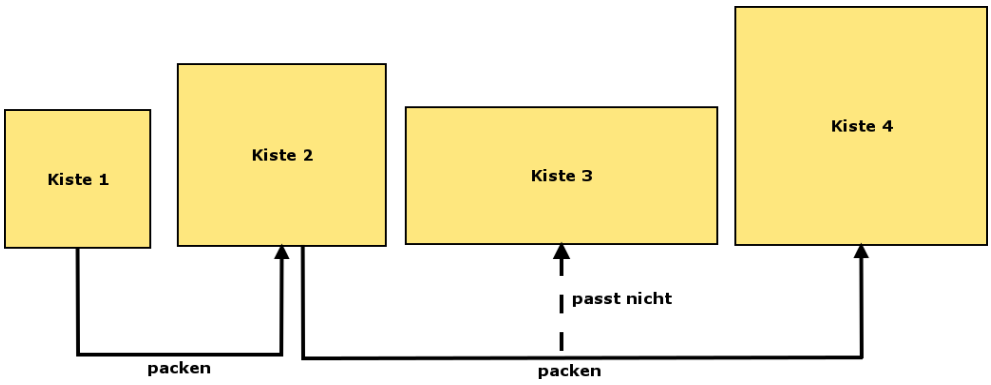

Abbildung 3: Methode A

Das Verpacken ist abgeschlossen, wenn keine Kiste mehr verpackt werden konnte.

Diese Methode habe ich vor allem zum Lösen eines Worst Case entwickelt. Ein Worst Case für die Brute-Force-Methode ist beispielsweise, dass es die Kisten (2, 2, 2), (3, 3, 3), (4, 4, 4), usw. zu verpacken gilt. Die triviale Lösung dazu ist, dass jede Kiste in die nächst größere gepackt wird und nur die letzte Kiste übrig bleibt.

1.2.2 Methode B: Zuerst zwei Kisten verpacken

Diese Methode ist der Methode A ähnlich, nur wird zunächst versucht immer zwei Kisten in größere Kisten zu packen. Die „Paarbildung“ beginnt bei den kleinsten Kisten.

Auch hier kann es wieder vorkommen, dass die beiden Kisten nicht in die nächst größere Kiste passen:

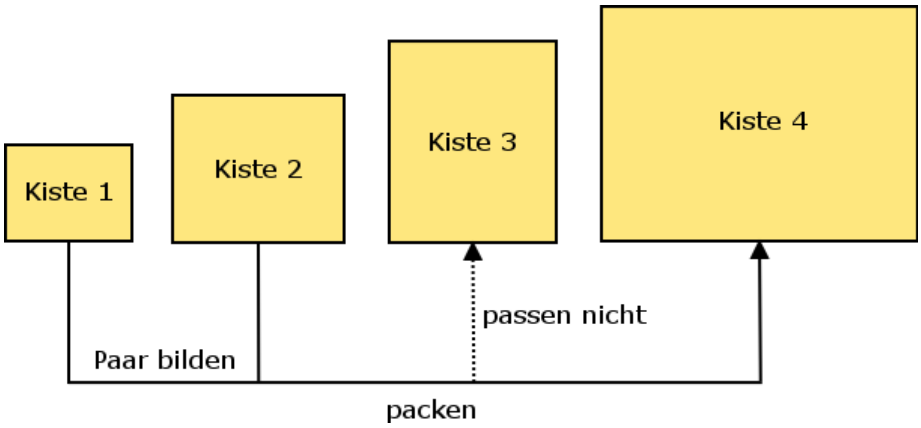

Abbildung 4: Methode B

Nachdem keine Paare von Kisten mehr verpackt werden können, werden die übrigen nicht-verpackten Kisten mit der Methode A verpackt.

1.2.3 Methode C: Quick-Brute-Force

Diese Methode der Suche nach einer Verschachtelung ist nach dem „divide and conquer“-Prinzip konstruiert worden.

Für je sieben Kisten wird ein Brute-Force-Algorithmus durchgeführt, der die optimale Verschachtelung dieser sieben Kisten herausfindet.

Der Brute-Force-Algorithmus durchläuft alle Möglichkeiten die sieben Kisten zu verschachteln und merkt sich nur die Verschachtelung mit dem geringsten Gesamtvolumen der äußeren Kisten.

Dabei werden die sieben Kisten wie folgt gewählt:

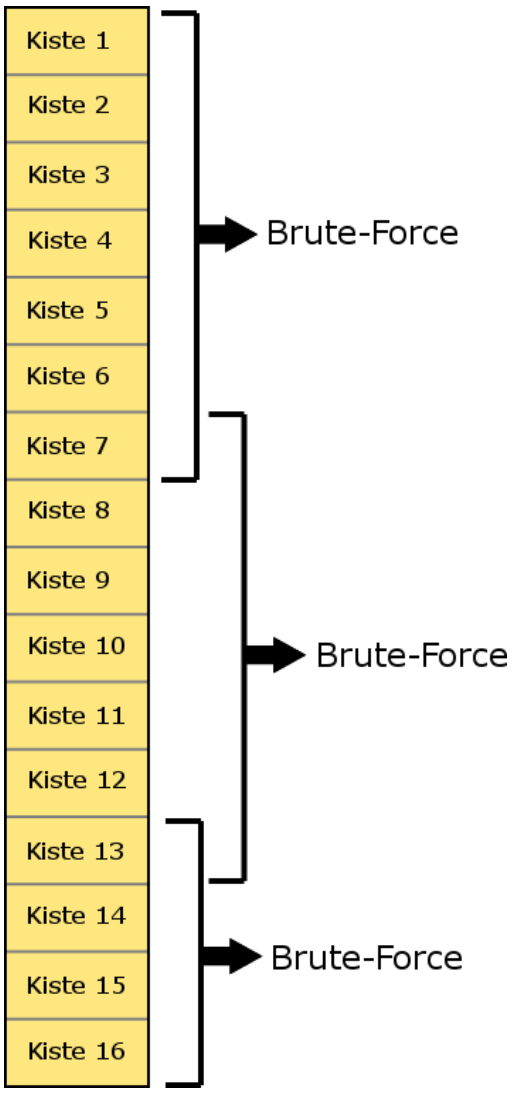

Abbildung 5: Quick-Brute-Force für 16 Kisten

Anschließend werden eventuell noch nicht verpackte Kisten mit den schnellen Methoden A und B verpackt.

Ich habe genau sieben Kisten gewählt, weil damit gewährleistet ist, dass die Methode auch noch bei sehr vielen Kisten schnell ist.

1.2.4 Methode D: Minimaler Zwischenraum

Diese Methode erfüllt einen ähnlichen Zweck, wie die Methoden A und B. Nur wird vor jeder „Verpack-Aktion“ die Kiste bzw. das Kistenpaar herausgefunden, welches mit minimalen Zwischenraum verpackt werden kann.

Den Zwischenraum habe ich definiert als die Differenz des Volumens der äußeren Kiste und des Volumens der darin verpackten Kiste(n).

Demzufolge wird herausgefunden, ob es jeweils besser ist nur eine Kiste oder zwei Kisten zu verpacken:

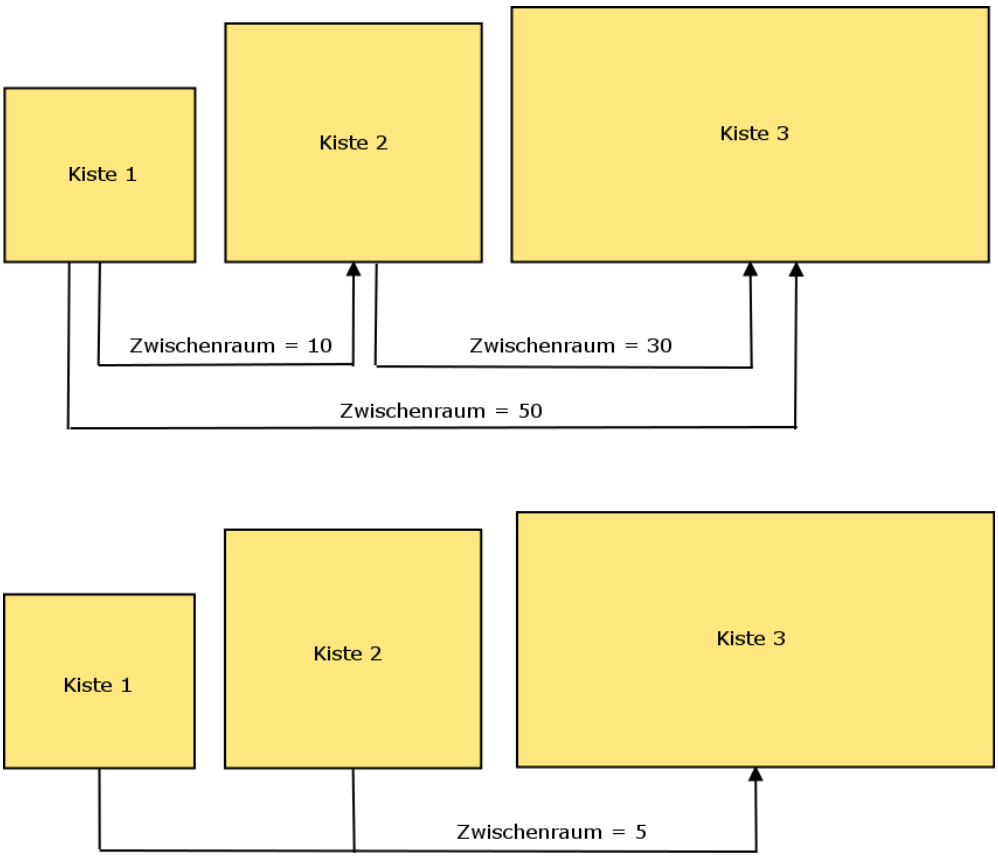

Abbildung 6: Bestimmung des minimalen Zwischenraumes

In diesem Beispiel erkennt der Algorithmus das es besser ist die Kisten 1 und 2 in Kiste 3 zu packen, weil dort der Zwischenraum kleiner ist.

Diese Methode versucht, den Platz in eine Kiste optimal zu nutzen. Dadurch soll erreicht werden, dass das Gesamtvolumen der äußeren Kisten möglichst gering wird.

2 Programm-Dokumentation

Die Lösungsidee wurde von mir in der Programmiersprache C umgesetzt. Die Kisten werden zunächst eingelesen, dann werden nacheinander die Methoden A - D auf die Kisten angewendet. Die Anleitung zur Verschachtelung wird dann auf dem Bildschirm ausgegeben.

2.1 Strukturen

Zur Umsetzung der Lösungsidee wurden die Strukturen **`kiste`** und **`keller`** benötigt.

Die Struktur **`kiste`** besteht aus drei **`integer`**-Variablen, die die Länge, Breite und Höhe der Kiste speichern. Des Weiteren gibt es eine **`char`**-Variable, welche den Wert 1 hat, wenn die Kiste in einer anderen Kiste verpackt ist. Andernfalls hat die Variable den Wert 0.

Das Volumen der Kiste wird in einer **`unsigned long`**-Variable gespeichert um selbst große Kisten abspeichern zu können.

Die Information welche und wie viele Kisten in der Kiste verpackt sind werden in Form von zwei **`kiste`**-Zeigern gespeichert. Haben beide Zeiger den Wert **`NULL`**, dann ist die Kiste leer. Ist nur der erste Zeiger ungleich **`NULL`**, dann enthält die Kiste eine andere Kiste. Sind beide Zeiger ungleich **`NULL`**, dann sind in der Kiste zwei andere Kisten verpackt.

Außerdem besitzt die Struktur **`kiste`** noch eine **`integer`**-Variable, in der der Index der Kiste gespeichert wird. Dies ist zum Kopieren einer **`keller`**-Struktur notwendig.

Die Struktur **`keller`** hat ein Array von **`kisten`**-Strukturen. Die Anzahl der in der **`keller`**-Struktur gespeicherten Kisten wird in einer **`integer`**-Variable gespeichert. Das Gesamtvolumen (das Volumen aller Kisten) und das Volumen der äußeren Kisten wird jeweils in einer **`unsigned long`**-Variable gespeichert.

2.2 Funktionen

Die Kisten werden zunächst aus einer Datei gelesen. Der Dateiname wird dem Programm als Parameter übergeben. Dieser wird dann an die Funktion **`neuer_keller_von_datei`** weitergereicht, die eine **`keller`**-Struktur mit den eingelesenen Kisten an die **`main`**-Funktion zurückgibt. Beim Einlesen der Kisten werden ungültige Kisten, wie z. B. (1, 0, 1) oder (-1, 2, -3) aussortiert.

In der **`main`**-Funktion werden dann nacheinander die Funktionen **`finde_loesung_schnell`**, **`finde_loesung_quickbruteforce`**, **`finde_loesung_minimaler_zwischenraum`** aufgerufen. Jede dieser Funktionen bekommt die eingelesene **`keller`**-Struktur als Parameter übergeben.

In der Funktion **`finde_loesung_schnell`** werden die Methoden A und B zusammengefasst: Die beide Methoden A und B werden nacheinander ausgeführt und nur die bessere Verschachtelung wird an die **`main`**-Funktion zurückgegeben. Die Methode A ist in der Funktion **`packe_schnell_nur_einzeln`** und die Methode B ist in der Funktion **`packe_schnell`** realisiert.

Danach wird die Methode C mit der Funktion **`finde_loesung_quickbruteforce`** ausgeführt. Die Funktion bildet 7er Gruppen von Kisten und ruft die Funktion **`bruteforce_rekursion_von_bis`** für jede dieser Gruppen auf. Diese Funktion führt den Brute-Force-Algorithmus für einen Abschnitt des **`kisten`**-Arrays durch und findet die optimale Verschachtelung für diese sieben Kisten.

Nachdem alle Gruppen durchlaufen wurden, werden nochmal die Funktionen **`packe_schnell`** und

`packe_schnell_nur_einzeln` auf die Kisten angewendet. Die keller-Struktur wird dann an die **`main`**-Funktion zurückgegeben.

Dann wird in der **`main`**-Funktion die Methode D mit der Funktion **`finde_loesung_minimaler_zwischenraum`** ausgeführt. In dieser Funktion wird wiederum die Funktion **`packe_mit_minimalen_zwischenraum`** aufgerufen. In dieser wird für jede Kiste herausgefunden in welche andere Kisten sie gepackt werden kann (sowohl einzeln auch als mit anderen Kisten). Zu jeder Kombinationsmöglichkeit der Kiste wird der Zwischenraum berechnet, der bei der Verschachtelung „übrig“ bleiben würde. Danach werden beginnend bei der Kombination mit dem kleinsten Zwischenraum die Kisten verpackt, bis keine Kiste mehr verpackt werden kann. Die keller-Struktur wird wieder an die **`main`**-Funktion zurückgegeben.

In der **`main`**-Funktion werden die Ergebnisse der einzelnen Methoden „gesammelt“ und dann herausgefunden, welche der Methoden die beste Verschachtelung liefert. Die Anleitung zu der Verschachtelung wird dann auf dem Bildschirm ausgegeben.

Falls eine maximale Kompression angegeben wurde wird das Programm beendet, sobald eine der Methoden die Kompression erreicht oder überschritten hat.

3 Programm-Ablaufprotokoll

Das Programm habe ich **`kisteninkisten`** genannt und wird mit dem Dateinamen einer Datei aufgerufen. Die Datei enthält die Daten der Kisten. Eine Kiste muss in der Datei mit drei ganzzahligen Werten (Länge, Breite, Höhe) innerhalb einer Zeile definiert werden. Die drei Werte müssen durch Leerzeichen oder Tabulatoren getrennt sein. Die Dateiendung der Dateien ist beliebig. Meinen Beispieldateien habe ich die Dateiendung **`.keller`** gegeben.

Zuerst werden die eingelesenen Kisten in sortierter Reihenfolge auf dem Bildschirm ausgegeben, dann wird eine Verschachtelung mit möglichst geringem Gesamtvolumen der äußeren Kisten gesucht. Zum Schluss wird die Anleitung zum Verschachteln der Kisten auf dem Bildschirm ausgegeben. Der Übersicht halber werden in der Anleitung nicht die Kisten angezeigt, die leer bleiben.

3.1 Zufällige Kisten

Zunächst habe ich mir Dateien mit zufälligen Kisten erzeugt. Beispielsweise enthält die Datei **`10kisten.keller`** 10 Kisten mit zufälligen Werten. Führt man das Programm mit dem Dateinamen als Parameter aus, so gibt das Programm folgendes auf den Bildschirm aus:

```
$ ./kisteninkisten 10kisten.keller

Kisten:
=======
Kiste 1: (  4,   5,  11)
Kiste 2: (  5,   8,   7)
Kiste 3: ( 15,   5,   8)
Kiste 4: ( 25,   9,   3)
Kiste 5: (  6,  19,   7)
Kiste 6: ( 29,   6,   8)
Kiste 7: (  4,  33,  17)
Kiste 8: ( 24,   8,  29)
Kiste 9: ( 16,   9,  42)
Kiste 10: ( 12,  38,  44)

Beste Verschachtelung gefunden mit Methode C (Quick-Brute-Force):
Kiste 1 (  4,   5,  11) in Kiste 3 ( 15,   5,   8) packen
Kiste 3 ( 15,   5,   8) und Kiste 5 (  6,  19,   7) in Kiste 8 ( 24,   8,  29) packen
Kiste 4 ( 25,   9,   3) in Kiste 7 (  4,  33,  17) packen
Kiste 2 (  5,   8,   7) in Kiste 6 ( 29,   6,   8) packen
Kiste 6 ( 29,   6,   8) in Kiste 9 ( 16,   9,  42) packen
Kiste 7 (  4,  33,  17) und Kiste 9 ( 16,   9,  42) in Kiste 10 ( 12,  38,  44) packen
Gesamtvolumen der aeusseren Kisten: 25632 (statt 37889)
Kompression: 1.4782
```

In diesem Fall hat der Quick-Brute-Force die beste Verschachtelung herausgefunden.
Weitere Dateien mit zufälligen Kisten sind z. B. **`200kisten.keller`**, **`400kisten.keller`** und **`1000kisten.keller`**.

Oft findet Methode D die beste Verschachtelung der Kisten, z. B. bei den Beispieldateien **`100kisten.keller`** oder **`200kisten.keller`**.

Bei 1000 Kisten benötigt das Programm schon mehr Zeit: Nach etwa 5 Minuten hat die Methode D die beste Verschachtelung gefunden mit einer Kompression von 3,4. Wenn Frau Y. nicht so lange warten möchte und ihr eine Kompression um Faktor 2 reicht, dann könnte sie das Programm aufrufen mit: **`./kisteninkisten -k 2 1000kisten.keller`**. Dann gibt das Programm nämlich schon nach 5 Sekunden eine Verschachtelung mit einer Kompression von 2,4 auf den

Bildschirm aus.

3.2 Sonderfälle

Auch habe ich mir Fälle überlegt, wo die Vor- und Nachteile jeder einzelnen Methoden deutlich werden:

In der Datei **`beispiel5.keller`** sind beispielsweise Kisten enthalten, wo die Methoden A – C nicht die beste Lösung finden und nur der Quick-Brute-Force die Kisten am besten verschachtelt:

```
$ ./kisteninkisten beispiel5.keller

Kisten:
=======
Kiste 1: (  2,   4,   8)
Kiste 2: (  2,   4,   8)
Kiste 3: (  3,   4,   8)
Kiste 4: (  5,   5,   9)
Kiste 5: (  3,   9,  11)

Beste Verschachtelung gefunden mit Methode C (Quick-Brute-Force):
Kiste 3 (  3,   4,   8) in Kiste 4 (  5,   5,   9) packen
Kiste 1 (  2,   4,   8) und Kiste 2 (  2,   4,   8) in Kiste 5 (  3,   9,  11) packen
Gesamtvolumen der aeusseren Kisten: 522 (statt 746)
Kompression: 1.4291
```

Ein Worst Case für einen Brute-Force-Algorithmus ist z. B. in der Datei **`worstcase10.keller`** enthalten:

```
$ ./kisteninkisten worstcase10.keller

Kisten:
=======
Kiste 1: (  2,   2,   2)
Kiste 2: (  3,   3,   3)
Kiste 3: (  4,   4,   4)
Kiste 4: (  5,   5,   5)
Kiste 5: (  6,   6,   6)
Kiste 6: (  7,   7,   7)
Kiste 7: (  8,   8,   8)
Kiste 8: (  9,   9,   9)
Kiste 9: ( 10,  10,  10)
Kiste 10: ( 11,  11,  11)

Beste Verschachtelung gefunden mit Methode A (nur einzeln verpacken):
Kiste 1 (  2,   2,   2) in Kiste 2 (  3,   3,   3) packen
Kiste 2 (  3,   3,   3) in Kiste 3 (  4,   4,   4) packen
Kiste 3 (  4,   4,   4) in Kiste 4 (  5,   5,   5) packen
Kiste 4 (  5,   5,   5) in Kiste 5 (  6,   6,   6) packen
Kiste 5 (  6,   6,   6) in Kiste 6 (  7,   7,   7) packen
Kiste 6 (  7,   7,   7) in Kiste 7 (  8,   8,   8) packen
Kiste 7 (  8,   8,   8) in Kiste 8 (  9,   9,   9) packen
Kiste 8 (  9,   9,   9) in Kiste 9 ( 10,  10,  10) packen
Kiste 9 ( 10,  10,  10) in Kiste 10 ( 11,  11,  11) packen
Gesamtvolumen der aeusseren Kisten: 1331 (statt 4355)
Kompression: 3.2720
```

Der Brute-Force-Algorithmus würde zum Durchlaufen aller Möglichkeiten 15 Minuten benötigen. Mit meinen Methoden braucht das Programm aber nicht mal eine Sekunde, um die optimale Verschachtelung zu bekommen. Die Methode A ist unter anderem genau für diese Fälle vorgesehen.

Das Programm benötigt für den Worst Case mit 100 Kisten nicht mal zwei Sekunden, um die optimale Verschachtelung zu finden. Ein Brute-Force-Algorithmus würde hier sehr lange brauchen,

obwohl die Lösung für dieses Problem trivial ist.

3.3 Triviale Fälle

Außerdem habe ich noch **`.keller`**-Dateien, die die trivialen Fälle testen sollen:

Z. B. enthält die Datei **`nureinekiste.keller`** nur eine Kiste und die Datei **`keinekiste.keller`** keine Kiste.