Aufgabe 1
„Die Kunst der Fuge“- Dokumentation

36. Bundeswettbewerb Informatik 2017/18 - 2. Runde

Lukas Rost

Teilnahme-ID: 44137

9. April 2018

Inhaltsverzeichnis

1 Lösungsidee

1.1 Mögliche Maximalhöhe der Mauer

Da in der Aufgabe explizit nach der Höhe der Mauer für verschiedene n gefragt ist, lohnt es sich, diese zunächst einmal mathematisch zu berechnen. Damit lässt sich der Suchraum effektiv eingrenzen, da ab dieser Maximalhöhe die Forderungen der Aufgabenstellung nicht mehr erfüllt werden können. Pro Reihe der Mauer gibt es $n-1$ verschiedene Fugen. Diese Zahl ergibt sich daraus, dass hinter jedem Klötzchen genau eine Fuge folgt. Nur beim letzten Klötzchen der Reihe zählt diese Position nicht, da sich dahinter kein weiteres Klötzchen befindet.

Fugen sind also per Definition nur die Stellen zwischen den Klötzchen, da die Position 0 sowie die Position $\frac{n\cdot(n+1)}{2}$ am Ende der Reihe in jeder Reihe belegt werden. Nun interessiert noch die Zahl der insgesamt belegbaren Fugenpositionen. Dies sind alle Positionen von 1 bis zur Länge der Mauer minus 1, da, wie bereits erwähnt, die Position am Ende nicht als Fuge zählt. Die Länge der Mauer ist die Summe aller natürlichen Zahlen bis n, also nach der Gaußschen Summenformel $\frac{n\cdot(n+1)}{2}$.

Teilt man nun die Anzahl der insgesamt verfügbaren Fugenpositionen durch die Anzahl der Fugen pro Reihe, erhält man die mögliche Anzahl an Reihen, also die Höhe.

$$H(n) = \frac{\frac{n\cdot(n+1)}{2} - 1}{n-1} = \frac{(n+2)\cdot(n-1)}{2\cdot(n-1)} = \lfloor\frac{n+2}{2}\rfloor \tag{1}$$

Die Abrundungsfunktion ist an dieser Stelle nötig, da sonst für ungerade n das Resultat keine natürliche Zahl ist. Halbe Reihen jedoch ergeben wenig Sinn. Die angegebene Funktion $H(n)$ ist für alle natürlichen Zahlen $n \geq 2$ definiert.

Der Fall $n = 1$ ist damit nicht berechenbar, da in der Originalgleichung durch 0 geteilt würde. Dieser muss auch ausgeschlossen werden, da es dabei keine Fugen im Sinne von Stellen zwischen den Klötzen gibt. Somit ist nicht eindeutig definiert, ob in diesem Fall nur eine Reihe (mit einem Klotz der Länge 1) oder unendlich viele Reihen erlaubt sind.

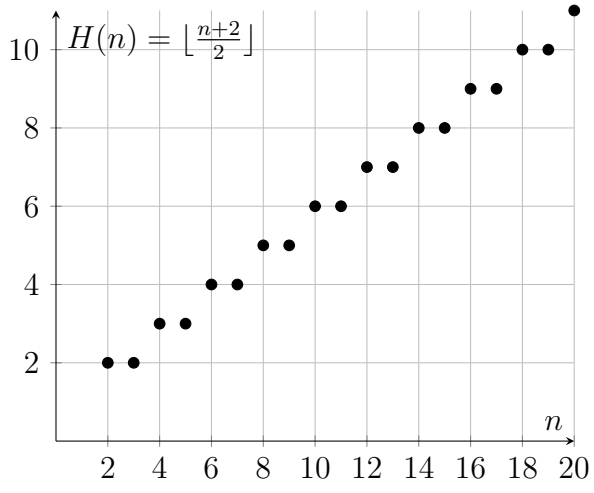

Abbildung 1: Werte für n = 2,...,20

1.2 Lösungsansatz: Backtracking

Zunächst kann man davon ausgehen, dass es für jedes beliebige n einer Mauer dieser Maximalhöhe gibt. Nun muss man diese nur noch finden. Der einfachste mögliche Ansatz für dieses Problem wäre es, in jeder Reihe alle Anordnungen der Klötze auszuprobieren. Da es jedoch pro Reihe $n!$ Permutationen der Klötze gibt, ist dieser Brute-Force-Lösungsansatz mit einer Laufzeit in $\mathcal{O}(n!^{\lfloor \frac{n+2}{2} \rfloor})$ gänzlich inakzeptabel.

Es wird also ein effizienterer Lösungsalgorithmus für das Problem benötigt. Dabei tritt jedoch das Problem auf, dass es, wie im nächsten Abschnitt besprochen, höchstwahrscheinlich keinen effizienten Lösungsalgorithmus mit polynomialer Laufzeit gibt. Also muss versucht werden, die Brute-Force-Suche effizienter zu gestalten. Ein guter Weg dazu ist rekursives Backtracking.

Zunächst wird für Backtracking ein Lösungsvektor benötigt. In diesem Fall kann das ein zweidimensionales Array sein, dessen erste Dimension die Reihe darstellt und dessen zweite Dimension für jede Reihe von links nach rechts die verwendeten Klötze angibt. Backtracking stellt dabei eine Tiefensuche auf einem impliziten Baum dar. Die Knoten des genannten Baumes stellen dabei die Zustände des Lösungsvektors dar und es existiert eine Kante zwischen Zuständen, die durch Hinzufügen eines Klotzes an das Ende des aktuellen Lösungsvektors auseinander hervorgehen. Die Wurzel des Baumes, an der die Backtracking-Suche begonnen wird, stellt ein leerer Lösungsvektor dar. Die Suche versucht nun, bis in die tiefste Ebene des Baumes vorzudringen.

Beim Backtracking wird nun also Klotz für Klotz eine Lösung aufgebaut. Dabei wird jeweils der längste in dieser Reihe noch nicht verwendete und an dieser Stelle noch nicht ausprobierte Klotz gewählt. Sobald jedoch sicher ist, dass eine solche Teillösung nicht zu einer Gesamtlösung ausgebaut werden kann, kommt das sogenannte Pruning zum Einsatz. Da es sinnlos ist, von einer solchen Teillösung aus weiter in die Tiefe zu gehen, kann man die Wahl des Klotzes auf der aktuellen Stufe rückgängig machen und in die vorherige Stufe zurückwechseln. Dort werden dann alle weiteren aktuellen Wahlmöglichkeiten des Klotzes nach dem gleichen Prinzip durchprobiert.

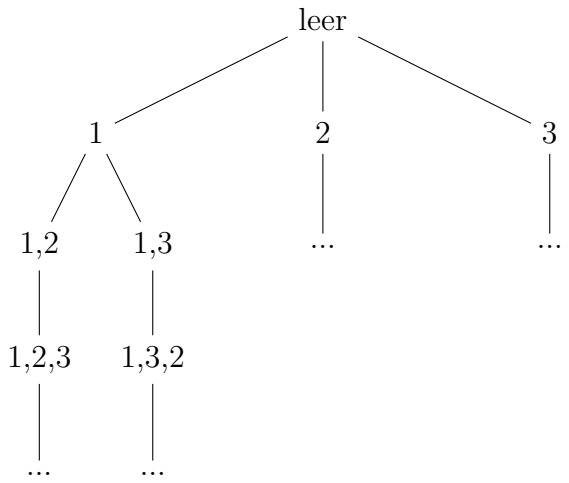

Abbildung 2: Ausschnitt des Rekursionsbaums für $n = 3$ ohne festgelegte erste Reihe

Ein solcher Algorithmus findet in jedem Fall eine maximal hohe Mauer, sofern es eine solche gibt. Es handelt sich hierbei nämlich nur um eine intelligentere Variante der

Brute-Force-Suche. Wichtig für die Funktionsfähigkeit dieses Algorithmus ist, dass die Höhe der Mauer durch einen Maximalwert begrenzt ist und wir somit wissen, wann eine vollständige Lösung erreicht ist. Sobald eine solche erreicht ist, kann die Suche auch abgebrochen werden, da nur eine und nicht mehrere Mauern mit maximaler Höhe gesucht sind.
Nun ist es noch wichtig, Pruning-Kriterien so zu definieren, dass möglichst viele nicht ausbaubaren Teillösungen möglichst früh gefunden und solche Teillösungen verworfen werden. Bei geeigneten Kriterien lässt sich nämlich die Laufzeit im Vergleich zu Brute Force erheblich verbessern.

Pruning-Kriterien

Es lassen sich zunächst zwei Grundannahmen für die ersten beiden Reihen der Mauer treffen, wodurch diese nicht durch Backtracking ermittelt werden müssen.

1. Die erste Reihe der Mauer ist im Format $(1, 2, 3, ..., n)$ aufgebaut. Diese Annahme kann gefahrlos getroffen werden, da sich alle anderen Reihen dann danach ausrichten. Dieses Vorgehen führt in der Praxis auch zu richtigen Ergebnissen.

2. Die zweite Reihe der Mauer ist meistens im Format $(2, 3, ..., n, 1)$ aufgebaut. Diese Annahme bestätigt sich, wenn man den Algorithmus unter Verwendung der ersten Annahme implementiert. Lediglich für $n = 6$ und $n = 10$ führt diese Annahme nicht zu gültigen Mauern und darf deshalb dort nicht verwendet werden.

Nun kann man zwei Pruning-Kriterien festlegen.

1. *Das offensichtliche Kriterium:* Gibt es an einer Stelle keine Möglichkeit mehr, einen in dieser Reihe noch nicht verwendeten Klotz so zur Mauer hinzuzufügen, dass keine Fuge doppelt belegt wird, so ist mit der bisherigen Mauer keine Lösung zu erreichen. Somit kann man hier prunen und auf die nächsthöhere Ebene zurückkehren.

2. *Das erweiterte Kriterium, Teil 1:* Befindet man sich noch nicht in der letzten Reihe und bemerkt, dass zwischen zwei benachbarten noch nicht belegten Fugenpositionen[1] der Abstand größer als n ist, so gibt es ebenfalls keine Lösung mit der bisherigen Mauer. Also kann ebenfalls geprunt werden, da kein genügend langer Klotz zur Überwindung dieses Abstands existieren kann.

3. *Das erweiterte Kriterium, Teil 2:* Befindet man sich noch nicht in der letzten Reihe und bemerkt, dass nach der Position, an der die aktuelle Reihe bisher endet, ein ebensolcher Abstand besteht, der größer als der längste in dieser Reihe noch nicht verwendete Klotz ist, ist Pruning ebenfalls möglich. In dieser Reihe müsste dieser Abstand nämlich noch überwunden werden, was jedoch keinesfalls möglich ist, da keine genügend langen Klötze mehr vorhanden sind.

1.3 Laufzeitanalyse und NP-Vollständigkeit

Die Laufzeit einer Backtracking-Lösung entspricht im schlimmsten Fall der Laufzeit einer Brute-Force-Lösung für das entsprechende Problem. Theoretisch ist die Laufzeitkomplexität hier also nur durch $\mathcal{O}(n!^{\lfloor\frac{n+2}{2}\rfloor})$ begrenzt. In der Realität wird die Laufzeit jedoch durch das Pruning in vertretbarem Rahmen gehalten. Mindestens bis $n = 14$ ist beispielsweise in der unten vorgestellten Implementierung eine Berechnung innerhalb einer Minute

[1] Hier zählen auch 0 und $\frac{n\cdot(n+1)}{2}$ als Fugenpositionen im erweiterten Sinne.

möglich, danach steigen die Laufzeiten bis ins Unendliche. Ein Überblick sowohl über die Laufzeit als auch über die Anzahl der benötigten Rekursionsaufrufe für verschiedene n findet sich unter den Beispielen.

Eine Frage, die sich hierbei auch stellt, ist diejenige, ob es für dieses Problem[2] einen in Polynomialzeit terminierenden Algorithmus geben kann. Dies entspricht der Frage, ob das Problem in der Klasse NPC[3] (NP-vollständig bzw. NP-complete) liegt. Ich vermute, dass dies der Fall ist, kann es jedoch nicht beweisen.

Zum Beweis, dass ein Problem in NPC liegt, werden zwei Voraussetzungen benötigt:

1. Eine deterministisch arbeitende Turingmaschine benötigt nur Polynomialzeit, um zu entscheiden, ob eine z.B. von einer Orakel-Turingmaschine vorgeschlagene Lösung tatsächlich eine Lösung des Problems ist. Dies ist hier der Fall, denn wenn eine Lösung vorgeschlagen wird, sind durch die Angabe aller Klötze auch alle Fugenpositionen bestimmt. Findet man eine Fugenposition mehr als einmal, so ist dies keine Lösung. Sonst handelt es sich um eine Lösung. Diese Überprüfung kann in $\mathcal{O}(n \cdot \lfloor \frac{n+2}{2} \rfloor) \approx \mathcal{O}(n^2)$, also in Polynomialzeit, vorgenommen werden.

2. Das Problem ist NP-schwer. Das bedeutet, dass alle anderen NP-schweren Probleme auf dieses Problem in Polynomialzeit zurückgeführt werden können. Es ist also eine Polynomialzeitreduktion notwendig. Dabei ist ein Problem aus NPC als Ausgangsproblem nötig, wie z.B. Satisfiability. Eine solche Reduktion zu vollziehen, ist mir jedoch nicht möglich.

1.4 Weitere Möglichkeiten zur Verbesserung der Laufzeit

Dieser Abschnitt soll weitere denkbare Lösungsansätze nennen und zeigen, warum diese für das Mauerproblem ungeeignet sind und von mir deshalb nicht gewählt wurden.

- *Der Greedy-Ansatz:* Dieser bestünde daraus, für jeden angelegten Klotz einfach den größten möglichen Klotz zu nehmen, der keine Fugenposition doppelt belegt. Alle vollständigen Reihen, die dabei entstehen, würden dann die Mauer bilden. Es sollte klar sein, dass dieser Ansatz keinesfalls immer eine maximal hohe Mauer liefern kann.

- *Heuristiken:* Probleme, die sich sonst nur durch Backtracking lösen lassen, versucht man oft mittels Heuristiken wie z.B. Simulated Annealing schneller zu lösen. Dies ist hier jedoch nur schwierig möglich, denn dafür benötigt man einen Lösungsraum, der hier aus allen möglichen Mauern der Maximalhöhe (unabhängig davon, ob diese die Anforderungen erfüllen) bestehen könnte. Dann benötigte man eine Bewertungsfunktion, die hier der Anzahl der doppelt belegten Fugenpositionen entsprechen könnte. Diese wäre dann zu minimieren. Dann wäre hier noch eine Idee nötig, wie man aus einer möglichen Lösung eine andere gewinnen könnte. Man könnte dazu beispielsweise zwei beliebige Klötze in einer beliebigen Reihe miteinander vertauschen.

  Aus welchen Gründen jedoch ist dieser Ansatz nicht geeignet? Zunächst kann man nicht verhindern, dass mögliche Lösungen doppelt betrachtet werden, wenn man

[2] nach aktuellem Kenntnisstand ($P = NP$?)

[3] Genaugenommen ist diese Klasse nur für Entscheidungsprobleme definiert, daher handelt es sich bei diesem Suchproblem um NP-Äquivalenz.

keinen enormen Speicherverbrauch in Kauf nehmen möchte. Außerdem kann man sich nicht sicher sein, dass die bisher beste Lösung (nach i Iterationen des Simulated Annealing) tatsächlich eine gültige Lösung (mit 0 doppelten Fugen) ist. Zuletzt betrachtet man schlimmstenfalls genau so viele Lösungsmöglichkeiten (oder gar mehr) wie bei Brute Force.

- *Weitere Pruning-Kriterien:* Grundsätzlich könnten weitere Pruning-Kriterien die Laufzeit im besten Fall dramatisch verbessern. Alle weiteren Pruning-Kriterien, die ich gefunden habe, brachten jedoch keine Verbesserung oder sogar eine Verschlechterung. Als Beispiel sei hier ein Kriterium genannt. Es ist offensichtlich, dass sich in einer richtigen Lösung alle Reihen untereinander vertauschen lassen und die Lösung trotzdem richtig bleibt.

  Umgekehrt gilt auch: Wenn eine Reihe[4] oder ein Teil einer Reihe beim Backtracking nicht zu einer Lösung führen, dann können diese an anderen Stellen in der Mauer ebenfalls nicht zu einer Lösung führen. Dies gilt zumindest solange, bis eine der Fugen, die die weitere Fortsetzung der Reihe blockiert haben (da sie belegt waren), wieder freigegeben wird. Diese nicht zu einer Lösung führenden (Teil-)Reihen könnte man als Pruning-Kriterium festlegen. Implementiert man dieses Kriterium jedoch, so verschlechtert es sowohl Laufzeit als auch Speicherverbrauch.

Möglicherweise reicht es Ilona jedoch, wenn ihre Mauer für größere n nicht maximal hoch ist, sie dafür aber wenigstens eine weniger hohe Mauer geliefert bekommt. Dazu kann man das Backtracking ganz einfach vorzeitig, also bei Erreichen einer geringeren Höhe, abbrechen.

In der nachfolgenden Implementierung ist dies so geregelt, dass eine wählbare Anzahl von Reihen (diese muss bei jedem n experimentell bestimmt werden) von der Maximalhöhe abgezogen wird. Sobald diese reduzierte Höhe erreicht ist, wird die Lösung ausgegeben. Insofern versuche ich mit diesem Schritt, eine möglichst hohe Mauer zu approximieren und gleichzeitig die Laufzeit erträglich zu halten.

Literatur

[1] Wikipedia-Artikel zur NP-Vollständigkeit, `https://de.wikipedia.org/wiki/NP-Vollst%C3%A4ndigkeit`

[2] Wikipedia-Artikel zu Backtracking, `https://de.wikipedia.org/wiki/Backtracking`

[3] Steven S. Skiena: The Algorithm Design Manual, ISBN 978-1-84800-069-8, Kapitel 7 beschäftigt sich mit Backtracking (und Heuristiken), Kapitel 9 mit NP-Vollständigkeit

[4] Im Sinne einer Folge von Klötzen, welche entsprechend unabhängig von der genauen Positionierung in der Mauer ist.

2 Umsetzung

2.1 Allgemeine Hinweise zur Benutzung

Das Programm ist in Java implementiert und wurde mit Java 9.0.4 kompiliert. Sofern bei der Bewertung eine ältere Java-Version ausgeführt wird, müsste also neu kompiliert werden. Auf der obersten Ebene des Implementierungs-Ordners befindet sich eine JAR-Datei, mit der das Programm auf der Kommandozeile ausgeführt werden kann (`java -jar KunstDerFuge.jar`). Die `*.class`-Datei befindet sich im Ordner `out/production`.

Als Eingabe erwartet das Programm n und gegebenenfalls die Anzahl an Reihen, um die die Höhe reduziert werden soll. 0 entspricht dabei der Maximalhöhe.

2.2 Umsetzung des Backtrackings

Das Programm besteht aus vier Funktionen in der Klasse `de.lukasrost.bwinf2017.r2_1.Main`. Gestartet wird das Programm über die Funktion `main()`. Diese fragt die Eingabeparameter ab und gibt schließlich die Mauer aus. Zusätzlich werden unzulässige n (< 2) abgewiesen und die Zeit für die Ausführung des Algorithmus gestoppt.

Die Funktion `getSolution()` bereitet das Backtracking vor. Dazu werden die ersten beiden Reihen (bzw. bei $n = 6$ oder $n = 10$ die erste Reihe) im zweidimensionalen Mauer-Array[5] vorbelegt. Weiterhin werden die anderen von der Backtracking-Funktion genutzten Datenstrukturen initialisiert. Dies sind:

- Der boolesche Bitvektor `usedFugen` speichert, welche Fugenpositionen schon belegt sind. `usedFugen[`i`]` gibt an, ob die Fuge i bereits belegt ist. Zu Beginn ist keine Fuge genutzt, anschließend werden die bereits genutzten Fugen der ersten beiden Reihen gespeichert. Die Nummerierung der Fugen folgt dabei der unten zu sehenden Abbildung.
- Der ebenfalls boolesche Bitvektor `usableKloetze` gibt an, welche der Klötze von 1 bis n an der aktuellen Position nutzbar sind. Am Anfang einer Reihe sind jeweils alle Klötze nutzbar, beim zweiten Klotz einer Reihe dann alle Klötze außer der als erster Klotz ausgewählte Klotz, usw.

Die Array-Position 0 hat dabei jeweils keine Bedeutung und bleibt ungenutzt. In beiden Fällen wurden Bitvektoren verwendet, da diese sehr laufzeit- und speichereffizient sind, was bei der Lösung von NP-Problemen enorm hilfreich sein kann und die Laufzeit verkürzt.

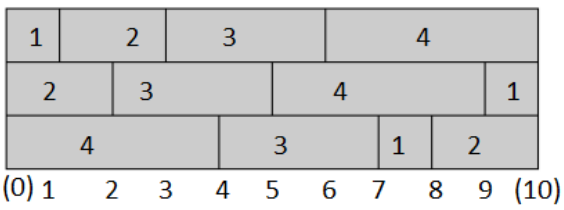

Abbildung 3: Nummerierung der Klötze und Fugen

[5] Erste Dimension ist die Reihe, zweite Dimension der Klotz in der Reihe.

`backTrack()` ist die für das Backtracking verantwortliche Funktion. Die Parameter der Funktion sind größtenteils selbsterklärend. Zu diesen wäre nur noch zu sagen, dass `height` die Höhe der gewünschten Mauer angibt und `reihe` bzw. `klotzInReihe` das aktuell zu belegende Element der Lösung, wobei hier natürlich von 0 aus gezählt wird.

Zuerst inkrementiert die Funktion die Klassenvariable `recursionCount`, die die Anzahl der Rekursionsaufrufe angibt. Wenn nun eine Reihe als Parameter angegeben wurde, die höher als die gewünschte Höhe ist, so kann die bisherige Mauer ungeändert zurückgegeben werden. Sonst wird die Summe aller Klötze in der aktuellen Reihe, die vor dem aktuellen Klotz stehen, berechnet. Anschließend werden die `usableKloetze` für den nächsten Rekursionsaufruf kopiert, da sie nun spezifisch für die aktuelle Position so verändert werden, dass nur noch diejenigen Klötze nutzbar sind, die keine Fuge doppelt belegen würden. Zusätzlich werden die erweiterten Pruning-Kriterien geprüft und entsprechend, falls diese erfüllt sind, in die nächsthöhere Rekursionsebene zurück gewechselt.

Anschließend werden alle jetzt noch nutzbaren Klötze in absteigender Reihenfolge durchprobiert. Dazu wird der entsprechende Klotz in der Lösung gesetzt und `usableKloetze` sowie `usedFugen` und `usableKloetzeNextEbene`, also die auf der nächsten Rekursionsebene nutzbaren Klötze, entsprechend geupdatet. Ist die Mauer nun vollständig, kann sie zurückgegeben werden. Sonst werden die Parameter für den nächsten Rekursionsaufruf vorbereitet (beim Wechsel in eine neue Reihe sind dann z.B. wieder alle Klötze nutzbar) und dieser schließlich gestartet.

Nun gibt es drei Möglichkeiten für das Ergebnis dieses Aufrufs:

1. Pruning durch das einfache Kriterium (`IMPOSSIBLE`[6]): Dann werden die Datenstrukturen entsprechend zurückgesetzt und mit dem nächsten möglichen Klotz auf der aktuellen Ebene fortgefahren.

2. Pruning durch die erweiterten Kriterien(`IMPOSSIBLEPARENT`): Die Besonderheit bei den erweiterten Kriterien ist, dass sie darauf hinweisen, dass die Entscheidung auf der Ebene direkt über der Ebene, die dies zurückgegeben hat, fehlerhaft war. Dementsprechend wird die gesamte Durchprobier-Schleife abgebrochen.

3. Die Ebene unter der aktuellen hat eine korrekte Lösung gefunden. Diese wird zurückgegeben.

Hat die ganze Schleife keinen passenden Klotz gefunden, wird `IMPOSSIBLE` zurückgegeben, d.h. dass es auf der aktuellen Ebene überhaupt keine passende Klotz-Wahl gibt.

Zuletzt ist `checkPruning()` dafür verantwortlich, die erweiterten Pruning-Kriterien zu überwachen. Dazu werden zunächst alle nicht belegten Fugen in einer Liste gespeichert. Außerdem wird der größte aktuell noch verfügbare Klotz für den zweiten Teil des Kriteriums bestimmt. Anschließend wird der erste Teil des Kriteriums überprüft, indem der Abstand von nebeneinanderliegenden nicht belegten Fugen ermittelt wird. Daraufhin wird in ähnlicher Weise der zweite Teil des Kriteriums überprüft, wobei dies natürlich erst ab der aktuellen Klotzsumme in der aktuellen Reihe geschieht.

[6]Sollten Sie sich wundern, weshalb diese Konstanten in ein Array verpackt werden (siehe Quellcode): Es liegt an der Datentypenkonsistenz der Rückgabewerte. Wir sind hier schließlich nicht bei Python oder gar PHP.

3 Beispiele

Lösung für n = 2:

1	2
2	1

Lösung für n = 3:

1	2	3
2	3	1

Lösung für n = 4:

1	2	3	4
2	3	4	1
4	3	1	2

Lösung für n = 5:

1	2	3	4	5
2	3	4	5	1
4	3	5	1	2

Lösung für n = 6:

1	2	3	4	5	6
5	6	3	2	4	1
4	3	6	5	1	2
2	6	1	3	5	4

Lösung für n = 7:

1	2	3	4	5	6	7
2	3	4	5	6	7	1
7	6	5	4	2	1	3
4	7	5	1	6	3	2

Lösung für n = 8:

1	2	3	4	5	6	7	8
2	3	4	5	6	7	8	1
8	5	6	7	3	1	2	4
7	5	6	4	2	1	8	3
4	7	5	1	6	8	3	2

Lösung für n = 9:

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

Lösung für n = 10:

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

Lösung für n = 11:

1	2	3	4	5	6	7	8	9	10	11
2	3	4	5	6	7	8	9	10	11	1
11	8	10	9	5	7	6	4	3	1	2
8	10	7	9	6	11	2	5	3	1	4
7	10	6	8	11	4	1	2	3	5	9
4	8	1	3	10	6	5	2	9	11	7

Lösung für n = 12:

1	2	3	4	5	6	7	8	9	10	11	12
2	3	4	5	6	7	8	9	10	11	12	1
12	11	10	9	8	7	6	5	4	3	1	2
11	8	12	10	7	5	9	2	6	3	1	4
8	10	12	4	3	1	5	6	2	9	11	7
7	10	5	3	1	6	8	12	4	2	11	9
4	9	3	8	5	10	7	1	12	2	6	11

Lösung für n = 13:

1	2	3	4	5	6	7	8	9	10	11	12	13
2	3	4	5	6	7	8	9	10	11	12	13	1
13	12	9	8	11	10	7	6	5	4	3	1	2
12	11	10	13	6	9	8	4	7	3	1	2	5

```
| 11 | 13 | 8 | 9 | 10 | 7 | 6 | 3 | 5 | 2 | 1 | 12 | 4 |
| 8 | 10 | 13 | 7 | 5 | 4 | 3 | 6 | 1 | 2 | 12 | 11 | 9 |
| 7 | 10 | 5 | 4 | 3 | 8 | 2 | 9 | 1 | 13 | 6 | 11 | 12 |

Lösung für n = 14:
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |
| 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 1 |
| 13 | 12 | 14 | 11 | 10 | 9 | 7 | 8 | 5 | 6 | 4 | 3 | 1 | 2 |
| 12 | 14 | 11 | 10 | 9 | 8 | 7 | 4 | 13 | 6 | 3 | 1 | 2 | 5 |
| 11 | 13 | 14 | 10 | 9 | 6 | 7 | 4 | 5 | 3 | 1 | 2 | 8 | 12 |
| 8 | 11 | 3 | 10 | 2 | 6 | 1 | 12 | 9 | 5 | 13 | 7 | 14 | 4 |
| 7 | 11 | 5 | 6 | 1 | 12 | 4 | 3 | 2 | 8 | 13 | 14 | 10 | 9 |
| 4 | 12 | 1 | 14 | 2 | 10 | 9 | 6 | 3 | 7 | 5 | 8 | 11 | 13 |

-- ab hier mit reduzierter Höhe --

Lösung für n = 15:
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
| 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 1 |
| 13 | 12 | 15 | 11 | 10 | 14 | 9 | 8 | 7 | 4 | 6 | 5 | 3 | 1 | 2 |
| 12 | 14 | 15 | 11 | 10 | 9 | 8 | 7 | 3 | 13 | 6 | 5 | 2 | 1 | 4 |
| 11 | 13 | 15 | 14 | 10 | 9 | 4 | 7 | 12 | 5 | 1 | 6 | 3 | 2 | 8 |
| 8 | 15 | 14 | 12 | 11 | 10 | 4 | 7 | 6 | 1 | 5 | 3 | 2 | 13 | 9 |
| 7 | 15 | 11 | 10 | 5 | 8 | 2 | 6 | 4 | 1 | 13 | 12 | 3 | 9 | 14 |

Lösung für n = 16:
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |
| 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 1 |
| 16 | 15 | 12 | 14 | 13 | 11 | 8 | 10 | 9 | 7 | 6 | 5 | 4 | 3 | 1 | 2 |
| 13 | 16 | 12 | 15 | 11 | 9 | 10 | 14 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 8 |
| 12 | 14 | 16 | 11 | 15 | 7 | 13 | 10 | 8 | 6 | 2 | 9 | 1 | 5 | 3 | 4 |
| 11 | 14 | 13 | 12 | 10 | 9 | 16 | 8 | 3 | 1 | 4 | 2 | 7 | 6 | 15 | 5 |

Lösung für n = 17:
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |
↪ 17 |
| 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17
↪ | 1 |
| 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 1
↪ | 2 |
| 16 | 15 | 12 | 17 | 14 | 11 | 10 | 8 | 13 | 7 | 6 | 5 | 3 | 9 | 2 | 1
↪ | 4 |
| 13 | 17 | 16 | 15 | 12 | 11 | 10 | 8 | 7 | 6 | 3 | 4 | 5 | 14 | 1 | 2
↪ | 9 |
| 12 | 17 | 13 | 16 | 14 | 11 | 10 | 7 | 6 | 15 | 3 | 4 | 2 | 1 | 9 | 5
↪ | 8 |
| 11 | 15 | 13 | 17 | 12 | 8 | 10 | 2 | 1 | 3 | 5 | 4 | 9 | 16 | 7 | 6 |
↪ 14 |

Lösung für n = 18:
```

```
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |
↪  17 | 18 |
| 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17
↪  | 18 | 1 |
| 18 | 16 | 17 | 13 | 15 | 14 | 10 | 12 | 11 | 8 | 9 | 7 | 6 | 5 | 4 | 3
↪  | 1 | 2 |
| 17 | 16 | 15 | 14 | 18 | 12 | 10 | 11 | 9 | 8 | 7 | 5 | 13 | 3 | 6 | 2
↪  | 1 | 4 |
| 16 | 15 | 18 | 14 | 13 | 12 | 11 | 17 | 9 | 7 | 6 | 3 | 5 | 2 | 1 | 10
↪  | 4 | 8 |
| 13 | 17 | 16 | 15 | 14 | 12 | 11 | 10 | 6 | 7 | 3 | 5 | 2 | 8 | 1 | 4
↪  | 18 | 9 |
| 12 | 17 | 18 | 13 | 10 | 16 | 9 | 5 | 7 | 2 | 8 | 1 | 15 | 14 | 4 | 6
↪  | 3 | 11 |
| 11 | 15 | 16 | 14 | 13 | 3 | 2 | 8 | 1 | 6 | 7 | 10 | 5 | 12 | 4 | 18
↪  | 9 | 17 |

Lösung für n = 19:
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |
↪  17 | 18 | 19 |
| 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17
↪  | 18 | 19 | 1 |
| 19 | 18 | 16 | 17 | 15 | 14 | 13 | 12 | 10 | 11 | 9 | 8 | 7 | 6 | 5 |
↪  4 | 3 | 1 | 2 |
| 18 | 16 | 17 | 13 | 19 | 15 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 2 |
↪  14 | 3 | 1 | 4 |
| 17 | 16 | 19 | 15 | 14 | 13 | 12 | 11 | 10 | 6 | 18 | 9 | 5 | 8 | 4 |
↪  1 | 3 | 2 | 7 |
| 16 | 15 | 19 | 18 | 14 | 13 | 12 | 9 | 10 | 17 | 6 | 1 | 7 | 2 | 8 | 5
↪  | 4 | 3 | 11 |
| 13 | 19 | 17 | 14 | 12 | 18 | 15 | 10 | 7 | 4 | 8 | 1 | 6 | 3 | 9 | 2
↪  | 5 | 11 | 16 |

Lösung für n = 20:
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |
↪  17 | 18 | 19 | 20 |
| 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17
↪  | 18 | 19 | 20 | 1 |
| 19 | 20 | 18 | 17 | 15 | 14 | 13 | 16 | 12 | 11 | 10 | 9 | 8 | 6 | 7 |
↪  5 | 4 | 3 | 1 | 2 |
| 18 | 20 | 15 | 19 | 16 | 14 | 13 | 12 | 11 | 10 | 9 | 7 | 17 | 6 | 5 |
↪  4 | 3 | 2 | 1 | 8 |
| 17 | 20 | 19 | 15 | 16 | 14 | 13 | 12 | 11 | 10 | 9 | 7 | 6 | 8 | 3 |
↪  18 | 5 | 2 | 1 | 4 |
| 16 | 18 | 17 | 19 | 15 | 14 | 12 | 20 | 11 | 9 | 10 | 7 | 8 | 3 | 6 |
↪  1 | 5 | 2 | 4 | 13 |
| 13 | 20 | 19 | 17 | 15 | 14 | 12 | 18 | 11 | 7 | 8 | 6 | 2 | 4 | 9 | 3
↪  | 5 | 1 | 10 | 16 |
```

Bei denjenigen Werten für n, bei denen aus Laufzeitgründen eine Höhenreduzierung notwendig war, ist in der folgenden Tabelle jeweils der erste Höhenreduzierungswert angegeben, bei dem die Laufzeit unter 5 Minuten lag.

n	Laufzeit (ca.)	Rekursionsaufrufe	Höhenred.	tats. Höhe	Max.-höhe
2	1 ms	1	-	2	2
3	1 ms	1	-	2	2
4	1 ms	4	-	3	3
5	1 ms	5	-	3	3
6	2 ms	160	-	4	4
7	3 ms	34	-	4	4
8	11 ms	1873	-	5	5
9	1 ms	168	-	5	5
10	217 ms	139627	-	6	6
11	5 ms	445	-	6	6
12	364 ms	322697	-	7	7
13	250 ms	89340	-	7	7
14	54531 ms	57291649	-	8	8
15	60818 ms	51658453	1	7	8
16	340 ms	1877989	3	6	9
17	523 ms	1552368	2	7	9
18	448 ms	558143	2	8	10
19	590 ms	2701066	3	7	10
20	772 ms	3237947	4	7	11

Für größere n wurde das Programm nicht ausprobiert, da bei diesen die Laufzeiten noch problematischer werden als ohnehin schon. Es lässt sich feststellen, dass bei einer Höhe von 7 bis 8 Reihen eine „magische Grenze“ zu liegen scheint, ab der die Laufzeiten unabhängig von n inakzeptabel werden. Es ist denkbar, dass möglicherweise keine höheren Mauern, die die Bedingungen erfüllen, existieren, was ich jedoch nicht verifizieren kann.

4 Quellcode

```
1  package de.lukasrost.bwinf2017.r2_1;
2
3  import java.util.*;
4
5  public class Main {
6      private final static int IMPOSSIBLE = -1; //Rückgabekonstanten
7      private final static int IMPOSSIBLEPARENT = -2;
8      private static int recursionCount = 0; //Rekursionsaufrufe
9
10     public static void main(String[] args) {
11         System.out.print("Bitte n der Mauer eingeben: ");
12         Scanner scanner = new Scanner(System.in);
13         int n = scanner.nextInt();
14         if (n < 2){
15             System.out.println("n ist zu klein. Keine Lösung möglich!");
16             System.exit(0);
17         }
18         System.out.print("Um wie viele Reihen soll die Höhe reduziert werden?
           ↪ (0=Maximalhöhe):");
```

```
19          int reihen = scanner.nextInt();
20          System.out.println("Starte Backtracking...");
21          long beforeTime = System.currentTimeMillis();
22          int[][] arr = getSolution(n, reihen);
23          long afterTime = System.currentTimeMillis();
24          System.out.println("Lösung: ");
25          for (int[] anArr : arr){
26              System.out.print(" | ");
27              for (int anAnArr : anArr) {
28                  System.out.print(anAnArr + " | ");
29              }
30          System.out.println();
31          }
32          System.out.println();
33          System.out.println("Benötigte Zeit: " + (afterTime-beforeTime) + "
            ↪  Millisekunden");
34          System.out.println("Rekursionsaufrufe: "+ recursionCount);
35      }
36
37      private static int[][] getSolution(int n, int reihen){
38          int height = ((n+2)/2) - reihen;
39          int[][] solution = new int[height][n]; //Lösungsvektor initialisieren
40          for (int z = 1; z <= n; z++){
41              solution[0][z-1] = z;
42              if (n != 6 && n != 10) {
43                  solution[1][z - 1] = z + 1;
44              }
45          }
46          solution[1][n-1] = 1;
47          boolean[] usedFugen = new boolean[(n*(n+1)/2)+1]; //Fugenliste
            ↪  initialisieren
48          Arrays.fill(usedFugen,false);
49          int fuge = 0;
50          for (int z = 1; z < n; z++){
51              fuge += z;
52              usedFugen[fuge] = true;
53          }
54          if (n != 6 && n!= 10) {
55              int fuge2 = 0;
56              for (int z = 2; z <= n; z++) {
57                  fuge2 += z;
58                  usedFugen[fuge2] = true;
59              }
60          }
61          boolean[] usableKloetzeNextEbene = new boolean[n+1]; //nutzbare
            ↪  Kloetze
62          Arrays.fill(usableKloetzeNextEbene, true);
63          int t = (n == 6 || n== 10) ? 1 : 2; //Startreihe festlegen
64          solution = backTrack(n, solution,t,0,usableKloetzeNextEbene,usedFugen,
            ↪  height);
65          return solution;
66      }
67
```

```
68      private static int[][] backTrack(int n, int[][] mauerBefore, int reihe,
        ↪  int klotzInReihe, boolean[] usableKloetze, boolean[] usedFugen, int
        ↪  height){
69          recursionCount++;
70          if (reihe > (height-1)) return mauerBefore;
71          int summe = 0; //vorherige Summe berechnen
72          for (int i = 0; i < klotzInReihe; i++){
73              summe += mauerBefore[reihe][i];
74          }
75          boolean[] usableKloetzeNextEbene =
            ↪  Arrays.copyOf(usableKloetze,usableKloetze.length); //Kloetze fuer
            ↪  Rekursion
76          for (int klotz=1; klotz<usableKloetze.length; klotz++) {
77              if (!usableKloetze[klotz]) continue;
78              if (usedFugen[summe + klotz] && ((summe + klotz) != ((n * (n + 1))
                ↪  / 2))) {
79                  usableKloetze[klotz] = false; //Klotz an dieser Position nicht
                    ↪  nutzbar
80              }
81          }
82          if (reihe != (height-1) && checkPruning(usedFugen, n,
            ↪  usableKloetzeNextEbene, summe)){
83              return new int[][]{{IMPOSSIBLEPARENT}}; //erweitertes Pruning
                ↪  checken
84          }
85          label:
86          for (int uk = usableKloetze.length - 1; uk >= 1; uk--) {
87              if (!usableKloetze[uk]) continue;
88              mauerBefore[reihe][klotzInReihe] = uk; //Datenstrukturen updaten
89              usableKloetzeNextEbene[uk] = false;
90              usableKloetze[uk] = false;
91              if (summe + uk != ((n) * (n + 1)) / 2) {
92                  usedFugen[summe + uk] = true;
93              }
94              if (reihe == (height - 1) && klotzInReihe == n - 1) {
95                  return mauerBefore; //Mauer vollstaendig
96              }
97              int reiheneu = reihe; //Rekursionsaufruf vorbereiten
98              int klotzInReiheNeu = klotzInReihe + 1;
99              boolean[] usableKloetzeNextEbene2 = new boolean[n + 1];
100             if (klotzInReihe == n - 1) {
101                 reiheneu = reihe + 1;
102                 klotzInReiheNeu = 0;
103                 Arrays.fill(usableKloetzeNextEbene2, true);
104             } else {
105                 usableKloetzeNextEbene2 =
                    ↪  Arrays.copyOf(usableKloetzeNextEbene,
                    ↪  usableKloetzeNextEbene.length);
106             }
107             int[][] mauerDanach = backTrack(n, mauerBefore, reiheneu,
                ↪  klotzInReiheNeu, usableKloetzeNextEbene2, usedFugen, height);
108             switch (mauerDanach[0][0]) { //Ergebnis auswerten
109                 case IMPOSSIBLE:
```

```
110                     usedFugen[summe + uk] = false;
111                     usableKloetzeNextEbene[uk] = true;
112                     mauerBefore[reihe][klotzInReihe] = 0;
113                     break;
114                 case IMPOSSIBLEPARENT:
115                     usedFugen[summe + uk] = false;
116                     usableKloetzeNextEbene[uk] = true;
117                     mauerBefore[reihe][klotzInReihe] = 0;
118                     break label;
119                 default:
120                     return mauerDanach;
121             }
122         }
123         return new int[][]{{IMPOSSIBLE}}; //auf dieser Ebene keine Loesung
124     }
125 
126     private static boolean checkPruning(boolean[] usedFugen, int n, boolean[]
        ↪ kloetze, int summe){
127         int k = n *(n+1) /2;
128         ArrayList<Integer> nichtBelegt = new ArrayList<>();
129         for (int i = 0; i <= k; i++) {
130             if (usedFugen[i]) continue;
131             nichtBelegt.add(i); // noch nicht belegte Fugen
132         }
133         int max = n; //groesster in der Reihe noch nicht verwendeter Klotz
134         for (int i = n; i >= 1; i--) {
135             if (kloetze[i]) {
136                 max = i;
137                 break;
138             }
139         }
140         for (int i=0; i < nichtBelegt.size(); i++) {
141             if(i == nichtBelegt.size() -1) continue;
142             if (nichtBelegt.get(i+1) - nichtBelegt.get(i) > n){
143                 return true; //Kriterium Teil 1 erfuellt
144             }
145         }
146         for (int i=0; i < nichtBelegt.size(); i++) {
147             if((i == nichtBelegt.size() -1)||(nichtBelegt.get(i) < summe))
                ↪ continue;
148             if (nichtBelegt.get(i+1) - nichtBelegt.get(i) > max){
149                 return true; //Kriterium Teil 2 erfuellt
150             }
151         }
152         return false;
153     }
154 }
```

Quellcode 1: Der Backtracking-Algorithmus (`Main.java`)