Aufgabe 1: Arukone

Teilnahme-ID: 69408

Bearbeiter dieser Aufgabe:
Tim Krome

31. Oktober 2023

Lösungsidee

Verfahren zum Generieren eines Arukone

Begonnen wird mit einem leeren Arukone Gitter der Größe n x n, in dem alle Felder frei sind. In diesem Gitter werden nun so lange Paare gleicher Zahlen platziert, bis keine weiteren Paare mehr platziert werden können.
Das Platzieren eines Paars gleicher Zahlen läuft dabei immer wie folgt ab:

1. Es wird im Gitter nach einem freien Feld gesucht, das möglichst wenige freie Nachbarfelder[1], aber noch mind. 1 freies Nachbarfeld hat. Das gefundene Feld wird ab sofort als *Startfeld*

[1] Ein freies Feld ist ein Feld, auf dem sich weder einer Zahl noch eine Linie befindet.

bezeichnet. Wenn mehrere Felder gleich gut geeignet sind, dann wird das Feld mit dem niedrigsten Index ausgewählt.
(Sollte es kein einziges geeignetes Feld mehr geben, dann lassen sich keine weiteren Paare im Gitter platzieren und das Arukone ist somit vollständig gefüllt.)

2. Das Startfeld wird im Gitter als erstes Feld markiert, das zum zu platzierenden Paar gehört.
3. Vom Startfeld aus wird begonnen, eine Linie zu ziehen.
4. Eine zufällige Anzahl an Iterationen im Bereich [n, 2n] und eine zufällige Richtung (links, rechts, oben oder unten) wird festgelegt.
5. In einer Schleife wird die in Schritt 4 festgelegte Anzahl an Iterationen durchlaufen.
   In jeder Iteration wird versucht, die Linie geradeaus in die vorher festgelegte Richtung fortzusetzen.
   Es wird überprüft, ob dies möglich ist: Die Linie kann nur auf ein Feld fortgesetzt werden, wenn es leer ist[2] und nicht Nachbar eines anderen Felds ist, durch das dieselbe Linie bereits gezogen wurde[3] (abgesehen von dem Feld, von dem aus die Linie fortgesetzt wird). Wenn die Linie nicht geradeaus fortgesetzt werden kann, weil das sich dort befindende Feld diese Bedingungen nicht erfüllt oder weil der Gitterrand erreicht wurde, wird die Richtung geändert und die Linie wird in den nächsten Iterationen in die neue Richtung fortgesetzt (bis erneut auf ein Hindernis, also ein Feld, das nicht die Bedingungen erfüllt, gestoßen wird.
   Sollte es keine Richtung mehr geben, in die die Linie fortgesetzt werden kann, dann wird die Linie beendet und die Schleife abgebrochen. Das Feld, auf dem die Linie beendet wurde, wird fortan als *Endfeld* bezeichnet.
6. Das Endfeld der Linie wird im Gitter als zweites zum Paar gehörendes Feld markiert. Das Zahlenpaar wurde somit erfolgreich platziert. Die zwischen Startfeld und Endfeld gezogene Linie wird im Gitter gespeichert.

Dieses Verfahren generiert Arukone-Gitter, die allen Anforderungen aus der Aufgabe nachkommen:

- Die generierten Arukone sind alle lösbar, da beim Platzieren der Zahlenpaare alle Regeln berücksichtigt werden (siehe Fußnote 2).
- Da die Generierung des Gitters von Zufallswerten abhängt, generiert das Verfahren bei mehrfacher Anwendung zu einer hohen Wahrscheinlichkeit verschiedene Ergebnisse.
- Wie sich experimentell gezeigt hat (siehe Kapitel *Beispiele*), sind generierten Arukone sind meist komplex genug, um vom BWinf-Arukone-Checker nicht gelöst werden zu können. Dies liegt wahrscheinlich daran, dass der Arukone-Checker versucht, die Arukone durch das Ziehen von möglichst kurzen Linien zu lösen[4], die mit dem Verfahren generierten Arukone erfordern jedoch meistens das Ziehen von längeren und komplizierteren Verbindungslinien zwischen den Paaren gleicher Zahlen[5].
- **Dadurch, dass solange Paare platziert werden, bis kein Platz mehr für ein weiteres Paar ist, kann nicht direkt festgelegt werden, wie viele Paare das generierte Arukone enthalten soll, dies ist von den im Generierungsprozess gewählten Zufallswerten abhängig.**
  Es werden jedoch immer mindestens n/2 Paare platziert: Die gezogenen Linien haben eine

[2] Ein Feld ist leer, wenn es keine Zahl enthält und von keiner Linie durchzogen wird. Dadurch, dass überprüft wird, ob das Feld, auf das die Linie fortgesetzt wird, leer ist, werden keine Arukone generiert, die den Regeln aus der Aufgabestellung nicht entsprechen.

[3] Es wird überprüft, dass das Feld, auf das die Linie fortgesetzt wird, kein anderes Nachbarfeld hat, durch das bereits dieselbe gezogen wurde. Hierdurch wird dafür gesorgt, dass es (außer in sehr seltenen Sonderfällen) keine Möglichkeiten gibt, im vollständig gefüllten Arukone die Linien zwischen den Zahlen abzukürzen.

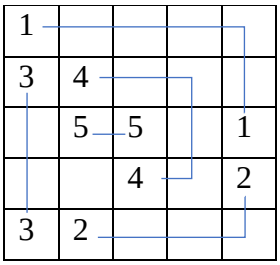
[4] Dies hat sich experimentell beim Ausprobieren des Arukone-Checkers erwiesen. Der genaue Lösungsalgorithmus, den der Arukone-Checker verwendet, ist unbekannt.

[5] Dem ist natürlich nicht zwangsläufig für jedes generierte Arukone der Fall, da die Länge der Verbindungslinie beim Platzieren eines Paars in Schritt 4 zufällig gewählt wird.

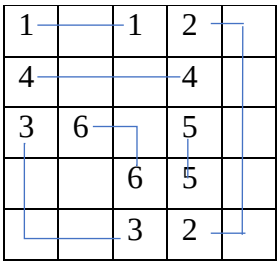
maximale Länge von 2*n (einschließlich Start- und Endfeld)[6], decken also maximal *2*n* Felder ab. Das Arukone-Gitter verfügt über *n*n* Felder. Angenommen, alle im Arukone platzierten Linien decken je *2*n* Felder ab, dann können im Arukone also $\frac{n*n}{2*n} = \frac{n}{2}$ Linien gezogen bzw. Paare platziert werden, was genau der in der Aufgabe geforderten Zahl entspricht.

Ich habe mich bewusst für ein etwas komplizierteres Verfahren entschieden, damit die generierten Arukone abwechslungsreich sind. Da im Generierungsprozess die Linien zwischen den Zahlenpaaren gezogen und gespeichert werden, fällt die Lösung des generierten Arukone als „Nebenprodukt“ mit an.

Beispiele für mit obigem Verfahren generierte Arukone:



Arukone 1



Arukone 2

Noch abwechslungsreichere Arukone erstellen

Alle mit obigem Verfahren generierten Arukone haben derzeit noch eine Gemeinsamkeit: In einer der Ecken befindet sich immer eine Eins. Dies liegt daran, dass beim Platzieren eines Paars gleicher Zahlen immer das Feld verwendet wird, dass am wenigsten freie Nachbarfelder hat (aber dennoch mind.1 freies Nachbarfeld hat), es werden also immer „Eckfelder“ als Startfelder verwendet. Beim Platzieren des ersten Paars kommen somit nur die vier Ecken des Arukone als Startfelder infrage.

Um an dieser Stelle Abwechslung in die generierten Rätsel zu bringen, wird das Verfahren angepasst: Beim Platzieren des ersten Paars wird das Startfeld fortan zufällig gewählt. Damit hierdurch keine Linien entstehen, die das ganze Arukone einmal durchziehen (dies hätte negative Auswirkungen auf den Generierungsprozess), wird festgelegt, dass das zufällig gewählte Startfeld in der oberen Hälfte des Arukone verortet sein muss und die vom Startfeld ausgehende Linie immer zunächst nach oben fortgesetzt wird.

„Seed“ zum Reproduzieren eines Ergebnisses

Das Verfahren generiert bei mehrfacher Anwendung verschiedene Ergebnisse, da die Platzierung der Zahlenpaare von Zufallswerten abhängt. Damit gezeigt werden kann, dass es ein bestimmtes Arukone tatsächlich vom Verfahren generiert wurde, wird das Verfahren so erweitert, dass es alle verwendeten Zufallswerte in der Reihenfolge der Verwendung speichert und als „Seed“ zusammen mit dem generierten Arukone ausgibt. Dieser Seed ermöglicht das Reproduzieren des Ergebnisses, da in ihm alle Zufallswerte gespeichert sind, die zum Erzielen des Ergebnisses benutzt wurden.

[6] Dem ist der Fall, da beim Platzieren eines Zahlenpaars eine zufällige Anzahl an Iterationen gewählt wird, die sich im Bereich [n; 2n] befindet.

Umsetzung

Die Lösungsidee wird in Python 3.10 implementiert. Wichtig ist zu beachten, dass das Programm bei einmaliger Ausführung nur ein einziges Arukone generiert. **Wird das Programm allerdings mehrfach ausgeführt (mit dem Parameter *seed* = None) wird es zu einer hohen Wahrscheinlichkeit verschiedene Arukone ausgeben**, da die Generierung von Zufallswerten abhängt, wenn kein Input-Seed angegeben wird.

Eingabe

Das Programm nimmt als Eingabeparameter einen Wert für die Gittergröße *n* sowie einen Input-Seed *seed*, der das Reproduzieren eines Ergebnisses ermöglicht[7]. Soll ein zufälliges Arukone erzeugt werden, dann muss *seed* auf None gesetzt werden. Den Parametern werden in den ersten fünf Programmzeilen ihre Werte zugewiesen.

Modellierung des Arukone-Gitters

Das Arukone-Gitter wird in einer eindimensionalen Liste namens *grid* gespeichert. Das 2D-Gitter wird auf die 1D-Liste abgebildet, indem die Gitterelemente nacheinander Zeile für Zeile angeordnet werden. Die 1D-Liste stellt die ursprüngliche Gitterstruktur also linear da. Dies ermöglicht es, mit nur einem Index auf jedes Feld zuzugeifen, was den Generierungsprozess vereinfacht. Leere Felder werden in der Liste als „0“ gespeichert. Mit der Zahl *x* belegte Felder werden in der Liste als „*x*“ dargestellt. Wenn sich auf einem Feld eine Verbindungslinie zwischen den beiden mit der Zahl *x* belegten Feldern befindet, dann wird das Feld als ***„.x“*** gespeichert. Auf die Art kann die Struktur des vollständig ausgefüllten Arukone in einer gewöhnlichen (eindimensionalen) Python-Liste modelliert werden.

Zu Beginn wird mit folgendem Programmcode ein leeres Arukone-Gitter erzeugt:

```
grid = [0 for i in range(n*n)]
```

Funktion zum Finden eines Startfelds

Um das Verfahren zum Platzieren eines Zahlenpaars umzusetzen, muss zunächst eine Funktion implementiert werden, die ein geeignetes Startfeld findet. Dies wiederum erfordert eine Funktion, die für ein beliebiges Feld die Anzahl an freien Nachbarfeldern bestimmen kann. Es wird also eine `num_free_neighbors(pos)` Funktion implementiert, die die Nachbarfelder des Felds am Index *pos* überprüft und die Anzahl an freien Nachbarfeldern zuzugeifen.

Anschließend wird eine `get_start_field()` Funktion implementiert, die ein mögliches Startfeld für Schritt 1 des Platzierverfahrens findet. Wie im Kapitel *Lösungsidee > Verfahren zum Generieren eines Arukone* beschrieben soll das Startfeld ein Feld sein, dass möglichst wenig freie Nachbarn hat, aber mind. 1 freies Nachbarfeld hat. Die Funktion iteriert in einer for-Schleife über alle Felder, bestimmt für jedes Feld mit der *num_free_neighbors* Funktion die Anzahl freier Nachbarfelder,

[7] Eine Programmausgabe kann reproduziert werden, in dem der in der Programmausgabe enthaltene Seed dem Programm als Eingabe-Seed bzw. als Wert für den Parameter *seed* gegeben wird

speichert das am besten geeignete Feld und gibt es nach Vollendung der for-Schleife zurück. Gibt es kein freies Feld, das nicht mind. einen freien Nachbarn hat, dann gibt die Funktion `None` zurück:

```
def get_start_field():
    feld = None # Bisher "bestes" gefundenes Feld
    best = 5 # Anzahl an freien Nachbarfeldern, die das "beste" gefundene Feld hat
    for _feld in range(n*n):
        if grid[_feld] == 0: # Überprüfen, ob das Feld selbst frei ist
            score = num_free_neighbors(_feld)
            if score != 0: # Sicherstellen, dass das Feld mind. 1 freien Nachbarn hat
                if score < best:
                    best = score
                    feld = _feld
    return feld
```

Überprüfen, ob eine Linie auf ein Feld fortgesetzt werden kann

Damit Schritt 5 des Platzierverfahrens (in Lösungsidee beschrieben) umgesetzt werden kann, wird eine Funktion namens `is_allowed_step(new_pos, old_pos, paar)` implementiert, die als Argumente die Felder *old_pos* und *new_pos* sowie den Integer *paar* nimmt, der angibt, welches Paar gerade platziert wird. Sie gibt als Boolean zurück, ob eine Linie vom Feld *old_pos* auf das benachbarten Feld *new_pos* fortgesetzt werden kann. Die Funktion gleich hierfür folgende Negativkriterien ab:

- Wenn sich *new_pos* außerhalb des Gitterrands befindet, gibt die Funktion False zurück
- Wenn beim Schritt von *old_pos* nach *new_pos* der linke oder rechte Gitterrand überschritten wird, gibt die Funktion False zurück
- Wenn das Feld *new_pos* nicht frei ist, gibt die Funktion False zurück
- Wenn eines der Nachbarfelder von *new_pos* (außer dem Nachbarfeld *old_pos*) bereits von der fortzusetzenden Linie durchzogen wird, gibt die Funktion False zurück

Ist keines der beschriebenen Negativkriterien erfüllt, kann die Linie auf das Feld *new_pos* fortgesetzt werden, die Funktion gibt demnach True zurück.

Generierung des Arukone

Die Zählervariable *num_paare*, die die Anzahl der platzierten Paare zählen wird, wird auf 0 initialisiert. Anschließend wird eine *while True* Schleife ausgeführt, die abgebrochen wird, sobald kein weiteres Paar mehr platziert werden kann. In jeder Iteration der Schleife versucht das Programm, ein Zahlenpaar zu platzieren, indem die im Kapitel *Lösungsidee > Verfahren zum Generieren eines Arukone* beschriebenen Schritte zum Platzieren eines Paars umgesetzt werden:

1. Zuerst wird das Startfeld mit folgendem Code festgelegt: `start_field = get_start_field()`
   Wenn get_start_field() None zurückgibt bzw. es kein geeignetes Startfeld mehr gibt, dann wird die *while True* Schleife abgebrochen.
   In der ersten Iteration der *while True* Schleife wird nicht die get_start_field() Funktion verwendet, sondern ein zufälliges Feld als Startfeld ausgewählt:
   `start_field = random.randint(0,n*round(n/2))`

2. *num_paare* wird um 1 erhöht und das Startfeld wird im Gitter markiert:
   `num_paare += 1`
   `grid[start_field] = num_paare`

3. Die Länge der zu generierenden Line wird festgelegt:

```
iterations = random.randint(n,2*n)
```

4. Mit diesem Code wird die Linie generiert:

```
for i in range(iterations):
```

Bei jeder Iteration wird versucht, die Linie in die Richtung *direction* fortzusetzen.

```
directions = [-n, 1, n, -1]
new_position = position + direction
```

Die while-Schleife wird ausgeführt, wenn die Linienerweiterung nach *new_position* nicht möglich ist:

```
while not is_allowed_step(position, new_position, num_paare):
```

Wenn die Linie in die Richtung *direction* nicht fortgesetzt werden kann, dann wird solange eine neue Richtung ausprobiert, bis eine Richtung gefunden wurde, in die sie fortgesetzt werden kann (oder alle Richtungen durchprobiert wurden).

```
    if directions == []:
```

Wenn alle Richtungen durchprobiert wurden und die Linie nicht fortgesetzt werden kann, dann wird dies gespeichert und die Schleife abgebrochen.

```
        found_direction = False
        break
    direction = directions.pop(0)
    new_position = position + direction
else:
```

Dieser Programmteil wird ausgeführt, wenn eine Richtung gefunden wurde, in die die Linie fortgesetzt werden kann.

```
    found_direction = True

if found_direction:
    # Linie kann fortgesetzt werden:
    grid[new_position] = "."+str(num_paare) # Neuen zur Linie
                            gehörenden Punkt in Gitter markieren
    position = new_position # Position aktualisieren
    continue
else:
    # Linie kann nicht fortgesetzt werden -> Linie beenden
    break
```

5. Das Endfeld der generierten Linie wird im Gitter markiert:

```
grid[position] = num_paare
```

Implementierung von Seeds

- Es wird implementiert, dass das Programm alle verwendeten Zufallswerte in Verwendungsreihenfolge in der Liste *output_seed* speichert und nach der Generierung des Arukone mit Semikolons verknüpft als Seed ausgibt.
- Es wird implementiert, dass das Programm einen Parameter *seed* als Eingabe-Seed nimmt. Wenn ein vom Programm ausgegebener Seed dem Programm als Eingabe-Seed gegeben wird (mit der richtigen Gittergröße), wird das Programm für die Genierung statt Zufallswerten die Werte aus dem Seed verwenden und dadurch dasselbe Rätsel noch einmal generieren.

Ausgabe

Die Programmausgabe ist in drei Abschnitte gegliedert:

- Abschnitt „Generiertes Rätsel“: Hier wird das erzeugte Arukone ausgegeben. Format:
  1. Zeile beschreibt die Gittergröße n
  2. Zeile beschreibt die Anzahl an Paaren
  Weitere n Zeilen mit jeweils n Zahlen, also ein Eintrag pro Feld: Die in dem Feld enthaltene Zahl oder 0 für leere Felder
  Dieses Ausgabeformat entspricht dem Eingabeformat vom BWinf-Arukone-Checker, die ausgegebenen Arukone können also direkt in den Arukone-Checker eingegeben werden.
- Abschnitt „Lösung des generierten Rätsels“: Hier wird die Lösung des generierten Rätsels ausgegeben.
  Format der Lösung: n Zeilen mit jeweils n Einträgen, also ein Eintrag pro Feld.
  Die Einträge sind wie folgt dargestellt: [0] bedeutet, dass das Feld leer ist. [m] bedeutet, dass das Feld die Zahl m enthält. (m) bedeutet, dass sich auf dem Feld die Verbindungslinie zwischen den beiden Feldern mit der Zahl m befindet.
- Abschnitt „Seed zum Reproduzieren“: Hier wird der Seed des generierten Rätsels ausgegeben.

Laufzeitanalyse

Im Programmteil, der die Arukone generiert, kommt eine while Schleife vor, die bei jedem Durchlauf ein Paar platziert. In einem Gitter der Seitenlänge n können maximal $n^2/2$ Paare platziert werden (wenn immer zwei benachbarte Felder zu einem Zahlenpaar gemacht werden), deshalb schätze ich die (theoretische) Worst-Case-Laufzeit der Schleife als $n^2/2$ ab. In fast jedem Durchlauf der Schleife wird die *get_start_field* Funktion aufgerufen, die eine Laufzeit von n^2 hat, da in ihr eine Schleife vorkommt, die über alle Felder iteriert. Die Laufzeit des Gesamtprogramms ist somit $(n^2/2) * n^2$, in O-Notation: $O(n^4)$. Die anderen Programmteile berücksichtige ich bei der Laufzeitanalyse nicht, da sie nur für Verarbeitung der Eingabe / Ausgabe zuständig sind und die Laufzeit dieser Programmteile außerdem niedriger als n^4 ist.

Beispiele

In Beispiel 1 – 12 wurden mit dem Programm für verschiedene Gittergrößen Arukone generiert. Der Parameter *seed* wurde dabei stets auf None gesetzt, damit die generierten Arukone alle zufällig sind. Alle Arukone aus den Beispielen können vom Programm jederzeit reproduziert werden, wenn n so wie im jeweiligen Beispiel gesetzt wird und der Seed aus der Programmausgabe als Eingabe-Seed

verwendet wird.
Alle generierten Arukone wurden in den BWinf Arukone-Checker eingegeben, zu jedem Beispiel ist ein Screenshot eingefügt. In 11 der 12 Beispiele war der Arukone-Checker nicht in der Lage, die generierten Arukone zu lösen, die Bedingung aus der Aufgabenstellung ist somit erfüllt. Nur in Beispiel 1 gelang dem Checker das Lösen.
Ausprobieren hat ergeben, dass der Arukone-Checker bei der Gittergröße n = 4 oft (aber nicht immer, siehe Beispiel 2) in der Lage ist, die vom Programm generierten Arukone zu lösen. Bei Gittergrößen n > 4 hingegen löst der Arukone-Checker die generierten Arukone nur sehr selten.

In der Aufgabe wird verlang, dass zu jedem Rätsel auch eine Lösung mitangegeben werden soll. Dies ist erfüllt, da das Programm zu jedem generierten Rätsel eine in Textzeichen dargestellte Lösung mit ausgibt.
In Kapitel *Umsetzung* > *Ausgabe* wird präzisiert, wie die ausgegebenen Lösungen formatiert sind, sollten diesbezüglich Unklarheiten bestehen.

Beispiel 1

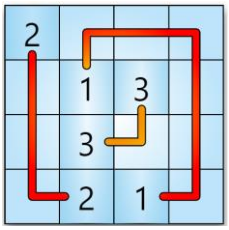
Parameter:
n = 4
seed = None

Ausgabe des Programms:
```
Generiertes Rätsel:
4
3
2 0 0 0
0 1 3 0
0 3 0 0
0 2 1 0

Lösung des generierten Rätsels:
[2] (1) (1) (1)
(2) [1] [3] (1)
(2) [3] (3) (1)
(2) [2] [1] (1)

Seed zum Reproduzieren:
5;7;6;10
```

Lösungsversuch des BWinf-Arukone-Checkers:
Der Arukone-Checker kann dieses Rätsel lösen.



Beispiel 2

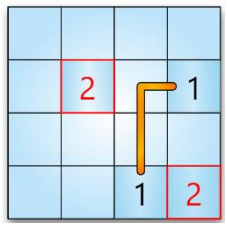
Parameter:
n = 4
seed = None

Ausgabe des Programms:
```
Generiertes Rätsel:
4
2
0 0 0 0
0 2 0 1
0 0 0 0
0 0 1 2

Lösung des generierten Rätsels:
(1) (1) (1) (1)
(1) [2] [0] [1]
(1) (2) (2) (2)
(1) (1) [1] [2]

Seed zum Reproduzieren:
7;9;4
```

Lösungsversuch des BWinf-Arukone-Checkers:
Der Arukone-Checker kann dieses Rätsel *nicht* lösen.



Beispiel 3

Eingabedatei:
n = 5
seed = None

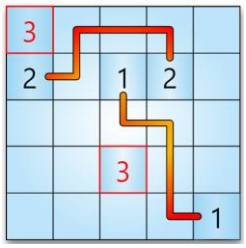
Ausgabe:
```
Generiertes Rätsel:
5
3
3 0 0 0 0
2 0 1 2 0
0 0 0 0 0
0 0 3 0 0
0 0 0 0 1

Lösung des generierten Rätsels:
[3] (3) (1) (1) (1)
[2] (3) [1] [2] (1)
(2) (3) [0] (2) (1)
```

```
(2) (3) [3] (2) (1)
(2) (2) (2) (2) [1]

Seed zum Reproduzieren:
7;7;9;6
```

Lösungsversuch des BWinf-Arukone-Checkers:
Der Arukone-Checker kann dieses Rätsel *nicht* lösen.



Beispiel 4

Eingabedatei:
n = 6
seed = None

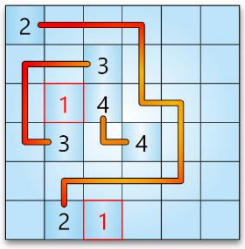
Ausgabe:

```
Generiertes Rätsel:
6
4
2 0 0 0 0 0
0 0 3 0 0 0
0 1 4 0 0 0
0 3 0 4 0 0
0 0 0 0 0 0
0 2 1 0 0 0

Lösung des generierten Rätsels:
[2] (1) (1) (1) (1) (1)
(2) (1) [3] (3) (3) (1)
(2) [1] [4] (4) (3) (1)
(2) [3] [0] [4] (3) (1)
(2) (3) (3) (3) (3) (1)
(2) [2] [1] (1) (1) (1)

Seed zum Reproduzieren:
13;14;6;12;17
```

Lösungsversuch des BWinf-Arukone-Checkers:
Der Arukone-Checker kann dieses Rätsel *nicht* lösen.



Beispiel 5

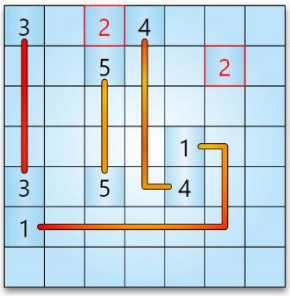
Eingabedatei:
n = 7
seed = None

Ausgabe:
```
Generiertes Rätsel:
7
5
3 0 2 4 0 0 0
0 0 5 0 0 2 0
0 0 0 0 0 0 0
0 0 0 0 1 0 0
3 0 5 0 4 0 0
1 0 0 0 0 0 0
0 0 0 0 0 0 0

Lösung des generierten Rätsels:
[3] (2) [2] [4] (1) (1) (1)
(3) (2) [5] (4) (1) [2] (1)
(3) (2) (5) (4) (1) (2) (1)
(3) (2) (5) (4) [1] (2) (1)
[3] (2) [5] (4) [4] (2) (1)
[1] (2) (2) (2) (2) (2) (1)
(1) (1) (1) (1) (1) (1) (1)

Seed zum Reproduzieren:
25;18;14;20;12;12
```

Lösungsversuch des BWinf-Arukone-Checkers:
Der Arukone-Checker kann dieses Rätsel *nicht* lösen.



Beispiel 6

Eingabedatei:
n = 8
seed = None

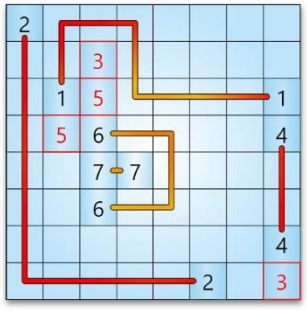
Ausgabe:
```
Generiertes Rätsel:
8
7
2 0 0 0 0 0 0 0
0 0 3 0 0 0 0 0
0 1 5 0 0 0 0 1
0 5 6 0 0 0 0 4
0 0 7 7 0 0 0 0
```

```
0 0 6 0 0 0 0 0
0 0 0 0 0 0 0 4
0 0 0 0 0 2 0 3

Lösung des generierten Rätsels:
[2] (1) (1) (1) (1) (1) (1) (1)
(2) (1) [3] (3) (3) (3) (3) (1)
(2) [1] [5] (5) (5) (5) (3) [1]
(2) [5] [6] (6) (6) (5) (3) [4]
(2) (5) [7] [7] (6) (5) (3) (4)
(2) (5) [6] (6) (6) (5) (3) (4)
(2) (5) (5) (5) (5) (5) (3) [4]
(2) (2) (2) (2) (2) [2] (3) [3]

Seed zum Reproduzieren:
17;10;12;11;10;23;14;16
```

Lösungsversuch des BWinf-Arukone-Checkers:
Der Arukone-Checker kann dieses Rätsel *nicht* lösen.



Beispiel 7

Eingabedatei:
n = 8
seed = None

Ausgabe:

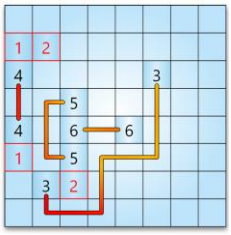
```
Generiertes Rätsel:
8
6
0 0 0 0 0 0 0 0
1 2 0 0 0 0 0 0
4 0 0 0 0 3 0 0
0 0 5 0 0 0 0 0
4 0 6 0 6 0 0 0
1 0 5 0 0 0 0 0
0 3 2 0 0 0 0 0
0 0 0 0 0 0 0 0

Lösung des generierten Rätsels:
(1) (1) (1) (1) (1) (1) (1) (1)
[1] [2] (2) (2) (2) (2) (2) (1)
[4] (3) (3) (3) (3) [3] (2) (1)
(4) (3) [5] (5) (5) (5) (2) (1)
[4] (3) [6] (6) [6] (5) (2) (1)
[1] (3) [5] (5) (5) (5) (2) (1)
(1) [3] [2] (2) (2) (2) (2) (1)
(1) (1) (1) (1) (1) (1) (1) (1)

Seed zum Reproduzieren:
```

```
8;24;14;8;9;15;8
```

Lösungsversuch des BWinf-Arukone-Checkers:
Der Arukone-Checker kann dieses Rätsel *nicht* lösen.



Beispiel 8

Eingabedatei:
n = 9
seed = None

Ausgabe:

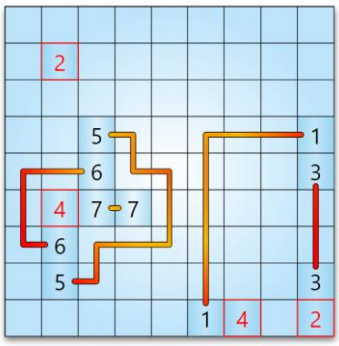
```
Generiertes Rätsel:
9
7
0 0 0 0 0 0 0 0 0
0 2 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0
0 0 5 0 0 0 0 0 1
0 0 6 0 0 0 0 0 3
0 4 7 7 0 0 0 0 0
0 6 0 0 0 0 0 0 0
0 5 0 0 0 0 0 0 3
0 0 0 0 0 1 4 0 2

Lösung des generierten Rätsels:
(1) (1) (1) (1) (1) (1) (1) (1) (1)
(1) [2] (2) (2) (2) (2) (2) (2) (1)
(1) (4) (4) (4) (4) (4) (4) (2) (1)
(1) (4) [5] (5) (5) (5) (4) (2) [1]
(1) (4) [6] (6) (6) (5) (4) (2) [3]
(1) [4] [7] [7] (6) (5) (4) (2) (3)
(1) [6] (6) (6) (6) (5) (4) (2) (3)
(1) [5] (5) (5) (5) (5) (4) (2) [3]
(1) (1) (1) (1) (1) [1] [4] (2) [2]

Seed zum Reproduzieren:
35;24;26;24;14;11;12;15
```

Lösungsversuch des BWinf-Arukone-Checkers:

Der Arukone-Checker kann dieses Rätsel *nicht* lösen.



Beispiel 9

Eingabedatei:
n = 9
seed = None

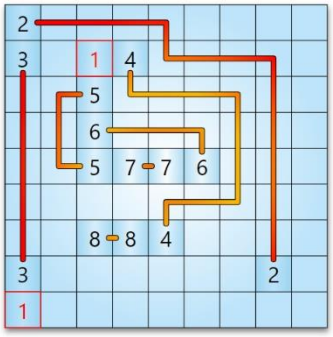
Ausgabe:

```
Generiertes Rätsel:
9
8
2 0 0 0 0 0 0 0 0
3 0 1 4 0 0 0 0 0
0 0 5 0 0 0 0 0 0
0 0 6 0 0 0 0 0 0
0 0 5 7 7 6 0 0 0
0 0 0 0 0 0 0 0 0
0 0 8 8 4 0 0 0 0
3 0 0 0 0 0 0 2 0
1 0 0 0 0 0 0 0 0

Lösung des generierten Rätsels:
[2] (2) (1) (1) (1) (1) (1) (1) (1)
[3] (2) [1] [4] (4) (4) (4) (4) (1)
(3) (2) [5] (5) (5) (5) (5) (4) (1)
(3) (2) [6] (6) (6) (6) (5) (4) (1)
(3) (2) [5] [7] [7] [6] (5) (4) (1)
(3) (2) (5) (5) (5) (5) (5) (4) (1)
(3) (2) [8] [8] [4] (4) (4) (4) (1)
[3] (2) (2) (2) (2) (2) (2) [2] (1)
[1] (1) (1) (1) (1) (1) (1) (1) (1)

Seed zum Reproduzieren:
11;23;14;14;12;14;12;17;19
```

Lösungsversuch des BWinf-Arukone-Checkers:
Der Arukone-Checker kann dieses Rätsel *nicht* lösen.



Beispiel 10

Eingabedatei:
n = 12
seed = None

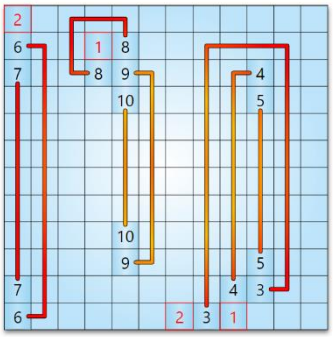
Ausgabe:

```
Generiertes Rätsel:
12
10
2 0 0 0 0 0 0 0 0 0 0 0
6 0 0 1 8 0 0 0 0 0 0 0
7 0 0 8 9 0 0 0 0 4 0 0
0 0 0 0 10 0 0 0 0 5 0 0
0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 10 0 0 0 0 0 0 0
0 0 0 0 9 0 0 0 0 5 0 0
7 0 0 0 0 0 0 0 4 3 0 0
6 0 0 0 0 0 2 3 1 0 0 0

Lösung des generierten Rätsels:
[2] (2) (2) (1) (1) (1) (1) (1) (1) (1) (1) (1)
[6] (6) (2) [1] [8] (8) (8) (3) (3) (3) (3) (1)
[7] (6) (2) [8] [9] (9) (8) (3) (4) [4] (3) (1)
(7) (6) (2) (8) [10] (9) (8) (3) (4) [5] (3) (1)
(7) (6) (2) (8) (10) (9) (8) (3) (4) (5) (3) (1)
(7) (6) (2) (8) (10) (9) (8) (3) (4) (5) (3) (1)
(7) (6) (2) (8) (10) (9) (8) (3) (4) (5) (3) (1)
(7) (6) (2) (8) (10) (9) (8) (3) (4) (5) (3) (1)
(7) (6) (2) (8) [10] (9) (8) (3) (4) (5) (3) (1)
(7) (6) (2) (8) [9] (9) (8) (3) (4) [5] (3) (1)
[7] (6) (2) (8) (8) (8) (8) (3) [4] [3] (3) (1)
[6] (6) (2) (2) (2) (2) [2] [3] [1] (1) (1) (1)

Seed zum Reproduzieren:
15;23;17;24;33;29;31;12;32;25;32
```

Lösungsversuch des BWinf-Arukone-Checkers:
Der Arukone-Checker kann dieses Rätsel *nicht* lösen.



Beispiel 11

Eingabedatei:

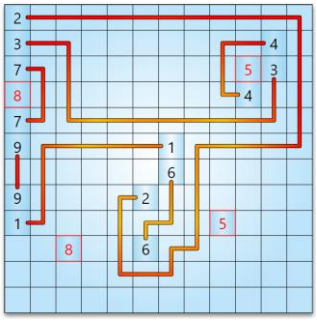
```
Generiertes Rätsel:
12
9
2 0 0 0 0 0 0 0 0 0 0 0
3 0 0 0 0 0 0 0 0 0 4 0
7 0 0 0 0 0 0 0 0 5 3 0
8 0 0 0 0 0 0 0 0 4 0 0
7 0 0 0 0 0 0 0 0 0 0 0
9 0 0 0 0 0 1 0 0 0 0 0
0 0 0 0 0 0 6 0 0 0 0 0
9 0 0 0 0 2 0 0 0 0 0 0
1 0 0 0 0 0 0 0 5 0 0 0
0 0 8 0 0 6 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0

Lösung des generierten Rätsels:
[2] (2) (2) (2) (2) (2) (1) (1) (1) (1) (1) (1)
[3] (3) (3) (3) (3) (2) (1) (4) (4) (4) [4] (1)
[7] (7) (7) (7) (3) (2) (1) (4) (5) [5] [3] (1)
[8] (8) (8) (7) (3) (2) (1) (4) (5) [4] (3) (1)
[7] (7) (8) (7) (3) (2) (1) (4) (5) (4) (3) (1)
[9] (7) (8) (7) (3) (2) [1] (4) (5) (4) (3) (1)
(9) (7) (8) (7) (3) (2) [6] (4) (5) (4) (3) (1)
[9] (7) (8) (7) (3) [2] (6) (4) (5) (4) (3) (1)
[1] (7) (8) (7) (3) [0] (6) (4) [5] (4) (3) (1)
(1) (7) [8] (7) (3) [6] (6) (4) (4) (4) (3) (1)
(1) (7) (7) (7) (3) (3) (3) (3) (3) (3) (3) (1)
(1) (1) (1) (1) (1) (1) (1) (1) (1) (1) (1) (1)

Seed zum Reproduzieren:
66;35;12;27;19;25;22;29;22;27
```

Lösungsversuch des BWinf-Arukone-Checkers:

Der Arukone-Checker kann dieses Rätsel *nicht* lösen.



Beispiel 12

Eingabedatei:
n = 25
seed = None

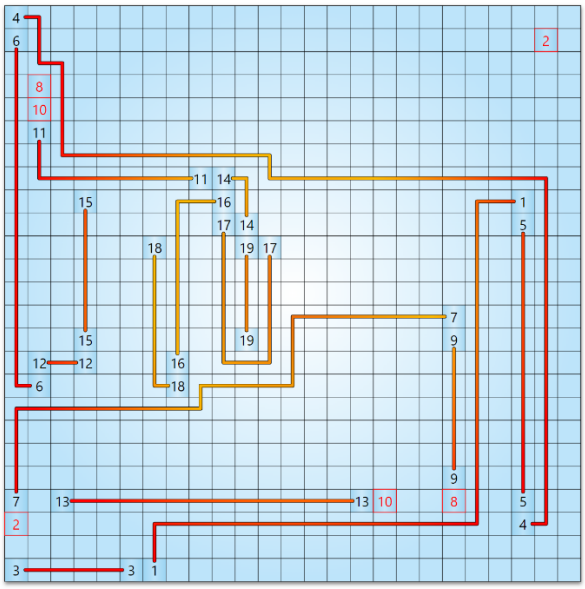
Ausgabe:

```
Generiertes Rätsel:
25
19
4 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
6 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 8 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 10 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 11 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 11 14 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 15 0 0 0 0 0 16 0 0 0 0 0 0 0 0 0 0 0 1 0 0
0 0 0 0 0 0 0 0 0 17 14 0 0 0 0 0 0 0 0 0 0 5 0 0
0 0 0 0 0 0 18 0 0 0 19 17 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 7 0 0 0 0 0
0 0 0 15 0 0 0 0 0 0 19 0 0 0 0 0 0 0 0 9 0 0 0 0 0
0 12 0 12 0 0 0 16 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 6 0 0 0 0 0 0 18 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 9 0 0 0 0 0
7 0 13 0 0 0 0 0 0 0 0 0 0 0 13 10 0 0 8 0 0 5 0 0
2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 4 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
3 0 0 0 0 3 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

Lösung des generierten Rätsels:
[4] (4) (4) (4) (4) (4) (4) (4) (4) (4) (4) (4) (4) (4) (4) (4) (4) (4) (4) (4) (4) (4) (1) (1) (1)
[6] (6) (6) (6) (6) (6) (6) (6) (6) (6) (6) (6) (6) (6) (6) (6) (6) (6) (6) (6) (6) (4) (1) [2] (1)
(7) (7) (7) (7) (7) (7) (7) (7) (7) (7) (7) (7) (7) (7) (7) (7) (7) (7) (7) (7) (6) (4) (1) (2) (1)
(7) [8] (8) (8) (8) (8) (8) (8) (8) (8) (8) (8) (8) (8) (8) (8) (8) (8) (8) (8) (7) (6) (4) (1) (2) (1)
(7) [10] (10) (10) (10) (10) (10) (10) (10) (10) (10) (10) (10) (10) (10) (10) (10) (10) (8) (7) (6) (4) (1) (2) (1)
(7) [11] (11) (11) (11) (11) (11) (11) (11) (11) (11) (11) (11) (11) (11) (11) (11) (10) (8) (7) (6) (4) (1) (2) (1)
(7) (12) (12) (12) (12) (12) (12) (12) (12) (12) (12) (12) (12) (12) (12) (12) (11) (10) (8) (7) (6) (4) (1) (2) (1)
(7) (12) (11) (11) (11) (11) (11) (11) [11] [14] (14) (14) (14) (14) (14) (12) (11) (10) (8) (7) (6) (4) (1) (2) (1)
(7) (12) (11) [15] (14) (14) (14) (14) (14) [16] (16) (16) (16) (16) (14) (12) (11) (10) (8) (7) (6) (4) [1] (2) (1)
(7) (12) (11) (15) (14) (16) (16) (16) (14) [17] [14] (14) (14) (16) (14) (12) (11) (10) (8) (7) (6) (4) [5] (2) (1)
(7) (12) (11) (15) (14) (16) [18] (16) (14) (17) [19] [17] (14) (16) (14) (12) (11) (10) (8) (7) (6) (4) (5) (2) (1)
(7) (12) (11) (15) (14) (16) (18) (16) (14) (17) (19) (17) (14) (16) (14) (12) (11) (10) (8) (7) (6) (4) (5) (2) (1)
(7) (12) (11) (15) (14) (16) (18) (16) (14) (17) (19) (17) (14) (16) (14) (12) (11) (10) (8) (7) (6) (4) (5) (2) (1)
(7) (12) (11) (15) (14) (16) (18) (16) (14) (17) (19) (17) (14) (16) (14) (12) (11) (10) (8) [7] (6) (4) (5) (2) (1)
(7) (12) (11) [15] (14) (16) (18) (16) (14) (17) [19] (17) (14) (16) (14) (12) (11) (10) (8) [9] (6) (4) (5) (2) (1)
(7) [12] (11) [12] (14) (16) (18) [16] (14) (17) (17) (17) (14) (16) (14) (12) (11) (10) (8) (9) (6) (4) (5) (2) (1)
(7) [6] (11) (12) (14) (16) (18) [18] (14) (14) (14) (14) (14) (16) (14) (12) (11) (10) (8) (9) (6) (4) (5) (2) (1)
(7) (6) (11) (12) (14) (16) (16) (16) (16) (16) (16) (16) (16) (16) (14) (12) (11) (10) (8) (9) (6) (4) (5) (2) (1)
(7) (6) (11) (12) (14) (14) (14) (14) (14) (14) (14) (14) (14) (14) (14) (12) (11) (10) (8) (9) (6) (4) (5) (2) (1)
(7) (6) (11) (12) (12) (12) (12) (12) (12) (12) (12) (12) (12) (12) (12) (12) (11) (10) (8) (9) (6) (4) (5) (2) (1)
(7) (6) (11) (11) (11) (11) (11) (11) (11) (11) (11) (11) (11) (11) (11) (11) (11) (10) (8) [9] (6) (4) (5) (2) (1)
[7] (6) [13] (13) (13) (13) (13) (13) (13) (13) (13) (13) (13) (13) (13) [13] [10] (10) (8) [8] (6) (4) [5] (2) (1)
[2] (6) (6) (6) (6) (6) (6) (6) (6) (6) (6) (6) (6) (6) (6) (6) (6) (6) (6) (6) (4) [4] (2) (1)
(2) (2) (2) (2) (2) (2) (2) (2) (2) (2) (2) (2) (2) (2) (2) (2) (2) (2) (2) (2) (2) (2) (2) (1)
[3] (3) (3) (3) (3) [3] [1] (1) (1) (1) (1) (1) (1) (1) (1) (1) (1) (1) (1) (1) (1) (1) (1) (1) (1)

Seed zum Reproduzieren:
222;52;46;26;60;53;66;49;56;69;34;63;52;25;61;60;49;53;74;62
```

Lösungsversuch des BWinf-Arukone-Checkers:
Der Arukone-Checker kann dieses Rätsel *nicht* lösen.



Quellcode

```
import random

# Parameter festlegen
n = 8 # Größe des Arukone-Gitters
seed = None # Startwert für die Arukone-Generierung. Bei gleichbleibendem Seed wird das Programm immer dasselbe Arukone ausgeben. Soll ein zufälliges Arukone generiert werden, dann muss seed auf None gesetzt werden

def num_free_neighbors(pos):
    """
    Returns:
        int: Die Anzahl an freien Nachbarfeldern, die das Feld am Index pos hat
    """
    num_free_neighbors = 0
    if pos - n >= 0:
        if grid[pos-n] == 0:
            num_free_neighbors += 1
    if pos + n < n*n:
        if grid[pos+n] == 0:
            num_free_neighbors += 1
    if pos % n != 0:
        if grid[pos-1] == 0:
            num_free_neighbors += 1
    if pos % n != n-1:
        if grid[pos+1] == 0:
            num_free_neighbors += 1
    return num_free_neighbors

def get_start_field():
    """
    Returns:
        int: Der Index von dem Feld, das am wenigsten freie Nachbarfelder hat, aber mind. 1 freies Nachbarfeld hat. Wenn es kein solches Feld gibt, wird None zurückgegeben
    """
    feld = None # Bisher "bestes" gefundenes Feld
    best = 5 # Anzahl an freien Nachbarfeldern, die das "beste" gefundene Feld hat
```

```
    for _feld in range(n*n):
        if grid[_feld] == 0: # Überprüfen, ob das Feld selbst frei ist
            score = num_free_neighbors(_feld)
            if score != 0: # Sicherstellen, dass das Feld mind. 1 freien Nachbarn hat
                if score < best:
                    best = score
                    feld = _feld
    return feld

def is_allowed_step(old_pos, new_pos, paar):
    """
    Überprüft, ob es möglich ist, eine Linie vom Feld am Index old_pos zum Nachbarfeld
am Index new_pos fortsetzen
    """
    if 0 > new_pos or new_pos >= n*n: # Sicherstellen, dass new_pos im Gitter liegt
        return False
    if old_pos % n == 0 and new_pos % n == n-1: # Sicherstellen, dass der Schritt nicht
den linken Gitterand überschreitet
        return False
    if old_pos % n == n-1 and new_pos % n == 0: # Sicherstellen, dass der Schritt nicht
den rechten Gitterand überschreitet
        return False
    if grid[new_pos] != 0: # Sicherstellen, dass das Feld new_pos frei ist
        return False
    # Sicherstellen, dass keines der Nachbarfelder von new_pos bereits von derselben
Linie durchzogen wird
    if new_pos - n >= 0 and new_pos-n != old_pos:
        if grid[new_pos-n] == paar or grid[new_pos-n] == f".{paar}":
            return False
    if new_pos + n < n*n and new_pos+n != old_pos:
        if grid[new_pos+n] == paar or grid[new_pos+n] == f".{paar}":
            return False
    if new_pos % n != 0 and new_pos-1 != old_pos:
        if grid[new_pos-1] == paar or grid[new_pos-1] == f".{paar}":
            return False
    if new_pos % n != n-1 and new_pos+1 != old_pos:
        if grid[new_pos+1] == paar or grid[new_pos+1] == f".{paar}":
            return False
    return True

# Input-Seed verarbeiten
if seed is None:
    input_seed = []
else:
    input_seed = seed.split(";") # Verarbeiteter Input-Seed
output_seed = [] # In dieser Liste werden Daten für den Output-Seed, der ausgegeben
wird, gespeichert

grid = [0 for i in range(n*n)] # Initialisieren eines leeren Arukone-Gitter
num_paare = 0 # Initialisieren einer Zählervariable, die die Anzahl der platzierten
Paare zählen wird

# Gitter solange füllen, bis kein Platz mehr für neue Paare ist
while True:
    # In jeder Iteration wird ein Zahlenpaar zum Arukone hinzugefügt. Die Schleife wird
abgebrochen, sobald mindestens n/2 Paare platziert wurden und es im Gitter keinen Platz
für weitere Paare mehr gibt
    if num_paare == 0:
```

```
        # Das Anfangsfeld des ersten Zahlenpaars wird anders gewählt als bei späterenm
Zahlenpaaren, damit die erstellten Arukone diverser sind
        if input_seed == []:
            # Wenn noch keine Paare platziert und der Input-Seed das Feld nicht
vorschreibt, wird für das Startfeld des 1. Zahlenpaars ein zufälliges Feld aus der
oberen Hälfte des Arukone gewählt
            start_field = random.randint(0,n*round(n/2))
        else:
            start_field = int(input_seed.pop(0))
            if start_field >= n*n:
                print(f"Ungülter Seed für ein Arukone mit der Gittergröße {n} * {n}")
                exit()
        output_seed.append(str(start_field))
    else:
        start_field = get_start_field() # Ein geeignetes Startfeld wird ermittelt
        if start_field is None:
            break

    num_paare += 1 # Zähler erhöhen
    grid[start_field] = num_paare # Startfeld des Zahlenpaars wird im Gitter markiert

    # Genieren einer Linie, die eine zufällige Länge hat:
    position = start_field
    direction = -n

    # (Maximale) Länge der Linie festlegen. Hierfür wird entweder ein Wert aus dem Seed
oder - wenn der Seed keinen Wert enthält - ein zufällig gewählter Wert gewählt
    if input_seed == []:
        iterations = random.randint(n,2*n) # Durch Ausprobieren hat sich
herausgestellt, dass (n,2*n) ein geeigneter Zahlenbereich für die Linienlänge ist
    else:
        iterations = int(input_seed.pop(0))
    output_seed.append(str(iterations)) # Festgelegte Länge zu Output-Seed hinzufügen
    for i in range(iterations):
        # Bei jeder Iteration wird versucht, die Linie in die Richtung direction
fortzusetzen. Wenn dabei auf ein Hindernis (eine andere Linie, eine Zahl oder den Rand
vom Feld) gestoßen wird, wird die Richtung geändert
        directions = [-n, 1, n, -1]
        new_position = position + direction
        # Die while-Schleife wird ausgeführt, wenn die Linienerweiterung nach
new_position nicht möglich ist:
        while not is_allowed_step(position, new_position, num_paare):
            # Wenn die Linie in die Richtung direction nicht fortgesetzt werden kann,
dann wird solange eine neue Richtung ausprobiert, bis eine Richtung gefunden wurde, in
die sie fortgesetzt werden kann (oder alle Richtungen durchprobiert wurden)
            if directions == []:
                # Wenn alle Richtungen durchprobiert wurden und die Linie nicht
fortgesetzt werden kann, dann wird dies gespeichert und die Schleife abgebrochen
                found_direction = False
                break
            direction = directions.pop(0)
            new_position = position + direction
        else:
            # Dieser Programmteil wird ausgeführt, wenn eine Richtung gefunden wurde,
in die die Linie fortgesetzt werden kann
            found_direction = True

        if found_direction:
```

```
            # Linie kann fortgesetzt werden:
            grid[new_position] = "."+str(num_paare) # Neuen zur Linie gehörenden Punkt
in Gitter markieren
            position = new_position # Position aktualisieren
            continue
        else:
            # Linie kann nicht fortgesetzt werden -> Linie beenden
            break

    grid[position] = num_paare # Endfeld des Zahlenpaars in Gitter markieren

# Gittergröße, Anzahl platzierte Paare, mit Zahlen gefülltes Gitter, Lösung des Rätsels
und Seed ausgeben:
seed = ";".join(output_seed) # Daten für den Seed zu einem zusammenhängenden String
zusammenfügen
str_raetsel = ""
str_loesung = ""
for row in range(n):
    output_raetsel = ""
    output_loesung = ""
    for column in range(n):
        field = str(grid.pop(0))
        output_raetsel += str(0 if "." in field else field) + " "
        output_loesung += str("("+field[1:]+")" if "." in field else "["+field+"]") + "
"
    str_raetsel += output_raetsel + "\n"
    str_loesung += output_loesung + "\n"

print("Generiertes Rätsel:\n"+str(n)+"\n"+str(num_paare)+"\n"+str_raetsel)
print("Lösung des generierten Rätsels:\n"+str_loesung)
print("Seed zum Reproduzieren:\n"+seed)
```