Aufgabe 3: Pancake Sort

Teilnahme-ID: 65433

Bearbeiter/-in dieser Aufgabe:
Paul Franosch

17. April 2023

Inhaltsverzeichnis

1 Lösungsidee

Im Nachfolgenden meint ein sortierter Pfannkuchenstapel einen Pfannkuchenstapel, bei dem der größte Pfannkuchen ganz unten, der zweitgrößte darüber etc. liegt.

1.1 Pfannkuchen-Sortierer

Um Pfannkuchenstapel zu sortieren, wird die Technik des Dynamic Programmings verwendet. Idee ist es, das Problem, also den zu sortierenden Pfannkuchenstapel, in identische Teilprobleme zu zerlegen. Im Kontext der Aufgabe bedeutet das, an allen möglichen Indizes die Wende-und-Essoperation beginnen zu lassen. Der jeweilige Index der Wende-und-Essoperation wird gemerkt. Nach jeder Wende-und-Essoperation wird der resultierende Pfannkuchen gemäß seiner neuen Höhe normiert. Das bedeutet alle Pfannkuchen, die größer als der gegessene Pfannkuchen sind, werden in ihrer Größe um eins reduziert. Damit ist gewährleistet, dass ein Pfannkuchenstapel [9,3,7] genauso behandelt werden kann wie der Stapel [3,1,2]. Dieses Verfahren wird so lange wiederholt, bis der Pfannkuchenstapel sortiert ist. Die möglichen Abfolgen von Wende-und-Essoperationen zur Sortierung eines Pfannkuchenstapels werden gesammelt und abschließend derjenige gewählt, der die wenigsten Wende-und-Essoperationen benötigt. Danach wird der aktuelle Pfannkuchenstapel und die jeweilige beste gefundene Lösung gemerkt. Zu Beginn eines jeden Rekursionsschritts wird überprüft, ob nicht genau dieser Pfannkuchenstapel bereits gelöst wurde. Ist der Pfannkuchenstapel bereits gelöst worden, so wird darauf verzichtet diesen wieder in alle möglichen Pfannkuchenstapel aufzuteilen und es wird direkt die Zugfolge des gelösten Pfannkuchenstapels verwendet.
Dieses Verfahren garantiert einerseits, dass die Zugfolge optimal kurz ist, und zugleich nicht unnötig oft derselbe Pfannkuchenstapel gelöst wird.

1.2 PWUE-Zahlen

Um die PWUE-Zahl einer Pfannkuchenstapelhöhe zu generieren, müssen alle Pfannkuchenstapel der besagten Höhe generiert, sortiert und abschließend bezüglich der Länge ihrer Zugfolge evaluiert werden. Problematisch hierbei ist die schiere Menge der zu prüfenden Pfannkuchenstapel, sowie die Menge der Einträge, die infolge des Dynamic Programming Ansatzes, gemerkt werden müssen.
Für die Anzahl der zu prüfenden Pfannkuchenstapel gilt $h!$ wobei h die Höhe der betrachteten Pfannkuchenstapel ist. Um die zu merkende Datenmenge zu reduzieren, wird die Art und Weise wie Pfannkuchenstapel gelöst werden modifiziert. Ziel ist es nun nicht mehr, die Zugfolge zu finden, sondern lediglich die Anzahl der nötigen Wende-und-Essoperationen. Zudem werden die PWUE-Zahlen dynamisch generiert. Sucht man also bspw. die PWUE-Zahl der Höhe 7, so wird zunächst die PWUE-Zahl der Höhe 1, dann 2, dann 3, etc. generiert, und dementsprechend auch alle Pfannkuchenstapel der Höhe 1, 2, 3 etc. Dieser Bottom-Up Ansatz hat den Vorteil, dass man sich sicher sein kann, dass alle Pfannkuchenstapel der vorherigen Höhen bereits gelöst wurden. Das eröffnet die Möglichkeit Einträge gezielt nicht zu merken. Darauf wird genau dann verzichtet, wenn die Lösung des aktuellen Pfannkuchenstapels derart trivial ist, dass absehbar ist, dass darauf aufbauende Pfannkuchenstapel (also solche, die dem aktuellen Pfannkuchen nach einer Wende-und-Essoperation entsprechen) nicht zu einer PWUE-Zahl führen werden. Beispielsweise 7 hohe Pfannkuchenstapel, welche in einer oder zwei Pfannkuchen Wende-und-Essoperation gelöst werden können. Im vorher verwendeten Top-Down Ansatz würde man bei fehlenden Einträgen davon ausgehen, dass diese schlicht noch nicht berechnet wurden und deswegen fehlen. Jetzt kann aber davon ausgegangen werden, dass diese absichtlich fehlen, schließlich wurden ja alle vorherigen Pfannkuchenstapel gelöst. Dadurch kann durch die *Absenz von Einträgen* Information ausgedrückt werden, ohne kostbaren Speicher aufzuwenden. Konkret werden, alle Einträge von $n-1$ gelöscht.

1.3 13er PWUE-Zahl

Für PWUE-Zahlen kleiner gleich 12 reicht das bis hierhin beschriebene Verfahren aus. Für die PWUE-Zahl 13 wird folgende Optimierung verwendet. Nachdem die PWUE-Zahl einer Zahl n ermittelt wurde, werden alle Einträge gelöscht, welche eine so kurze Zugfolge aufweisen, sodass sie für die PWUE-Zahl von $n+1$ nicht relevant sind. Die Zugfolge ist genau dann zu kurz, wenn sie kleiner gleich $PWUE(n)-k$ ist. K wird im nächsten Absatz genauer erklärt, zwischengeschoben folgende Überlegung.
Angenommen es existiert $PWUE(n+1) < PWUE(n)$, dann wäre man in der Lage einen Pfannkuchenstapel der Höhe $n+1$ zu konstruieren, welcher aus einem worst-case Pfannkuchenstapel der Höhe n

besteht und zusätzlich seinen größten Pfannkuchen ganz unten hat. Somit existiert ein Pfannkuchenstapel der Höhe $n+1$ mit $PWUE(n)$ benötigten Wende-und-Essoperationen, was ein Widerspruch zu der Annahme wäre. Daraus folgt, dass $PWUE(n+1) >= PWUE(n)$ gilt.
Betrachtet wird das Beispiel der PWUE-Zahl von 10. In früheren Schritten wurden kleinere PWUE-Zahlen ermittelt.

Höhe	1	2	3	4	5	6	7	8	9
PWUE	0	1	2	2	3	3	4	4	5

Tabelle 1: PWUE-Zahlen bis 9

Um die PWUE-Zahl von 10 zu ermitteln, sind nur noch alle 9er (gemeint sind Pfannkuchenstapel der Höhe 9) relevant, die in $PWUE(9)$ oder $PWUE(9)-1$ (also 5 oder 4) Wende-und-Essoperationen lösbar sind, weil die neue PWUE-Zahl mindestens so groß ist wie die vorherige und nun eine Wende-und-Essoperation ausgeführt wird. Für diese 9er–Stapel werden entsprechend die 8er gebraucht mit $PWUE(8)$ oder $PWUE(8)-1$ (also 4 oder 3) Wende-und-Essoperationen. Für die 8er–Stapel braucht man nun die 7er mit $PWUE(7)$, $PWUE(7)-1$ und $PWUE(7)-2$, (also 4, 3 oder 2) Wende-und-Essoperationen lösbar sind da sich zwischen 7 und 8 die PWUE-Zahl nicht geändert hat. Je niedriger die betrachtete Pfannkuchenhöhe ist, desto mehr Einträge müssen behalten werden. Der Übersicht halber wird eine Tabelle, auch im Quellcode, verwendet, man könnte diese Werte aber auch errechnen.
Die Höhe innerhalb der Tabelle meint die Höhe des Stapels und die Anzahl beschreibt die Menge der

Höhe h	1	2	3	4	5	6	7	8	9
Anzahl k	6	5	5	4	4	3	3	2	2

Lösungen, welche gemerkt werden müssen. Eine Anzahl von 2 bedeutet alle Lösungen, welche $PWUE(h)$ und $PWUE(h)-1$ Wende-und-Essoperationen benötigen.
Somit können einige Einträge gespart werden.

2 Umsetzung

Die Implementierung erfolgt in Java 17.
Die Pfannkuchenstapel werden als Bytearrays dargestellt. Um eine PWUE-Zahl einer großen Höhe zu finden, müssen sehr viele Pfannkuchenstapel sortiert werden. Um diesen Prozess zu Parallelisieren wird die Reactive Streams [1] Bibliothek verwendet.

Die in 1.3 angesprochene Tabelle sieht wie folgt aus:

```
private final static Byte[][] cleanseOperations = new Byte[][]{
        new Byte[]{}, // 0
        new Byte[]{}, // 1
        new Byte[]{}, // 2
        new Byte[]{3, 2}, // 3
        new Byte[]{3, 2, 2}, // 4
        new Byte[]{4, 3, 3, 2}, // 5
        new Byte[]{4, 3, 3, 2, 2}, // 6
        new Byte[]{5, 4, 4, 3, 3, 2}, // 7
        new Byte[]{5, 4, 4, 3, 3, 2, 2},// 8
        new Byte[]{6, 5, 5, 4, 4, 3, 3, 2}, // 9
        new Byte[]{6, 5, 5, 4, 4, 3, 3, 2, 2}, // 10
        new Byte[]{7, 6, 6, 5, 5, 4, 4, 3, 3, 2}, // 11
        new Byte[]{7, 6, 6, 5, 5, 4, 4, 3, 3, 2, 2}, // 12
        new Byte[]{8, 7, 7, 6, 6, 5, 5, 4, 4, 3, 3, 2} // 13
};
```

Diese Tabelle wird in der Methode #cleanse verwendet um die oben erklärten Fälle zu löschen.

```
private void cleanse(int current, int max) {
```

[1] https://github.com/reactive-streams/reactive-streams-jvm

```
    this.flippingOrderLengthMap.entrySet().removeIf(entry -> {
        int entryHeight = entry.getKey().getPancakes().length;
        int entryValue = entry.getValue();

        if (entryHeight <= current - 1) {
            return true;
        }

        int pwue = this.heightToPWUE.get(entryHeight).get();
        return entryValue <= pwue - cleanseOperations[max][current - 1];
    });

}
```

Um alle Permutationen von Pfannkuchenstapel einer gewissen Höhe zu bekommen wird die Methode #perumte verwendet.

```
    private void permute(List<Byte> pancakes, int height, int initialHeight) {
        for (int i = height; i < pancakes.size(); i++) {
            java.util.Collections.swap(pancakes, i, height);
            permute(pancakes, height + 1, initialHeight);
            java.util.Collections.swap(pancakes, height, i);
        }
        if (height == pancakes.size() - 1) {
            this.sortAndApplyResults(pancakeStack, height, initialHeight)
                    .subscribeOn(Schedulers.parallel())
                    .doOnNext(integer -> finishedTasks.incrementAndGet())
                    .subscribe();
        }
    }
```

Es werden $\frac{1}{n}$ der Pfannkuchen nicht betrachtet, da diese mit dem Pfannkuchen mit der Größe n an unterster Stelle beginnnen und somit einem Pfannkuchenstapel der Höhe $n-1$ entsprechen.

3 Komplexitätsbetrachtung

Zunächst ein Beispiel. Ein Pfannkuchenstapel der Höhe 4 soll sortiert werden. Dafür gibt es zunächst 4 mögliche Wende-und-Essoperationen. Nun gibt es maximal 4–3er Stapel. Auf diese wird wiederum jede mögliche Wende-und-Essoperation ausgeführt. Für einen einzelnen 3er-Stapel sind das 3 resultierende 2er-Stapel. Für 4–3er Stapel sind es demnach 12–2er Stapel. Diese werden wiederum zu 24–1r SStapel". Allgemein formuliert sind das $1 + n + n * (n-1) + n * (n-1) * (n-2) + ...$ oder $\sum_{k=1}^{n} \frac{n!}{k!}$. Soweit der Brute-Force Ansatz. In der Dynamischen Programmierung wird unteranderem ausgenutzt, dass es nur maximal $k!$ verschiedene Pfannkuchenstapel der Höhe k gibt. Dementsprechend ist für ein jeweiliges k den größeren der beiden Werte aus obigem Summanden und $k!$. Im obigen Beispiel gab es 24 Einser, tatsächlich gibt es aber natürlich nur einen Einser (normiert).

4 Beispiele

4.1 Beispiel Dateien

Es folgen die Pfannkuchenstapel der Beispieldateien und die jeweilige Programmausgabe.

4.1.1 pancake0.txt

Wende-und-Essoperation an Index 0: [1, 5, 4, 2, 3] -> [3, 2, 4, 5]
Wende-und-Essoperation an Index 0: [3, 2, 4, 5] -> [5, 4, 2]
Ursprünglicher Pfannkuchenstapel -> Sortierter Pfannkuchenstapel -> Indizes benötigter Operationen
[1, 5, 4, 2, 3]
[5, 4, 2]

[0, 0]
Benötigte Wende-und-Essoperationen 2

4.1.2 pancake1.txt

Wende-und-Essoperation an Index 1: [5, 2, 4, 7, 1, 3, 6] -> [5, 6, 3, 1, 7, 4]
Wende-und-Essoperation an Index 1: [5, 6, 3, 1, 7, 4] -> [5, 4, 7, 1, 3]
Wende-und-Essoperation an Index 2: [5, 4, 7, 1, 3] -> [5, 4, 3, 1]
Ursprünglicher Pfannkuchenstapel -> Sortierter Pfannkuchenstapel -> Indizes benötigter Operationen
[5, 2, 4, 7, 1, 3, 6]
[5, 4, 3, 1]
[1, 1, 2]
Benötigte Wende-und-Essoperationen 3

4.1.3 pancake2.txt

Wende-und-Essoperation an Index 4: [2, 4, 6, 3, 5, 7, 1, 8] -> [2, 4, 6, 3, 8, 1, 7]
Wende-und-Essoperation an Index 5: [2, 4, 6, 3, 8, 1, 7] -> [2, 4, 6, 3, 8, 7]
Wende-und-Essoperation an Index 3: [2, 4, 6, 3, 8, 7] -> [2, 4, 6, 7, 8]
Wende-und-Essoperation an Index 0: [2, 4, 6, 7, 8] -> [8, 7, 6, 4]
Ursprünglicher Pfannkuchenstapel -> Sortierter Pfannkuchenstapel -> Indizes benötigter Operationen
[2, 4, 6, 3, 5, 7, 1, 8]
[8, 7, 6, 4]
[4, 5, 3, 0]
Benötigte Wende-und-Essoperationen 4

4.1.4 pancake3.txt

Wende-und-Essoperation an Index 0: [6, 3, 7, 9, 2, 8, 4, 11, 1, 10, 5] -> [5, 10, 1, 11, 4, 8, 2, 9, 7, 3]
Wende-und-Essoperation an Index 5: [5, 10, 1, 11, 4, 8, 2, 9, 7, 3] -> [5, 10, 1, 11, 4, 3, 7, 9, 2]
Wende-und-Essoperation an Index 8: [5, 10, 1, 11, 4, 3, 7, 9, 2] -> [5, 10, 1, 11, 4, 3, 7, 9]
Wende-und-Essoperation an Index 2: [5, 10, 1, 11, 4, 3, 7, 9] -> [5, 10, 9, 7, 3, 4, 11]
Wende-und-Essoperation an Index 0: [5, 10, 9, 7, 3, 4, 11] -> [11, 4, 3, 7, 9, 10]
Wende-und-Essoperation an Index 1: [11, 4, 3, 7, 9, 10] -> [11, 10, 9, 7, 3]
Ursprünglicher Pfannkuchenstapel -> Sortierter Pfannkuchenstapel -> Indizes benötigter Operationen
[6, 3, 7, 9, 2, 8, 4, 11, 1, 10, 5]
[11, 10, 9, 7, 3]
[0, 5, 8, 2, 0, 1]
Benötigte Wende-und-Essoperationen 6

4.1.5 pancake4.txt

Wende-und-Essoperation an Index 1: [2, 8, 3, 9, 12, 13, 1, 6, 10, 5, 11, 4, 7] -> [2, 7, 4, 11, 5, 10, 6, 1, 13, 12, 9, 3]
Wende-und-Essoperation an Index 7: [2, 7, 4, 11, 5, 10, 6, 1, 13, 12, 9, 3] -> [2, 7, 4, 11, 5, 10, 6, 3, 9, 12, 13]
Wende-und-Essoperation an Index 2: [2, 7, 4, 11, 5, 10, 6, 3, 9, 12, 13] -> [2, 7, 13, 12, 9, 3, 6, 10, 5, 11]
Wende-und-Essoperation an Index 1: [2, 7, 13, 12, 9, 3, 6, 10, 5, 11] -> [2, 11, 5, 10, 6, 3, 9, 12, 13]
Wende-und-Essoperation an Index 0: [2, 11, 5, 10, 6, 3, 9, 12, 13] -> [13, 12, 9, 3, 6, 10, 5, 11]
Wende-und-Essoperation an Index 6: [13, 12, 9, 3, 6, 10, 5, 11] -> [13, 12, 9, 3, 6, 10, 11]
Wende-und-Essoperation an Index 2: [13, 12, 9, 3, 6, 10, 11] -> [13, 12, 11, 10, 6, 3]
Ursprünglicher Pfannkuchenstapel -> Sortierter Pfannkuchenstapel -> Indizes benötigter Operationen
[2, 8, 3, 9, 12, 13, 1, 6, 10, 5, 11, 4, 7]
[13, 12, 11, 10, 6, 3]
[1, 7, 2, 1, 0, 6, 2]
Benötigte Wende-und-Essoperationen 7

4.1.6 pancake5.txt

Wende-und-Essoperation an Index 13: [11, 5, 6, 12, 1, 14, 9, 7, 3, 2, 8, 10, 13, 4] -> [11, 5, 6, 12, 1, 14, 9, 7, 3, 2, 8, 10, 13]
Wende-und-Essoperation an Index 0: [11, 5, 6, 12, 1, 14, 9, 7, 3, 2, 8, 10, 13] -> [13, 10, 8, 2, 3, 7, 9, 14, 1, 12, 6, 5]
Wende-und-Essoperation an Index 5: [13, 10, 8, 2, 3, 7, 9, 14, 1, 12, 6, 5] -> [13, 10, 8, 2, 3, 5, 6, 12, 1, 14, 9]
Wende-und-Essoperation an Index 2: [13, 10, 8, 2, 3, 5, 6, 12, 1, 14, 9] -> [13, 10, 9, 14, 1, 12, 6, 5, 3, 2]
Wende-und-Essoperation an Index 5: [13, 10, 9, 14, 1, 12, 6, 5, 3, 2] -> [13, 10, 9, 14, 1, 2, 3, 5, 6]
Wende-und-Essoperation an Index 3: [13, 10, 9, 14, 1, 2, 3, 5, 6] -> [13, 10, 9, 6, 5, 3, 2, 1]
Ursprünglicher Pfannkuchenstapel -> Sortierter Pfannkuchenstapel -> Indizes benötigter Operationen
[11, 5, 6, 12, 1, 14, 9, 7, 3, 2, 8, 10, 13, 4]
[13, 10, 9, 6, 5, 3, 2, 1]
[13, 0, 5, 2, 5, 3]
Benötigte Wende-und-Essoperationen 6

4.1.7 pancake6.txt

Wende-und-Essoperation an Index 1: [6, 10, 5, 9, 3, 11, 7, 15, 1, 2, 13, 12, 4, 8, 14] -> [6, 14, 8, 4, 12, 13, 2, 1, 15, 7, 11, 3, 9, 5]
Wende-und-Essoperation an Index 4: [6, 14, 8, 4, 12, 13, 2, 1, 15, 7, 11, 3, 9, 5] -> [6, 14, 8, 4, 5, 9, 3, 11, 7, 15, 1, 2, 13]
Wende-und-Essoperation an Index 6: [6, 14, 8, 4, 5, 9, 3, 11, 7, 15, 1, 2, 13] -> [6, 14, 8, 4, 5, 9, 13, 2, 1, 15, 7, 11]
Wende-und-Essoperation an Index 2: [6, 14, 8, 4, 5, 9, 13, 2, 1, 15, 7, 11] -> [6, 14, 11, 7, 15, 1, 2, 13, 9, 5, 4]
Wende-und-Essoperation an Index 7: [6, 14, 11, 7, 15, 1, 2, 13, 9, 5, 4] -> [6, 14, 11, 7, 15, 1, 2, 4, 5, 9]
Wende-und-Essoperation an Index 3: [6, 14, 11, 7, 15, 1, 2, 4, 5, 9] -> [6, 14, 11, 9, 5, 4, 2, 1, 15]
Wende-und-Essoperation an Index 0: [6, 14, 11, 9, 5, 4, 2, 1, 15] -> [15, 1, 2, 4, 5, 9, 11, 14]
Wende-und-Essoperation an Index 0: [15, 1, 2, 4, 5, 9, 11, 14] -> [14, 11, 9, 5, 4, 2, 1]
Ursprünglicher Pfannkuchenstapel -> Sortierter Pfannkuchenstapel -> Indizes benötigter Operationen
[6, 10, 5, 9, 3, 11, 7, 15, 1, 2, 13, 12, 4, 8, 14]
[14, 11, 9, 5, 4, 2, 1]
[1, 4, 6, 2, 7, 3, 0, 0]
Benötigte Wende-und-Essoperationen 8

4.1.8 pancake7.txt

Wende-und-Essoperation an Index 5: [11, 16, 14, 1, 9, 12, 4, 2, 6, 13, 7, 3, 15, 10, 5, 8] -> [11, 16, 14, 1, 9, 8, 5, 10, 15, 3, 7, 13, 6, 2, 4]
Wende-und-Essoperation an Index 0: [11, 16, 14, 1, 9, 8, 5, 10, 15, 3, 7, 13, 6, 2, 4] -> [4, 2, 6, 13, 7, 3, 15, 10, 5, 8, 9, 1, 14, 16]
Wende-und-Essoperation an Index 0: [4, 2, 6, 13, 7, 3, 15, 10, 5, 8, 9, 1, 14, 16] -> [16, 14, 1, 9, 8, 5, 10, 15, 3, 7, 13, 6, 2]
Wende-und-Essoperation an Index 8: [16, 14, 1, 9, 8, 5, 10, 15, 3, 7, 13, 6, 2] -> [16, 14, 1, 9, 8, 5, 10, 15, 2, 6, 13, 7]
Wende-und-Essoperation an Index 10: [16, 14, 1, 9, 8, 5, 10, 15, 2, 6, 13, 7] -> [16, 14, 1, 9, 8, 5, 10, 15, 2, 6, 7]
Wende-und-Essoperation an Index 5: [16, 14, 1, 9, 8, 5, 10, 15, 2, 6, 7] -> [16, 14, 1, 9, 8, 7, 6, 2, 15, 10]
Wende-und-Essoperation an Index 2: [16, 14, 1, 9, 8, 7, 6, 2, 15, 10] -> [16, 14, 10, 15, 2, 6, 7, 8, 9]
Wende-und-Essoperation an Index 3: [16, 14, 10, 15, 2, 6, 7, 8, 9] -> [16, 14, 10, 9, 8, 7, 6, 2]
Ursprünglicher Pfannkuchenstapel -> Sortierter Pfannkuchenstapel -> Indizes benötigter Operationen
[11, 16, 14, 1, 9, 12, 4, 2, 6, 13, 7, 3, 15, 10, 5, 8]
[16, 14, 10, 9, 8, 7, 6, 2]
[5, 0, 0, 8, 10, 5, 2, 3]
Benötigte Wende-und-Essoperationen 8

4.2 Weitere Pfannkuchenstapel

4.2.1 Höhe 18

Wende-und-Essoperation an Index 3: [17, 10, 8, 14, 12, 6, 11, 5, 4, 9, 18, 2, 1, 15, 7, 3, 16, 13] -> [17, 10, 8, 13, 16, 3, 7, 15, 1, 2, 18, 9, 4, 5, 11, 6, 12]
Wende-und-Essoperation an Index 1: [17, 10, 8, 13, 16, 3, 7, 15, 1, 2, 18, 9, 4, 5, 11, 6, 12] -> [17, 12, 6, 11, 5, 4, 9, 18, 2, 1, 15, 7, 3, 16, 13, 8]
Wende-und-Essoperation an Index 12: [17, 12, 6, 11, 5, 4, 9, 18, 2, 1, 15, 7, 3, 16, 13, 8] -> [17, 12, 6, 11, 5, 4, 9, 18, 2, 1, 15, 7, 8, 13, 16]
Wende-und-Essoperation an Index 7: [17, 12, 6, 11, 5, 4, 9, 18, 2, 1, 15, 7, 8, 13, 16] -> [17, 12, 6, 11, 5, 4, 9, 16, 13, 8, 7, 15, 1, 2]
Wende-und-Essoperation an Index 6: [17, 12, 6, 11, 5, 4, 9, 16, 13, 8, 7, 15, 1, 2] -> [17, 12, 6, 11, 5, 4, 2, 1, 15, 7, 8, 13, 16]
Wende-und-Essoperation an Index 1: [17, 12, 6, 11, 5, 4, 2, 1, 15, 7, 8, 13, 16] -> [17, 16, 13, 8, 7, 15, 1, 2, 4, 5, 11, 6]
Wende-und-Essoperation an Index 10: [17, 16, 13, 8, 7, 15, 1, 2, 4, 5, 11, 6] -> [17, 16, 13, 8, 7, 15, 1, 2, 4, 5, 6]
Wende-und-Essoperation an Index 5: [17, 16, 13, 8, 7, 15, 1, 2, 4, 5, 6] -> [17, 16, 13, 8, 7, 6, 5, 4, 2, 1]
Ursprünglicher Pfannkuchenstapel -> Sortierter Pfannkuchenstapel -> Indizes benötigter Operationen
[17, 10, 8, 14, 12, 6, 11, 5, 4, 9, 18, 2, 1, 15, 7, 3, 16, 13]
[17, 16, 13, 8, 7, 6, 5, 4, 2, 1]
[3, 1, 12, 7, 6, 1, 10, 5]
Benötigte Wende-und-Essoperationen 8

4.3 PWUE-Zahlen

4.3.1 Höhe 1

PWUE of number 1 is 0
Pfannkuchenstapel -> Sortierter Pfannkuchenstapel -> Indizes benötigter Operationen
Example Worstcase Pancakestack
[1]
[1]
[]
Sorter map contains 0 entries
Timings report
Time spend finding PWUE nr 2 ms

4.3.2 Höhe 2

PWUE of number 2 is 1
Pfannkuchenstapel -> Sortierter Pfannkuchenstapel -> Indizes benötigter Operationen
Example Worstcase Pancakestack
[1, 2]
[2]
[0]
Sorter map contains 1 entries
Timings report
Time spend finding PWUE nr 1 ms

4.3.3 Höhe 3

PWUE of number 3 is 2
Pfannkuchenstapel -> Sortierter Pfannkuchenstapel -> Indizes benötigter Operationen
Example Worstcase Pancakestack
[1, 3, 2]
[1]
[1, 1]
Sorter map contains 6 entries

Timings report
Time spend finding PWUE nr 6 ms

4.3.4 Höhe 4

PWUE of number 4 is 2
Pfannkuchenstapel -> Sortierter Pfannkuchenstapel -> Indizes benötigter Operationen
Example Worstcase Pancakestack
[2, 4, 1, 3]
[4, 1]
[0, 0]
Sorter map contains 5 entries
Timings report
Time spend finding PWUE nr 13 ms

4.3.5 Höhe 5

PWUE of number 5 is 3
Pfannkuchenstapel -> Sortierter Pfannkuchenstapel -> Indizes benötigter Operationen
Example Worstcase Pancakestack
[1, 4, 5, 2, 3]
[4, 3]
[2, 0, 0]
Sorter map contains 16 entries
Timings report
Time spend finding PWUE nr 19 ms

4.3.6 Höhe 6

PWUE of number 6 is 3
Pfannkuchenstapel -> Sortierter Pfannkuchenstapel -> Indizes benötigter Operationen
Example Worstcase Pancakestack
[4, 1, 6, 2, 5, 3]
[5, 3, 1]
[3, 0, 2]
Sorter map contains 24 entries
Timings report
Time spend finding PWUE nr 94 ms

4.3.7 Höhe 7

PWUE of number 7 is 4
Pfannkuchenstapel -> Sortierter Pfannkuchenstapel -> Indizes benötigter Operationen
Example Worstcase Pancakestack
[4, 6, 7, 5, 1, 3, 2]
[7, 5, 2]
[1, 0, 3, 2]
Sorter map contains 86 entries
Timings report
Time spend finding PWUE nr 49 ms

4.3.8 Höhe 8

PWUE of number 8 is 5
Pfannkuchenstapel -> Sortierter Pfannkuchenstapel -> Indizes benötigter Operationen
Example Worstcase Pancakestack
[2, 4, 8, 1, 5, 7, 3, 6]
[8, 4, 3]
[5, 4, 0, 2, 0]
Sorter map contains 2666 entries

Timings report
Time spend finding PWUE nr 176 ms

4.3.9 Höhe 9

PWUE of number 9 is 5
Pfannkuchenstapel -> Sortierter Pfannkuchenstapel -> Indizes benötigter Operationen
Example Worstcase Pancakestack
[3, 7, 1, 9, 2, 6, 5, 8, 4]
[6, 5, 4, 1]
[1, 2, 4, 0, 2]
Sorter map contains 1705 entries
Timings report
Time spend finding PWUE nr 1123 ms

4.3.10 Höhe 10

PWUE of number 10 is 6
Pfannkuchenstapel -> Sortierter Pfannkuchenstapel -> Indizes benötigter Operationen
Example Worstcase Pancakestack
[5, 10, 1, 4, 9, 3, 6, 8, 2, 7]
[5, 4, 3, 2]
[2, 1, 2, 4, 5, 2]
Example Worstcase Pancakestack
[1, 6, 9, 3, 7, 2, 10, 4, 8, 5]
[10, 8, 7, 6]
[3, 3, 6, 4, 0, 3]
Example Worstcase Pancakestack
[3, 2, 8, 5, 9, 1, 6, 10, 4, 7]
[5, 4, 2, 1]
[2, 0, 4, 5, 1, 3]
Example Worstcase Pancakestack
[1, 5, 8, 4, 10, 2, 6, 9, 3, 7]
[8, 7, 6, 2]
[1, 3, 2, 5, 0, 1]
Example Worstcase Pancakestack
[2, 6, 10, 1, 7, 4, 9, 3, 8, 5]
[5, 4, 3, 1]
[0, 7, 1, 5, 3, 1]
Example Worstcase Pancakestack
[1, 5, 10, 6, 8, 4, 7, 3, 9, 2]
[10, 8, 4, 2]
[1, 0, 5, 1, 1, 3]
Example Worstcase Pancakestack
[2, 6, 10, 5, 9, 4, 7, 3, 8, 1]
[10, 9, 4, 1]
[1, 0, 5, 1, 1, 3]
Example Worstcase Pancakestack
[4, 6, 8, 3, 7, 5, 10, 2, 9, 1]
[10, 7, 3, 1]
[2, 3, 5, 0, 1, 1]
Example Worstcase Pancakestack
[2, 5, 10, 3, 8, 1, 6, 9, 4, 7]
[9, 7, 3, 1]
[1, 6, 6, 2, 0, 1]
Example Worstcase Pancakestack
[5, 3, 7, 2, 6, 9, 1, 8, 10, 4]
[10, 8, 7, 2]
[4, 4, 0, 2, 2, 2]

Sorter map contains 254385 entries
Timings report
Time spend finding PWUE nr 9721 ms

4.3.11 Höhe 11

PWUE of number 11 is 6
Pfannkuchenstapel -> Sortierter Pfannkuchenstapel -> Indizes benötigter Operationen
Example Worstcase Pancakestack
[3, 9, 7, 1, 10, 4, 8, 5, 11, 2, 6]
[9, 8, 4, 2, 1]
[0, 0, 7, 3, 4, 1]
Example Worstcase Pancakestack
[5, 11, 3, 10, 1, 8, 9, 6, 2, 7, 4]
[11, 10, 8, 7, 2]
[0, 0, 5, 1, 5, 1]
Example Worstcase Pancakestack
[3, 5, 11, 8, 2, 10, 7, 4, 1, 9, 6]
[11, 10, 7, 4, 1]
[1, 0, 7, 2, 1, 5]
Example Worstcase Pancakestack
[3, 8, 4, 1, 7, 2, 5, 9, 11, 6, 10]
[11, 10, 8, 5, 2]
[2, 3, 3, 0, 3, 0]
Example Worstcase Pancakestack
[3, 6, 1, 9, 7, 2, 10, 4, 8, 11, 5]
[11, 8, 4, 2, 1]
[0, 4, 6, 0, 0, 3]
Example Worstcase Pancakestack
[6, 8, 11, 4, 5, 1, 7, 2, 10, 3, 9]
[11, 8, 5, 3, 2]
[0, 7, 0, 3, 6, 4]
Example Worstcase Pancakestack
[5, 9, 4, 10, 2, 7, 11, 1, 8, 3, 6]
[11, 9, 4, 3, 2]
[0, 5, 2, 7, 0, 3]
Example Worstcase Pancakestack
[2, 1, 11, 10, 8, 4, 7, 3, 6, 9, 5]
[11, 10, 8, 6, 3]
[0, 1, 5, 0, 6, 0]
Example Worstcase Pancakestack
[3, 4, 10, 7, 1, 9, 5, 8, 2, 6, 11]
[11, 10, 8, 5, 1]
[0, 2, 2, 1, 2, 4]
Example Worstcase Pancakestack
[2, 4, 9, 11, 3, 6, 8, 1, 7, 5, 10]
[10, 5, 4, 3, 1]
[0, 5, 7, 4, 5, 2]
Sorter map contains 49110 entries
Timings report
Time spend finding PWUE nr 129081 ms

4.3.12 Höhe 12

PWUE of number 12 is 7
Pfannkuchenstapel -> Sortierter Pfannkuchenstapel -> Indizes benötigter Operationen
Example Worstcase Pancakestack
[4, 9, 5, 11, 8, 3, 10, 6, 2, 7, 12, 1]
[12, 11, 10, 6, 2]

[0, 0, 4, 3, 0, 6, 1]
Example Worstcase Pancakestack
[2, 12, 3, 8, 11, 7, 4, 9, 6, 1, 10, 5]
[12, 11, 9, 6, 1]
[3, 8, 9, 3, 0, 5, 0]
Example Worstcase Pancakestack
[8, 5, 1, 11, 2, 7, 12, 3, 6, 10, 4, 9]
[11, 10, 9, 7, 3]
[2, 0, 1, 7, 3, 1, 2]
Example Worstcase Pancakestack
[7, 3, 12, 4, 9, 5, 11, 1, 6, 10, 2, 8]
[11, 10, 9, 4, 3]
[0, 4, 0, 6, 1, 2, 4]
Example Worstcase Pancakestack
[4, 7, 1, 9, 5, 12, 8, 2, 10, 3, 11, 6]
[12, 9, 7, 6, 3]
[2, 9, 6, 3, 3, 0, 2]
Example Worstcase Pancakestack
[1, 8, 4, 9, 6, 11, 3, 7, 10, 2, 12, 5]
[12, 11, 10, 7, 4]
[0, 0, 5, 3, 5, 0, 4]
Example Worstcase Pancakestack
[1, 7, 9, 4, 11, 5, 8, 3, 12, 6, 10, 2]
[12, 6, 5, 4, 2]
[1, 2, 6, 4, 4, 0, 4]
Example Worstcase Pancakestack
[5, 7, 11, 4, 9, 3, 12, 6, 8, 1, 10, 2]
[12, 10, 9, 8, 7]
[0, 4, 2, 5, 0, 4, 0]
Example Worstcase Pancakestack
[2, 12, 3, 10, 4, 6, 11, 5, 8, 1, 9, 7]
[7, 5, 4, 3, 1]
[0, 3, 7, 3, 6, 3, 1]
Example Worstcase Pancakestack
[3, 6, 11, 4, 7, 9, 1, 8, 10, 2, 5, 12]
[12, 10, 8, 7, 4]
[0, 1, 5, 5, 1, 3, 3]
Sorter map contains 2512306 entries
Timings report
Time spend finding PWUE nr 1303151 ms

4.3.13 Höhe 13

PWUE of number 13 is 7
Pfannkuchenstapel -> Sortierter Pfannkuchenstapel -> Indizes benötigter Wende-und-Essoperationen
Example Worstcase Pancakestack
Ursprünglicher Pfannkuchenstapel -> Sortierter Pfannkuchenstapel -> Indizes benötigter Operationen
[12, 9, 3, 11, 1, 6, 8, 2, 5, 10, 4, 7, 13]
[11, 9, 8, 6, 5, 4]
[11, 2, 0, 1, 6, 2, 5]
Example Worstcase Pancakestack
Ursprünglicher Pfannkuchenstapel -> Sortierter Pfannkuchenstapel -> Indizes benötigter Operationen
[5, 8, 11, 9, 6, 1, 12, 4, 2, 10, 3, 13, 7]
[13, 12, 11, 9, 6, 2]
[0, 0, 0, 2, 5, 1, 2]
Example Worstcase Pancakestack
Ursprünglicher Pfannkuchenstapel -> Sortierter Pfannkuchenstapel -> Indizes benötigter Operationen
[6, 11, 4, 3, 8, 12, 5, 13, 2, 9, 10, 7, 1]
[13, 12, 10, 9, 4, 3]

[1, 5, 0, 9, 1, 1, 2]
Example Worstcase Pancakestack
Ursprünglicher Pfannkuchenstapel -> Sortierter Pfannkuchenstapel -> Indizes benötigter Operationen
[5, 1, 2, 10, 8, 3, 13, 4, 12, 6, 9, 7, 11]
[10, 8, 7, 6, 4, 1]
[2, 0, 3, 8, 4, 6, 2]
Example Worstcase Pancakestack
Ursprünglicher Pfannkuchenstapel -> Sortierter Pfannkuchenstapel -> Indizes benötigter Operationen
[8, 7, 4, 9, 6, 3, 5, 1, 11, 10, 13, 2, 12]
[13, 10, 9, 6, 3, 1]
[0, 0, 0, 0, 0, 2, 5]
Example Worstcase Pancakestack
Ursprünglicher Pfannkuchenstapel -> Sortierter Pfannkuchenstapel -> Indizes benötigter Operationen
[11, 3, 6, 4, 7, 13, 5, 12, 9, 1, 10, 2, 8]
[12, 9, 7, 6, 3, 2]
[0, 2, 6, 4, 7, 4, 0]
Example Worstcase Pancakestack
Ursprünglicher Pfannkuchenstapel -> Sortierter Pfannkuchenstapel -> Indizes benötigter Operationen
[3, 10, 4, 6, 13, 5, 11, 2, 8, 9, 12, 1, 7]
[12, 9, 8, 7, 6, 5]
[1, 6, 6, 8, 2, 0, 3]
Example Worstcase Pancakestack
Ursprünglicher Pfannkuchenstapel -> Sortierter Pfannkuchenstapel -> Indizes benötigter Operationen
[1, 2, 10, 5, 11, 8, 6, 12, 7, 13, 4, 3, 9]
[13, 7, 6, 5, 3, 2]
[0, 2, 6, 8, 5, 0, 4]
Example Worstcase Pancakestack
Ursprünglicher Pfannkuchenstapel -> Sortierter Pfannkuchenstapel -> Indizes benötigter Operationen
[4, 10, 2, 8, 9, 6, 1, 11, 7, 12, 5, 3, 13]
[13, 12, 9, 8, 2, 1]
[0, 2, 1, 9, 5, 2, 6]
Example Worstcase Pancakestack
Ursprünglicher Pfannkuchenstapel -> Sortierter Pfannkuchenstapel -> Indizes benötigter Operationen
[3, 8, 2, 9, 7, 4, 10, 13, 6, 5, 11, 1, 12]
[13, 12, 11, 9, 7, 4]
[1, 0, 9, 6, 4, 0, 3]
Sorter map contains 516671 entries
Timings report
Time spend finding PWUE nr 26150971 ms
Du befindest dich nun im PWUE-Zahl Modus.
Bitte gib die Höhe des Pfannkuchenstapels an für die die PWUE-Zahl ermittelt werden soll!

5 Quellcode

```
@Getter
@Setter
@AllArgsConstructor
public class PancakeStack implements Cloneable {
    private PancakeStackData normalizedPancakes; // bottom to top
    private PancakeStackData pancakes; // bottom to top, using actual pancake sizes

    public PancakeStack(Byte[] pancakes) {
        this.normalizedPancakes = new PancakeStackData(pancakes);
        this.pancakes = new PancakeStackData(pancakes);
    }
```

```
    public boolean isSolved() {
        int last = - 1;
        for (final Byte pancake : this.normalizedPancakes.getPancakes()) {
            if (last == - 1) {
                last = pancake;
                continue;
            }
            if (last <= pancake) {
                return false;
            }
            last = pancake;
        }
        return true;
    }
}

@Getter
@AllArgsConstructor
public class PancakeStackData implements Cloneable {
    private Byte[] pancakes; // bottom to top
}


public class PancakeStackSorter {
    private final PancakeFlipper pancakeFlipper;
    private final PancakeFlippingOrderApplier pancakeFlippingOrderApplier;

    private final Map<PancakeStackData, FlippingOrder> flippingOrderMap = new
        ConcurrentHashMap<>();

    public PancakeStackSorter(final PancakeFlipper pancakeFlipper, final
        PancakeFlippingOrderApplier pancakeFlippingOrderApplier) {
        this.pancakeFlipper = pancakeFlipper;
        this.pancakeFlippingOrderApplier = pancakeFlippingOrderApplier;

    }

    public int getMapEntryCount() {
        return this.flippingOrderMap.size();
    }


    public PancakeStackSortingResult sort(PancakeStack pancakeStack) {
        FlippingOrder flippingOrder = this.optimalFlippingOrder(pancakeStack, new
            FlippingOrder());
        PancakeStack solved = this.pancakeFlippingOrderApplier.apply(pancakeStack.clone(),
            flippingOrder);
        return PancakeStackSortingResult.of(flippingOrder, solved, pancakeStack);
    }

    public PancakeStackSortingResult sort(PancakeStack pancakeStack, boolean debugPrints) {
        FlippingOrder flippingOrder = this.optimalFlippingOrder(pancakeStack, new
            FlippingOrder());
        PancakeStack solved = debugPrints ?
                this.pancakeFlippingOrderApplier.applyAndPrintDebug(pancakeStack.clone(),
                    flippingOrder) :
                this.pancakeFlippingOrderApplier.apply(pancakeStack.clone(), flippingOrder);
        return PancakeStackSortingResult.of(flippingOrder, solved, pancakeStack);
    }

    private FlippingOrder optimalFlippingOrder(PancakeStack pancakeStack, FlippingOrder
        flippingOrder) {
```

```
        if (pancakeStack.isSolved()) {
            return flippingOrder;
        }
        if (this.flippingOrderMap.containsKey(pancakeStack.getNormalizedPancakes())) {
            return
                flippingOrder.append(this.flippingOrderMap.get(pancakeStack.getNormalizedPancakes()));
        }
        Set<FlippingOrder> options = new HashSet<>();
        for (byte i = 0; i < pancakeStack.getNormalizedPancakes().getPancakes().length; i++) {
            PancakeStack clonedPancakeStack = pancakeStack.clone();
            this.pancakeFlipper.flip(clonedPancakeStack, i);
            FlippingOrder clonedFlippingOrder = flippingOrder.clone();
            clonedFlippingOrder.add(i);
            options.add(this.optimalFlippingOrder(clonedPancakeStack, clonedFlippingOrder));
        }

        FlippingOrder best = findBest(options);

        this.flippingOrderMap.put(pancakeStack.getNormalizedPancakes(),
                slice(calcSlicingStart(best, pancakeStack),
                        best));
        return best;
    }

    private FlippingOrder findBest(Set<FlippingOrder> flippingOrders) {
        int length = Integer.MAX_VALUE;
        FlippingOrder best = null;
        for (FlippingOrder flippingOrder : flippingOrders) {
            if (flippingOrder.getFlippingOperations().size() < length) {
                length = flippingOrder.getFlippingOperations().size();
                best = flippingOrder;
            }
        }
        return best;
    }

    private FlippingOrder slice(int from, FlippingOrder flippingOrder) {
        if (from == 0) {
            return flippingOrder.clone();
        }
        FlippingOrder result = new FlippingOrder();
        for (int i = from; i < flippingOrder.getFlippingOperations().size(); i++) {
            result.add(flippingOrder.getFlippingOperations().get(i));
        }
        return result;
    }

    private int calcSlicingStart(FlippingOrder flippingOrder, PancakeStack pancakeStack) {
        for (int i = flippingOrder.getFlippingOperations().size() - 1; i >= 0; i--) {
            PancakeStack applied = this
                    .pancakeFlippingOrderApplier
                    .apply(pancakeStack.clone(), slice(i, flippingOrder));
            if (applied.isSolved()) {
                return i;
            }
        }
        throw new IllegalStateException();
    }
}

public class FlippingOrder implements Cloneable {
    @Getter
    private final List<Byte> flippingOperations = new ArrayList<>();
```

```
    public FlippingOrder add(Byte integer) {
        this.flippingOperations.add(integer);
        return this;
    }

    public FlippingOrder append(FlippingOrder flippingOrder) {
        FlippingOrder copy = new FlippingOrder();
        copy.getFlippingOperations().addAll(this.getFlippingOperations());
        copy.getFlippingOperations().addAll(flippingOrder.getFlippingOperations());
        return copy;
    }
}


public class PancakeFlipper {

    /**
     * Flips the pancake stack between index - 1 and index, then eats the last pancake.
     * Also performs the same operations on the normalized pancake stack.
     *
     * @param pancakeStack The pancake stack to flip and eat.
     * @param index The index to flip the pancakes.
     */
    public void flip(PancakeStack pancakeStack, int index) {
        Byte[] normalizedPancakes = pancakeStack.getNormalizedPancakes().getPancakes();
        if (index < 0 || index > normalizedPancakes.length) {
            throw new IllegalArgumentException("Flip operation out of bounds. Trying to flip
                between " + (index - 1) +
                   " and " + index + " but current stack only has " + normalizedPancakes.length
                       + " pancakes!");
        }

        Byte[] normalizedFlipped = flip(index, normalizedPancakes);
        byte aboutToBeEaten = normalizedFlipped[normalizedFlipped.length - 1];
        pancakeStack.setNormalizedPancakes(new
            PancakeStackData(normalize(eat(normalizedFlipped), aboutToBeEaten)));

        Byte[] pancakes = pancakeStack.getPancakes().getPancakes();
        Byte[] flipped = flip(index, pancakes);
        pancakeStack.setPancakes(new PancakeStackData(eat(flipped)));
    }

    /**
     * Flips a given pancake stack between index - 1 and index.
     *
     * @param index The index to flip the pancakes.
     * @param pancakes The pancake stack to flip.
     * @return The flipped pancake stack.
     */
    private Byte[] flip(int index, Byte[] pancakes) {
        if (index == 0 || index == pancakes.length) {
            return reverse(pancakes);
        }

        Byte[] flipped = Arrays.copyOf(pancakes, pancakes.length);
        for (int i = index, j = pancakes.length - 1; i < j; i++, j--) {
            byte temp = flipped[i];
            flipped[i] = flipped[j];
            flipped[j] = temp;
        }

        return flipped;
```

```
    }

    /**
     * Removes the last pancake from the stack.
     *
     * @param current The pancake stack to remove the last pancake from.
     * @return The pancake stack with the last pancake removed.
     */
    private Byte[] eat(Byte[] current) {
        Byte[] eaten = new Byte[current.length - 1];
        System.arraycopy(current, 0, eaten, 0, current.length - 1);
        return eaten;
    }

    /**
     * Adjusts the pancake stack after eating the last pancake.
     *
     * @param current The pancake stack to normalize.
     * @param removed The value of the removed pancake.
     * @return The normalized pancake stack.
     */
    private Byte[] normalize(Byte[] current, byte removed) {
        for (int i = 0; i < current.length; i++) {
            byte value = current[i];
            if (value > removed) {
                current[i] = (byte) (value - 1);
            }
        }
        return current;
    }

    /**
     * Reverses the order of the given pancake stack.
     *
     * @param current The pancake stack to reverse.
     * @return The reversed pancake stack.
     */
    private Byte[] reverse(Byte[] current) {
        Byte[] reversed = Arrays.copyOf(current, current.length);
        for (int i = 0, j = current.length - 1; i < j; i++, j--) {
            byte temp = reversed[i];
            reversed[i] = reversed[j];
            reversed[j] = temp;
        }
        return reversed;
    }

}

@RequiredArgsConstructor
public class PancakeFlippingOrderApplier {

    private final PancakeFlipper flipper;

    public PancakeStack apply(PancakeStack pancakeStack, FlippingOrder flippingOrder) {
        if (flippingOrder.getFlippingOperations().size() == 0) {
            return pancakeStack;
        }
        for (Byte flippingOperation : flippingOrder.getFlippingOperations()) {
            flipper.flip(pancakeStack, flippingOperation);
        }
        return pancakeStack;
    }
```

```
    public PancakeStack applyAndPrintDebug(PancakeStack pancakeStack, FlippingOrder
        flippingOrder) {
        if (flippingOrder.getFlippingOperations().size() == 0) {
            System.out.println(Arrays.toString(pancakeStack.getPancakes().getPancakes()) + " ist
                bereits gelöst.");
            return pancakeStack;
        }
        for (final Byte flippingOperation : flippingOrder.getFlippingOperations()) {
            final PancakeStack copy = pancakeStack.clone();
            flipper.flip(pancakeStack, flippingOperation);
            System.out.println("Wende-und-Essoperation an Index " + flippingOperation + ": " +
                    Arrays.toString(copy.getPancakes().getPancakes()) + " -> " +
                    Arrays.toString(pancakeStack.getPancakes().getPancakes()));
        }
        return pancakeStack;
    }

}

@RequiredArgsConstructor
public class PWUENumberCalculator {

    private final PancakeStackSorter pancakeStackSorter;
    private final PancakeFlipper pancakeFlipper;

    private final Map<Integer, AtomicInteger> heightToPWUE = new ConcurrentHashMap<>(); //
        height to pwue nr
    private final Map<PancakeStackData, Integer> flippingOrderLengthMap = new
        ConcurrentHashMap<>(); // pancake stack to flipping order length
    private final static Byte[][] cleanseOperations = new Byte[][]{
            new Byte[]{}, // 0
            new Byte[]{}, // 1
            new Byte[]{}, // 2
            new Byte[]{3, 2}, // 3
            new Byte[]{3, 2, 2}, // 4
            new Byte[]{4, 3, 3, 2}, // 5
            new Byte[]{4, 3, 3, 2, 2}, // 6
            new Byte[]{5, 4, 4, 3, 3, 2}, // 7
            new Byte[]{5, 4, 4, 3, 3, 2, 2},// 8
            new Byte[]{6, 5, 5, 4, 4, 3, 3, 2}, // 9
            new Byte[]{6, 5, 5, 4, 4, 3, 3, 2, 2}, // 10
            new Byte[]{7, 6, 6, 5, 5, 4, 4, 3, 3, 2}, // 11
            new Byte[]{7, 6, 6, 5, 5, 4, 4, 3, 3, 2, 2}, // 12
            new Byte[]{8, 7, 7, 6, 6, 5, 5, 4, 4, 3, 3, 2} // 13
    };


    private int startedTasks = 0;
    private final AtomicInteger finishedTasks = new AtomicInteger(0);
    private final static int BATCH_SIZE = 10000;


    // bottom up pwue number generation
    public PancakeStackSortingResult calcPWUE(int height) {
        int pwue = this.calcPWUEForHeight(height);
        return this.findAndSolveStackWithPWUECountAmount(pwue, height);
    }

    public Set<PancakeStackSortingResult> calcPWUE(int height, int amount) {
        int pwue = this.calcPWUEForHeight(height);
        Set<PancakeStackSortingResult> set = new HashSet<>();
        for (int i = 0; i < amount; i++) {
```

```
            set.add(this.findAndSolveStackWithPWUECountAmount(pwue, height));
        }
        return set;
    }

    private PancakeStackSortingResult findAndSolveStackWithPWUECountAmount(int pwue, int
         height) {
        List<Byte> pancakes = this.generateInitialPancakes(height);
        PancakeStack pancakeStack;
        PancakeStackSortingResult result;
        do {
            Collections.shuffle(pancakes);
            pancakeStack = new PancakeStack(pancakes.toArray(Byte[]::new));
            result = this.pancakeStackSorter.sort(pancakeStack);
        } while (result.getFlippingOrder().getFlippingOperations().size() != pwue);
        return result;
    }

    public int calcPWUEForHeight(int height) {
        this.flippingOrderLengthMap.put(new PancakeStackData(new Byte[]{1}), 0);
        this.heightToPWUE.put(1, new AtomicInteger(0));

        this.flippingOrderLengthMap.put(new PancakeStackData(new Byte[]{1, 2}), 1);
        this.heightToPWUE.put(2, new AtomicInteger(1));
        for (int i = 3; i <= height; i++) {
            // System.out.println("NOW GENERATING MAP FOR " + i);
            this.permute(i, i != height);
            while (startedTasks - finishedTasks.get() != 0) {
                try {
                    System.out.println("waiting for " + (startedTasks - finishedTasks.get()) + "
                        tasks");
                    Thread.sleep(5);
                } catch (InterruptedException e) {
                    throw new RuntimeException(e);
                }
            }
            if (i != height) {
                this.cleanse(i, height);
            }
        }
        return this.heightToPWUE.get(height).get();
    }

    private void cleanse(int current, int max) {

        this.flippingOrderLengthMap.entrySet().removeIf(entry -> {
            int entryHeight = entry.getKey().getPancakes().length;
            int entryValue = entry.getValue();

            if (entryHeight <= current - 1) {
                return true;
            }

            int pwue = this.heightToPWUE.get(entryHeight).get();
            return entryValue <= pwue - cleanseOperations[max][current - 1];
        });

    }

    private void permute(int height, boolean writeToMap) {
        List<Byte> pancakes = this.generateInitialPancakes(height);
        permute(pancakes, 0, height, writeToMap);
    }
```

```
private List<Byte> generateInitialPancakes(int height) {
    List<Byte> bytes = new ArrayList<>();
    for (int i = 1; i <= height; i++) {
        byte b = (byte) i;
        bytes.add(b);
    }
    return bytes;
}

private void permute(List<Byte> pancakes, int height, int initialHeight, boolean
     writeToMap) {
    for (int i = height; i < pancakes.size(); i++) {
        java.util.Collections.swap(pancakes, i, height);
        permute(pancakes, height + 1, initialHeight, writeToMap);
        java.util.Collections.swap(pancakes, height, i);
    }
    if (height == pancakes.size() - 1) {

        PancakeStack pancakeStack = new PancakeStack(pancakes.toArray(new Byte[0]));
        while (startedTasks - finishedTasks.get() > BATCH_SIZE) {
            try {
                Thread.sleep(20);
            } catch (InterruptedException e) {
                throw new RuntimeException(e);
            }
            // TODO: 17.04.2023 this is very hacky
        }
        startedTasks++;
        this.sortAndApplyResults(pancakeStack, writeToMap, height, initialHeight)
                .subscribeOn(Schedulers.parallel())
                .doOnNext(integer -> finishedTasks.incrementAndGet())
                .subscribe();
    }

}


private Mono<Integer> sortAndApplyResults(PancakeStack pancakeStack, boolean writeToMap,
     int height, int initialHeight) {
    Mono<Integer> mono = !writeToMap && pancakeStack.getPancakes().getPancakes()[0] ==
         height + 1 ?
            Mono.just(-1) : Mono.just(this.sort(pancakeStack, writeToMap));
    return mono.doOnNext(requiredFlippingOperationsCount -> {
        if (requiredFlippingOperationsCount == -1) {
            return;
        }
        AtomicInteger currentWorstCase = heightToPWUE.getOrDefault(initialHeight, new
             AtomicInteger(Integer.MIN_VALUE));
        int currentWorstCaseValue = currentWorstCase.get();
        if (requiredFlippingOperationsCount <= currentWorstCaseValue) {
            return;
        }
        currentWorstCase.compareAndSet(currentWorstCaseValue,
             requiredFlippingOperationsCount);
        this.heightToPWUE.put(initialHeight, currentWorstCase);
    });
}

public int sort(PancakeStack pancakeStack, boolean writeToMap) {
    return this.optimalFlippingOrder(pancakeStack, writeToMap);
}
```

```
    private int optimalFlippingOrder(PancakeStack pancakeStack, boolean writeToMap) {
        if (pancakeStack.isSolved()) {
            return -1;
        }
        Set<Integer> possibleNextOperations = new HashSet<>();
        for (byte i = 0; i < pancakeStack.getNormalizedPancakes().getPancakes().length; i++) {
            PancakeStack clonedPancakeStack = pancakeStack.clone();
            this.pancakeFlipper.flip(clonedPancakeStack, i);

            if (clonedPancakeStack.isSolved()) {
                this.flippingOrderLengthMap.put(pancakeStack.getNormalizedPancakes(), 1);
                return 1;
            }

            if
                 (!this.flippingOrderLengthMap.containsKey(clonedPancakeStack.getNormalizedPancakes()))
                 {
                return -1;
            }
            int operations =
                 this.flippingOrderLengthMap.get(clonedPancakeStack.getNormalizedPancakes());
            possibleNextOperations.add(operations);
        }

        if (possibleNextOperations.isEmpty()) return -1;
        int best = this.findBest(possibleNextOperations) + 1;

        if (writeToMap) {
            this.flippingOrderLengthMap.put(pancakeStack.getNormalizedPancakes(), best);
        }
        return best;
    }

    private int findBest(Set<Integer> flippingOrders) {
        int length = Integer.MAX_VALUE;
        for (Integer integer : flippingOrders) {
            if (integer < length) {
                length = integer;
            }
        }
        return length;
    }
}
```