41. Bundeswettbewerb Informatik

Aufgabe 3: Pancake Sort

Florian Bange
Teilnahme-ID: 64810

17. April 2023

Inhaltsverzeichnis

0. Einleitung

Dieses Dokument beinhaltet meine Dokumentation der dritten Aufgabe der zweiten Runde des 41. Bundeswettbewerbs Informatik[1] aus dem Jahr 2023.

Bereits bekannt ist das sogenannte „Pancake Problem“, welches im Jahre 1975 erstmals von Jacob E. Goodman[2] unter dem Pseudonym Harry Dweighter erwaehnt wurde [1, 2]. Dabei geht es darum, einen Stapel an n unterschiedlich groszen Pfannkuchen so zu sortieren, dass der kleinste Pfannkuchen ganz oben, der zweitegroeszte darunter, usw. und der groeszte Pfankuchen ganz unten liegt. Um den Stapel zu sortieren, duerfen allerdings ausschlieszlich einer oder mehrere Pfannkuchen von der Spitze des Stapels umgedreht werden (Prefix Reversal; Praefixumdrehung).
Wie in [3] beschrieben wird, folgen aus dem Original [1, 2] zwei Probleme:

1. **Das MIN-SBPR Problem** (Sorting By Prefix Reversals):
   Fuer einen gegebenen Stapel S die kuerzeste Sequenz an Praefixumdrehung finden, sodass S in einen geordneten Stapel ueberfuehrt wird.

2. **Berechnung von** $f(n)$ (fuer $n \in \mathbb{N}$):
   Wie im Original [1, 2]: Fuer ein $n \in \mathbb{N}$ die maximale Anzahl an Praefixumdrehungen von Pfannkuchen, die fuer irgendeinen Pfannkuchenstapel der Groesze n mindestens benoetigt werden, finden.

In [3] wird gezeigt, dass das MIN-SBPR Problem NP-schwer ist. Eine Variation des Pancake Problems besteht in Form des „Burnt Pancake Problems“ und wurde zuerst von Bill Gates[3] und Christos H. Papadimitriou vorgestellt [7]. Dabei haben die Pfannkuchen je eine verbrannte Seite und eine nicht verbrannte Seite und das Ziel ist es, dass die verbrannte je unten liegt (also nicht sichbar ist).
Sowohl fuer die „burnt“ als auch fuer die „unburnt“ Variante des zweiten Problems wurden Abschaetzungen (Unter- und Obergrenzen) fuer $f(n)$ (bzw. $g(n)$ fuer „Burnt Pancakes“) gefunden [4–6].

Die gegebene Aufgabe besteht aus zwei Teilproblemen, welche analog zu den zuvor beschriebenen Problemen funktionieren. Dabei wird anstatt der Praefixumdrehung die sogenannte „Wende-und-Ess-Operation“ (kurz: WUE-Operation) genutzt, um den Stapel zu sortieren: Es wird die Praefixumdrehung durchgefuehrt und anschlieszend wird der oberste Pfannkuchen entfernt (gegessen).
Die erste Teilaufgabe (a) ist analog zum MIN-SBPR Problem: Fuer einen gegebenen Stapel S wird eine moeglichst kurze Sequenz an WUE-Operationen gesucht, die S in einen sortierten Stapel ueberfuehrt. Im Folgenden wird dieses Problem als „MIN-WUE-Operations-Problem“ (kurz: MWO-Problem) bezeichnet.
Die zweite Teilaufgabe (b) wiederum ist analog zum zweiten Problem definiert: Es gilt $P(n)$ zu berechnen und ein dazugehoeriges Beispiel zu liefern. Dabei ist $P(n)$ analog zu $f(n)$ (s.o.) definiert, wobei natuerlich mit WUE-Operationen gearbeitet wird. Dabei wird $P(n)$ als „Pfannkuchen-Wende-Und-Ess-Zahl“ (PWUE-Zahl) bezeichnet und dessen Berechnung wird im folgenden als „Pfannkuchen-Wende-Und-Ess-Zahl-Problem“ (kurz: PWZ-Problem) bezeichnet. Zusaetzlich soll eine Pfannkuchenstapel der Groesze n angegeben werden, fuer welchen mindestens $P(n)$ WUE-Operationen benoetigt werden, um in in einen sortierten Stapel zu ueberfuehren.

Diese Dokumentation ist so aufgebaut, dass zunaechst einige mathematischen Modellierungen der fuer die Probleme noetigen Komponenten vorgenommen werden (Kapitel 1), sodass anschlieszend in Kapitel 2 die zum Loesen der Probleme notwendigen formalen Definitionen der WUE-Operation und der eigentliche Probleme durchgefuehrt werden koennen. Insbesondere werden dazu in Kapitel 1 Pfannkuchenstapel durch Permutationen dargestellt und die WUE-Operation wird getreu der Aufgabenstellung definiert. Daraufhin werden die beiden Probleme Kapitel 3 mit konkreten Loesungsvorschlaegen geloest und jeweils mit (halb-)formale Korektheitsbegruendungen erklaert.
Dann wird in Kapitel 4 eine vereinfache Definition der WUE-Operation angegeben, die eine Abbildung zwischen Permutationsgruppen ist. Mit Hilfe dieser Definition koennen danach in Kapitel 5 verbesserte Loesungsvorschlaege gemacht werden. Auch diese werden (halb-)formal bezueglich ihrer Korrektheit begruendet. Im darauffolgenden Kapitel 6 werden die Loesungsalgorithmen mit Hilfe der Dynamischen Programmierung deutlich verbessert.
Zu Schluss wird in Kapitel 7 erlaeutert, wie die zuvor beschriebenen Loesungsvorschlaege in ein Java 8 Programm umgesetzt wurden und fuer beide Probleme werden in Kapitel 8 zahlreiche Beispiele, bestehend aus denen der BwInf.-Website und eigenen, dargelegt, die zeigen, dass die Programme tatsaechlich funktionieren. Daraufhin werden die wichtigsten Bestandteile ebendieser Programme in Kapitel 9 gezeigt und erklaert.

[1] s. 41. BwInf. Runde 2: `https://bwinf.de/bundeswettbewerb/41/2/`
[2] Jacob Eli Goodman (1933 – 2021): `https://en.wikipedia.org/wiki/Jacob_E._Goodman`
[3] Tatsaechlich der Gruender von Microsoft: `https://de.wikipedia.org/wiki/Bill_Gates`

1. Grundlegende Definitionen

Um die beiden Probleme theoretisch betrachten zu koennen, werden sie nun mathematisch modelliert. Dazu werden zunaechst die Pfannkuchenstapel als Permutationen[4] der Zahlen $1,\ 2,\ \ldots\ ,\ n$ mit $n \in \mathbb{N}$ definiert. Da dabei haeufig der Begriff der Abbildung benutzt werden wird, wird dieser zuerst Definiert.

Definition 1: Korrespondenz

Es seien M und N zwei Mengen. Dann nennt man

$$M \times N \coloneqq \{(x,\ y)\ :\ x \in M \wedge y \in N\}$$

das kartesische Produkt von M und N, dabei ist

$$(x,\ y) \in M \times N$$

ein Tupel (d.h., geordnet!), wobei x das erste und y das zweite Element des Tupels ist.

Nun nennt man jede Teilmenge

$$K \subseteq M \times N$$

Korrespondenz zwischen M und N.

Definition 2: Abbildung

Es seien M und N zwei nichtleere Mengen und $f \subseteq M \times N$ Korrespondenz zwischen M und N.
Dann nennt man f eine Abbildung von M nach N, wenn

$$(\forall x \in M(\exists! y \in N)[\ (x,\ y) \in f\ ]$$

gilt. D.h., fuer jedes x der Menge M existiert genau ein y der Menge N, sodass $(x,\ y) \in f$.

Bemerkung 3: Notation

Es sei $f \subseteq M \times N$ eine Abbildung von M nach N.

1. Im Folgenden wird fuer eine Abbildung auch $f : M \longrightarrow N$ geschrieben.
2. Auszerdem sei
$$(f(x) = y) :\overset{Def.}{\iff} (x,\ y) \in f$$
wobei man y das Bild von x unter f nennt.
3. Im Weiteren wird fuer M die Bezeichnung Definitionsmenge und fuer N die Bezeichnung Bildmenge genutzt.
4. Gibt es eine Abbildungsvorschrift $f(x)$, so wird dafuer geschrieben $f : M \longrightarrow N$ mit $x \longmapsto f(x)$ oder kuerzer $f : M \ni x \longmapsto f(x) \in N$.

Definition 4: Eigenschaften von Abbildungen

Es sei $f : M \longrightarrow N$ eine Abbildung. Dann nennt man f

1. injektiv, wenn
$$(\forall x_1,\ x_2 \in M)[\ f(x_1) = f(x_2)\ \implies\ x_1 = x_2\ ]$$
D.h., jedes $y \in N$ wird maximal von einem $x \in N$ „getroffen“.
2. surjektiv, wenn
$$(\forall y \in N)(\exists x \in N)[\ f(x) = y\ ]$$
D.h., jedes $y \in \mathbb{N}$ wird von mindestens einem $x \in \mathbb{N}$ „getroffen“.
3. bijektiv, wenn
$$(\forall y \in N)(\exists! x \in N)[\ f(x) = y\ ]$$
D.h., jedes $y \in \mathbb{N}$ wird von genau einem $x \in \mathbb{N}$ „getroffen“. Bzw. f ist injektiv und surjektiv.

[4] Siehe auch `https://en.wikipedia.org/wiki/Permutation`

Definition 5: Permutationen der Zahlen von 1 bis n

Es sei $n \in \mathbb{N}$. Nun wird S_n definiert als die Menge aller bijektiven Abbildungen

$$\pi : \{1,\ 2,\ \dots\ ,\ n\} \longrightarrow \{1,\ 2,\ \dots\ ,\ n\}$$

Also

$$S_n \coloneqq \{\pi \mid \{1,\ 2,\ \dots\ ,\ n\} \longrightarrow \{1,\ 2,\ \dots\ ,\ n\} \text{ und } \pi \text{ bijektiv}\}$$

Die Elemente aus S_n werden Permutationen der Zahlen $1,\ 2,\ \dots\ ,\ n$ genannt. Im Folgenden wird dafuer vereinfacht die Bezeichnung Permutation benutzt.

Bemerkung 6: Permutationen der Zahlen von 1 bis n

Diese Definition ergibt wie folgt Sinn:
Da eine Permutation $\pi \in S_n$ ist bijektive Abbildung von $\{1,\ 2,\ \dots\ ,\ n\}$ auf sich selber ist, wird jedem $x \in \{1,\ 2,\ \dots\ ,\ n\}$ genau ein $y = \pi(x) \in \{1,\ 2,\ \dots\ ,\ n\}$ zugeordnet. Wuerde man nun nun die Bilder $\pi(1),\ \pi(2), \dots\ ,\ \pi(n)$ in dieser Reihenfolge notieren, so erhaelt man die jeweilige Permutation π.

Bemerkung 7: Schreibweisen

Eine Permutationen $\pi \in S_n$ fuer $n \in \mathbb{N}$ kann man unter anderem als n-Tupel darstellen:

$$\pi = \Big(\pi(1),\ \pi(2),\ \dots\ ,\ \pi(n)\Big) \in S_n$$

wobei $\{\ \pi(1),\ \pi(2),\ \dots\ ,\ \pi(n)\ \} = N$ ist.
Eine allgemeinere Darstellung von Permutationen besteht in Cauchys[5] zweizeiliger Notation, die er 1815 erstmals benutzte [8, 9]:
Eine Permutation $\pi \in S_n$ wird mit Hilfe einer Matrix aus zwei Zeilen und n Spalten beschrieben. Dabei stehen in der ersten Zeile alle Zahlen von 1 bis n genau einmal in beliebiger Reihenfolge und unter jeder Zahl $s \in \{1,\ 2,\ \dots\ ,\ n\}$ der ersten Zeile in der zweiten Zeile das jeweilige Bild $\pi(s)$:

$$\pi = \begin{pmatrix} s_1 & s_2 & \dots & s_n \\ \pi(s_1) & \pi(s_2) & \dots & \pi(s_n) \end{pmatrix} \in S_n$$

Dabei ist $\{s_1,\ s_2,\ \dots\ ,\ s_n\} = \{1,\ 2,\ \dots\ ,\ n\} = \{\pi(s_1),\ \pi(s_2),\ \dots\ ,\ \pi(s_n)\}$. D.h. Sowohl in der ersten als auch in der zweiten Zeile kommen die Zahlen von 1 bis n genau einmal vor.
Auszerdem gilt:

$$\pi = \{\ (s_i,\ \pi(s_i))\ : i \in N\ \}$$

Beispiel 8: Permutationen

Hier einige Beispiele:

1. $$\pi = \begin{pmatrix} 1 & 2 & \dots & n \\ \pi(1) & \pi(2) & \dots & \pi(n) \end{pmatrix} \in S_n \tag{1}$$

   Hier wird eine moegliche Beschreibung eines beliebigen $\pi \in S_n$ gezeigt. Dabei wird in der j-ten Spalte $(1,\ 2,\ 3,\ \dots\ ,\ n)$ dargestellt, dass $j \in N$ auf $\pi(j)$ abbildet.

2. $$\sigma = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 3 & 2 & 4 & 1 \end{pmatrix} \in S_4$$

   Hier wird ein konkretes Beispiel $\sigma \in S_4$ gezeigt, wobei $\sigma = \{\ (1,\ 3)\,,\ (2,\ 2)\,,\ (3,\ 4)\,,\ (4,\ 1)\ \}$ ist.

3. $$\sigma = \begin{pmatrix} 2 & 4 & 3 & 1 \\ 2 & 1 & 4 & 2 \end{pmatrix} \in S_4$$

   Hier wird fuer das $\sigma \in S_4$ des vorherigen Beispiels eine weitere Darstellung gezeigt.

[5] Augustin-Louis Cauchy (1789 – 1857): `https://en.wikipedia.org/wiki/Augustin-Louis_Cauchy`

Bemerkung 9:

S_n besteht aus genau $n!$ (n Fakultaet) Elementen. D.h., es gibt $n!$ Permutationen (also Anordnungen) von $1,\ 2,\ \dots\ n$.

Bemerkung 10: Permutationsgruppe

Die soeben beschriebene Menge S_n fuer $n \in \mathbb{N}$ aus Permutationen von $N \coloneqq \{1,\ 2,\ \dots\ ,\ n\}$ bildet zusammen mit der Verknuepfung von Abbildungen (Komposition) $\circ$ eine Gruppe. Die sogenannte Permutationsgruppe[6]. Das heiszt fuer $n \in \mathbb{N}$ ist $(S_n,\ \circ)$ eine Gruppe und es gilt:

1. Die Komposition von zwei Permutationen $\pi,\ \sigma \in S_n$ mit $n \in \mathbb{N}$ ist ebenfalls in S_n.

$$(\forall \pi,\ \sigma \in S_n)[\ \pi \circ \sigma \in S_n\ ]$$

2. Die Komposition von Permutationen ist assoziativ:

$$(\forall \pi,\ \sigma,\ \tau \in S_n)[\ (\pi \circ \sigma) \circ \tau = \pi \circ (\sigma \circ \tau) \in S_n\ ]$$

3. Es existiert ein Neutralelement $e \in S_n$ mit $(\forall \pi \in S_n)[\ e \circ \pi = \pi\ ]$:

$$e = \begin{pmatrix} 1 & 2 & \dots & n \\ 1 & 2 & \dots & n \end{pmatrix} =: \mathrm{id} \in S_n$$

Bemerkung: *id* ist sowohl links- als auch rechtsneutral.

4. Fuer jedes $\pi \in S_n$ existiert ein inverses Element π^{-1} mit $\pi^{-1} \circ \pi = \mathrm{id}$. Fuer

$$\pi = \begin{pmatrix} 1 & 2 & \dots & n \\ \pi(1) & \pi(2) & \dots & \pi(n) \end{pmatrix}$$

ist

$$\pi^{-1} = \begin{pmatrix} \pi(1) & \pi(2) & \dots & \pi(n) \\ 1 & 2 & \dots & n \end{pmatrix}$$

Bemerkung: Die inversen Elemente sind sowohl links- als auch rechtsinvers.

Bemerkung 11: Cauchys Schreibweise

Im Folgenden wird Cauchys zweizeilige Notation so benutzt, dass in der ersten Zeilen die Zahlen von 1 bis n in ueblicher Reihenfolge stehen und in der zweiten Zeile die dazugehoerigen Werte stehen (vgl. 1).

Auszerdem wird diese Schreibweise auch fuer beliebige andere Abbildungen von $\{1,\ 2,\ \dots\ ,\ n\}$ (fuer $n \in \mathbb{N}$) nach A, wobei A eine beliebige Menge ist, benutzt werden.

Definition 12: Pfannkuchenstabel I

Ein Pfannkuchenstabel S kann modelliert werden als Permutation $\pi \in S_n$ der Größe $n \in \mathbb{N}$. Denn ein Pfannkuchenstabel der Groeße $n \in \mathbb{N}$ ist nichts anderes als eine Permutation der Zahlen von 1 bis n.
Im Foglenden werden Pfannkuchen bzw. Permutationen via der zuvor benutzten Matrix dargestellt:

$$\pi = \begin{pmatrix} 1 & 2 & \dots & n \\ \pi(1) & \pi(2) & \dots & \pi(n) \end{pmatrix}$$

Dabei bedeutet diese Darstellung, dass der j-te Pfannkuchen (von oben nach unten) in der j-ten Spalte dargestellt ist und die Groesze $\pi(j)$ fuer $j \in \{1,\ 2,\ \dots\ ,\ n\}$ hat.

Beispiel 13: Pfannkuchenstabel

Es sei $n \coloneqq 5$. Dann laesst sich der sortierte Pfannkuchenstapel der Groesze n wie folgt darstellen:

$$\mathrm{id} \coloneqq \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 1 & 2 & 3 & 4 & 5 \end{pmatrix}$$

Weiter laesst sich der Pfannkuchenstapel der Aufgabenstellung bestehend aus den Pfannkuchen der Groeszen (v. o. n. u.) 3, 2, 4, 5 und 1 so darstellen:

$$\mathrm{S} \coloneqq \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 3 & 2 & 4 & 5 & 1 \end{pmatrix}$$

[6] Siehe hier: `https://de.wikipedia.org/wiki/Permutationsgruppe`

2. Modellierung der originalen WUE-Operation und Definition der Probleme

Nachdem nun eine formale Modellierung fuer Pfannkuchenstapel abgeschlossen wurde, wird nun zunaechst eine weitere Definition fuer Pfannkuchenstapel nach null oder mehr Anwendung von WUE-Operationen gegeben. Anschlieszend wird die WUE-Operation als Abbildung formal definiert, sodass danach $A(S)$, $P(n)$ fuer $n \in \mathbb{N}$, sowie die beiden Probleme formal definiert werden.

Definition 14: Pfannkuchenstabel II

Wurde eine Permutation (Pfannkuchenstabel) $\pi \in S_n$ mit $n \in \mathbb{N}$ durch $k \in \mathbb{N}_0$ mit $0 \leq k \leq n-1$ WUE-Operationen[7] (die im folgenden formal definiert werden werden wird) veraendert, so ist es moeglich, dass es sich beim entstandenen Pfannkuchenstapel nicht mehr um eine Permutation aus S_{n-k} handelt.
Deswegen wird nun die Menge

$$W_n^{(k)} \coloneqq \left\{\sigma \ : \{1,\ 2,\ \dots\ ,\ n-k\} \longrightarrow \{1,\ 2,\ \dots\ ,\ n\} \mid \sigma \text{ injektiv}\right\}$$

definiert.
Diese enthaelt alle moeglichen Pfannkuchenstapel, wobei ein Pfannkuchenstabel der Groesze $n \in \mathbb{N}$ durch $k \in \mathbb{N}_0$ $(0 \leq k \leq n-1)$ WUE-Operation veraendert wurde, sodass dieser nun die Groesze $n-k$ hat.

Bemerkung 15: Pfannkuchenstabel

Fuer $n \in \mathbb{N}$ und $k = 0$ ist $W_k^{(n)}$ logischer Weise dasselbe wie S_n. D.h[8].,

$$\begin{aligned} W_0^{(n)} &\coloneqq \left\{\sigma \ : \{1,\ 2,\ \dots\ ,\ n-0\} \longrightarrow \{1,\ 2,\ \dots\ ,\ n\} \mid \sigma \text{ injektiv}\right\} \\ &= \left\{\sigma \ : \{1,\ 2,\ \dots\ ,\ n\} \longrightarrow \{1,\ 2,\ \dots\ ,\ n\} \mid \sigma \text{ bijektiv}\right\} \\ &=: S_n \end{aligned}$$

Weiter waere fuer $k = n$

$$\begin{aligned} W_n^{(k)} &= W_n^{(n)} \\ &= \left\{\sigma \ : \{1,\ 2,\ \dots\ ,\ n-n\} \longrightarrow \{1,\ 2,\ \dots\ ,\ n\} \mid \sigma \text{ injektiv}\right\} \\ &= \left\{\sigma \ : \emptyset \longrightarrow \{1,\ 2,\ \dots\ ,\ n\} \mid \sigma \text{ injektiv}\right\} \\ &= \{\emptyset\} \end{aligned}$$

also die Menge, die eine Abbildung ohne Element enthaelt. Allerdings ist hier $W_k^{(n)}$ fuer $k \geq n$ nicht definiert.

Bemerkung 16: Pfannkuchenstabel

Im Folgenden wird ein Pfannkuchenstabel der Form $\pi \in S_n$ fuer $n \in \mathbb{N}$ als urspruenglicher Pfannkuchenstapel bezeichnet, da dieser noch nicht durch WUE-Operationen veraendert wurde und daher eine echte Permutationen der Zahlen von 1 bis n ist.

$\sigma \in W_n^{(k)}$ (mit $k \in \mathbb{N}$ und $k \geq 1$) wird hingegen als veraenderter Pfannkuchenstabel bezeichnet, da dieser keine Permutation der Zahlen von 1 bis n ist und bereits durch WUE-Operationen veraendert wurde.

Beispiel 17: Pfannkuchenstabel

Mit Hilfe von Cauchys zweizeiliger Notation werden nun Pfannkuchenstapel nach der vorheringen Definition von $W_n^{(k)}$ mit $n,\ k \in \mathbb{N}$ mit $0 \leq k \leq n-1$ dargestellt:
Fuer $n = 6$ und $k = 2$ ist

$$\begin{pmatrix} 1 & 2 & 3 & 4 \\ 1 & 6 & 5 & 3 \end{pmatrix} \text{ oder auch } \begin{pmatrix} 1 & 2 & 3 & 4 \\ 2 & 5 & 3 & 1 \end{pmatrix}, \text{ sowie } \begin{pmatrix} 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \end{pmatrix}$$

moeglich. Hingegen sind diese Pfannkuchenstapel nicht moeglich:

$$\begin{pmatrix} 1 & 2 & 3 & 4 \\ 7 & 5 & 3 & 1 \end{pmatrix}, \begin{pmatrix} 1 & 2 & 6 & 5 \\ 1 & 2 & 3 & 4 \end{pmatrix}$$

[7] $0 \leq k \leq n-1$, damit auch keine moeglich ist, aber hoechsten $k-1$ getaetigt werden koennen, damit immer mindestens ein Pfannkuchen ueberbleibt.

[8] Bemerkung: Fuer eine Abbildung $f : M \longrightarrow N$ mit $|M| = |N|$ ist (f bijektiv) logisch aequivalent zu (f injektiv $\vee$ f surjektiv)

Satz 18: Maechtigkeit der Menge $W_n^{(k)}$

Es sei $n \in \mathbb{N}$ und $k \in \mathbb{N}_0$ mit $0 \leq k \leq n-1$. Die Maechtigkeit (Groesze) der Menge $W_n^{(k)}$

$$\left|W_n^{(k)}\right|$$

entspricht genau

$$\frac{n!}{k!}.$$

Beweis:
Die Element dieser Menge entsprechen genau allen Permutationen der Zahlen von 1 bis n der Groesze $n-k$, wobei die Reihenfolge beachtet wird, aber keine doppelten Elemente vorkommen duerfen. Somit gibt es fuer das Erste Element ($\pi(1)$) n Moeglichkeiten, fuer das zweite Element nur noch $n-1$ Moeglichkeiten, usw. bis es fuer das $(n-k)$-te Element genau $n-(n-k)+1 = k+1$ Moeglichkeiten. Multipliziert man diese Werte, so erhaelt man die Anzahl an Moeglichen Permutationen der Groesze $n-k$ mit n verschiedenen Elementen:

$$\begin{aligned}&(n)\cdot(n-1)\cdot\dots\cdot(k+2)\cdot(k+1)\\ =&\frac{(n)\cdot(n-1)\cdot\dots\cdot(k+2)\cdot(k+1)\cdot(k)\cdot\dots\cdot 2\cdot 1}{(k)\cdot(k-1)\cdot\dots\cdot(2)\cdot(1)}\\ =&\frac{n!}{k!}\end{aligned}$$

Definition 19: Wende-und-Ess-Operation I

Es sei $n \in \mathbb{N}$ und $k \in \mathbb{N}_0$ mit $n \geq 2$ und $0 \leq k \leq n-2$, sodass $0 \leq k+1 \leq n-1$, damit sowohl $W_n^{(k)}$ als auch $W_n^{(k+1)}$ wohl definiert sind. Weiter sei $\pi \in W_n^{(k)}$ ein Pfannkuchenstapel der Groesze $n-k \geq 2$, der bereits durch k WUE-Operationen veraendert wurde.
Nun wird die Wende-und-Ess-Operation an der Stelle $i \in \mathbb{N}$ mit $1 \leq i \leq n-k$ durch die Abbidlung

$$\psi_i^{(n,\ k)} : W_n^{(k)} \longrightarrow W_n^{(k+1)}$$

dargestellt, welche einem Pfannkuchenstapel der Groesze $n-k$ auf einen der Grosze $n-k-1$ abbildet.
Dabei sind $n \geq 2$ und $0 \leq k \leq n-2$, damit keine Stapel der Groesze 0 entstehen koennen und i ist der Index des Pfannkuchens unter den der Pfannenwender geschoben wird.
Nun wird $\psi_i^{(n,\ k)}$ definiert:

$$\begin{aligned}&\psi_i^{(n,\ k)} : W_n^{(k)} \ni \pi\\ &\qquad\longmapsto\\ &\left(\pi' : \{1,\ \dots\ ,\ n-k-1\} \ni x \longmapsto \begin{Bmatrix}\pi(x+1), & \text{falls } x \geq i\\ \pi(i-x), & \text{falls } x < i\end{Bmatrix} \in \{1,\ 2,\ \dots\ ,\ n\}\right) \in W_n^{(k+1)}\end{aligned}$$

Bemerkung 20:

In der Tat ist das Bild $\psi_i^{(n,\ k)}(\pi)$ eines $\pi \in W_n^{(k)}$ ein Element der Menge $W_n^{(k+1)}$.
Denn die neue Definitionsmenge ist offensichtlich $\{1,\ \dots\ ,\ n-k-1\}$ und die Bildmenge ist $\{1,\ 2,\ \dots\ ,\ n\}$.
Weiter ist die entstandene Abbildung injektiv, da jedes Element der Menge

$$\{\pi(1),\ \dots\ ,\ \pi(n-k)\} \setminus \{\pi(i)\}$$

genau einmal „getroffen“ wird.
Anschauchlich bedeutet diese Definition, dass aus dem Stapel

$$\begin{pmatrix}1 & \dots & i-1 & i & i+1 & \dots & n-k\\ \pi(1) & \dots & \pi(i-1) & \pi(i) & \pi(i+1) & \dots & \pi(n-k)\end{pmatrix}$$

der Stapel

$$\begin{pmatrix}1 & \dots & i-1 & i & i+1 & \dots & n-k-1\\ \pi(i-1) & \dots & \pi(1) & \pi(i+1) & \pi(i+2) & \dots & \pi(n-k)\end{pmatrix}$$

wird.
Die obere Zeile ist nun $(1,\ 2,\ \dots\ ,\ n-k-1)$ und die untere Zeile besteht genau aus den Elementen Menge

$$\{\pi(1),\ \dots\ \pi(i-1),\ \pi(i+1),\ \dots\ ,\ \pi(n-k)\} = \{\pi(1),\ \dots\ ,\ \pi(n-k)\} \setminus \{\pi(i)\}$$

Die entstande Abbildung ist injektiv und es gilt $\psi_i^{(n,\ k)}(\pi) \in W_n^{(k+1)}$.

Beispiel 21: WUE-Operation

Es sei $n := 5$ und $k := 0$. Weiter sei

$$\pi := \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 3 & 2 & 4 & 5 & 1 \end{pmatrix} \in S_5$$

der Beispielstapel der Aufgabenstellung und $i := 3$. Dann ist

$$\psi_3^{(5,\ 0)}(\pi) = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 2 & 3 & 5 & 1 \end{pmatrix} =: \pi'.$$

Denn fuer den dritten und vierten Pfannkuchen gilt $3,\ 4 \geq 3 = i$, sodass nach Definition $\pi'(4) = \pi(4+1) = \pi(5) = 1$, sowie $\pi'(3) = \pi(3+1) = \pi(4) = 5$ gilt. Weiter gilt fuer den ersten und den zweiten Pfannkuchen, dass $1,\ 2 < 3 = i$, sodass $\pi'(2) = \pi(3-2) = \pi(1) = 3$, sowie $\pi'(1) = \pi(3-1) = \pi(2) = 2$.
Auszerdem ist

$$\psi_2^{(5,\ 1)}(\pi') = \begin{pmatrix} 1 & 2 & 3 \\ 2 & 5 & 1 \end{pmatrix} =: \sigma,$$

da wegen $2,\ 3 \geq 2 = i$, $\sigma(3) = \pi'(3+1) = \pi'(4) = 1$, und $\sigma(2) = \pi'(2+1) = \pi'(3) = 5$ und wegen $1 < 2$, $\sigma(1) = \pi'(2-1) = \pi'(1) = 2$.

Satz 22: Eigenschaften der Abbildung $\psi_i^{(n,\ k)}$

Es seien $n \in \mathbb{N}$, mit $n \geq 2$, $k \in \mathbb{N}_0$ mit $0 \leq k \leq n-2$ und $i \in \mathbb{N}$ mit $1 \leq i \leq n-k$, dann ist die Abbildung

$$\psi_i^{(n,\ k)} : W_n^{(k)} \longrightarrow W_n^{(k+1)}$$

surjektiv, aber fuer $k \neq 0$ nicht injektiv[9]. D.h., fuer $k = 0$ ist $\psi_i^{(n,\ k)}$ sowohl surjektiv als auch injektiv, also bijektiv.

Beweis:
Es seien $n \in \mathbb{N}$, mit $n \geq 2$, $k \in \mathbb{N}_0$ mit $0 \leq k \leq n-2$ und $i \in \mathbb{N}$ mit $1 \leq i \leq n-k$.

1. Z.z.: $\psi_i^{(n,\ k)}$ ist surjektiv. D.h.,

   $$\big(\forall \sigma \in W_n^{(k+1)}\big)\big(\exists \pi \in W_n^{(k)}\big)\big[\ \psi_i^{(n,\ k)}(\pi) = \sigma\ \big]$$

   Es sei $\sigma \in W_n^{(k+1)}$ mit

   $$\sigma = \begin{pmatrix} 1 & \dots & i-1 & i & i+1 & \dots & n-k-1 \\ \sigma(1) & \dots & \sigma(i-1) & \sigma(i) & \sigma(i+1) & \dots & \sigma(n-k-1) \end{pmatrix}.$$

   Weiter sei

   $$\pi = \begin{pmatrix} 1 & \dots & i-1 & i & i+1 & i+2 & \dots & n-k \\ \sigma(i-1) & \dots & \sigma(1) & x & \sigma(i) & \sigma(i+1) & \dots & \sigma(n-k-1) \end{pmatrix}$$

   fuer

   $$x \in \{1,\ 2,\ \dots,\ n\} \setminus \{\sigma(1),\ \dots,\ \sigma(n-k-1)\},$$

   sodass

   $$\psi_i^{(n,\ k)}(\pi) = \begin{pmatrix} 1 & \dots & i-1 & i & i+1 & \dots & n-k-1 \\ \sigma(1) & \dots & \sigma(i-1) & \sigma(i) & \sigma(i+1) & \dots & \sigma(n-k-1) \end{pmatrix} = \sigma$$

   ist.
   Somit existiert in der Tat fuer jedes $\sigma \in W_n^{(k+1)}$ ein $\pi \in W_n^{(k)}$, sodass $\psi_i^{(n,\ k)}(\pi) = \sigma$.

2. Z.z.: $\psi_i^{(n,\ k)}$ ist fuer $k \neq 0$ nicht injektiv und fuer $k = 0$ injektiv.
   Nach Satz 18 gilt, dass die Groesze der Definitionsmenge $W_n^{(k)}$

   $$\left|W_n^{(k)}\right| = \frac{n!}{k!}$$

   ist, waehrend die Groesze der Bildmenge $W_n^{(k+1)}$ (nur)

   $$\left|W_n^{(k)}\right| = \frac{n!}{(k+1)!}$$

   ist.
   Das heiszt, dass sich die Definitionsmenge um einen Faktor von $(k+1)$ groeszer ist als die Bildmenge.

[9] Es ist gemeint, dass i (sowie n und m) beliebig aber fest sind.

(a) $k = 0$
Fuer $k = 0$ ist

$$\mid W_n^{(k)} \mid = \frac{n!}{k!} = \frac{n!}{0!} = n! = \frac{n!}{1!} = \frac{n!}{(0+1)!} = \frac{n!}{(k+1)!} = \mid W_n^{(k+1)} \mid$$

D.h., die Definitionsmenge und die Bildmenge sind gleich grosz (und endlich). Da $\psi_i^{(n,\ k)}$ surjektiv ist, folgt daraus sofort, dass (fuer $k = 0$) $\psi_i^{(n,\ k)}$ auch injektiv ist. Somit ist $\psi_i^{(n,\ k)}$ fuer $k = 0$ bijektiv.

(b) $k \neq 0$
Fuer $k \neq 0$, also fuer $0 < k \leq n-2$ ist $k+1 > 1$, sodass die Definitionsmenge um einen Faktor $k+1 > 1$ groeszer ist als die Bildmenge (und beide sind endlich). Da $\psi_i^{(n,\ k)}$ surjektiv ist, folgt daraus sofort, dass (fuer $k \neq 0$) $\psi_i^{(n,\ k)}$ nicht injektiv ist.

Somit ist $\psi_i^{(n,\ k)}$ surjektiv, fuer $n \neq 0$ nicht injektiv und fuer $k = 0$ injektiv, also fuer $k = 0$ bijektiv. □

Definition 23: Operationsfolge

Es sei $n \in \mathbb{N}$ und $m \in \mathbb{N}_0$ mit $0 \leq m \leq n-1$. Weiter sei

$$\Lambda_n^{(m)} \coloneqq \left\{ \lambda = (\lambda_1,\ \ldots\ ,\ \lambda_m) : \big(\forall j \in \{1,\ \ldots\ ,\ m\}\big) \big[\ \lambda_j \in \{1,\ \ldots\ ,\ n-j+1\}\ \big] \right\}$$

die Menge aller moeglichen Operationsfolgen der Laenge m auf Permutationen (Pfannkuchenstapeln) der Laenge n.
D.h., fuer eine Operationsfolge $\lambda = (\lambda_1,\ \ldots\ ,\ \lambda_m) \in \Lambda_n^{(m)}$ gilt, dass fuer jedes $j \in \{1,\ \ldots\ ,\ m\}$,

$$\lambda_j \in \{1,\ \ldots\ ,\ n-j+1\}.$$

Fuer $m = 0$ sei

$$\Lambda_n^{(m)} = \Lambda_n^{(0)} \coloneqq \{(\ )\}$$

die Menge, die eine leere Operationsfolge enthaelt.

Satz 24: Maechtigkeit der Menge $\Lambda_n^{(m)}$

Es sei $n \in \mathbb{N}$ und $m \in \mathbb{N}_0$ mit $0 \leq m \leq n-1$. Dann ist die Maechtigkeit (Groesze) der Menge $\Lambda_n^{(m)}$, also die Anzahl an moeglichen Operationsfolgen der Laenge m, genau

$$\left|\Lambda_n^{(m)}\right| = \frac{n!}{(n-m)!}$$

Beweis:
In einer Operationsfolge $\lambda \in \Lambda_n^{(m)}$ gibt es fuer das erste Element $\lambda_1 \in \{1,\ \ldots\ ,\ n\}$ genau n Moeglichkeiten, fuer das zweite Element $\lambda_2 \in \{1,\ \ldots\ ,\ n-1\}$ genau $n-1$ Moeglichkeiten, usw. bis fuer das m-te Element, fuer welches es genau $n-m+1$ Moeglichkeiten gibt. Multipliziert man diese Werte, so ergibt sich die Maechtigkeit der Menge $\Lambda_n^{(m)}$.

$$\begin{aligned} &\left|\Lambda_n^{(m)}\right| \\ =&(n) \cdot (n-1) \cdot \cdots \cdot (n-m+2) \cdot (n-m+1) \\ =&\frac{n!}{(n-m)!} \end{aligned}$$

Definition 25: Anwendung von Operationsfolgen

Es sei $n \in \mathbb{N}$ und $m \in \mathbb{N}_0$, mit $0 \leq m \leq n-1$. Weiter sei $\pi \in S_n$ und

$$\lambda = (\lambda_1,\ \ldots\ ,\ \lambda_m) \in \Lambda_n^{(m)}.$$

Dann ist die Abbildung

$$\psi_\lambda^{(n)} : W_n^0 = S_n \longrightarrow W_n^{(m)}$$

wie folgt definiert:

$$\psi_\lambda^{(n)} : S_n \ni \pi \longmapsto \left(\psi_{\lambda_k}^{(n,\ m-1)} \circ \psi_{\lambda_{k-1}}^{(n,\ m-2)} \circ \cdots \circ \psi_{\lambda_2}^{(n,\ 1)} \circ \psi_{\lambda_1}^{(n,\ 0)}\right)\Big(\pi\Big) \in W_n^{(m)}$$

Das heiszt, dass

$$\begin{aligned}\psi_{\lambda}^{(n)}(\pi) &= \left(\psi_{\lambda_k}^{(n,\ m-1)} \circ \psi_{\lambda_{k-1}}^{(n,\ m-2)} \circ \cdots \circ \psi_{\lambda_2}^{(n,\ 1)} \circ \psi_{\lambda_1}^{(n,\ 0)}\right)\left(\pi\right) \\ &= \psi_{\lambda_k}^{(n,\ m-1)}\left(\psi_{\lambda_{k-1}}^{(n,\ m-2)}\left(\cdots \psi_{\lambda_2}^{(n,\ 1)}\left(\psi_{\lambda_1}^{(n,\ 0)}(\pi)\right)\right)\right)\end{aligned}$$

Fuer $m = 0$ muss $\lambda = (\ )$ sein, und $\psi_{\lambda}^{(n)}$ ist wie folgt definiert:

$$\psi_{()}^{(n)}(\pi) \coloneqq \pi$$

Bemerkung 26: Operationsfolge

Die Definition, dass fuer eine Operationsfolge $\lambda = (\lambda_1,\ \ldots\ ,\ \lambda_m) \in \Lambda_n^{(m)}$ gilt, dass fuer jedes $j \in \{1,\ \ldots\ ,\ m\}$

$$\lambda_j \in \{1,\ \ldots\ ,\ n - j + 1\},$$

ist sinnvoll, da dadurch gewaehrleitstet ist, dass der Index, der je fuer die WUE-Operationen (siehe Def. 22) genutzt wird, im passenden Bereich ist, also fuer die aktuelle Groesze des Pfannkuchenstapel sinnvoll ist.

Beispiel 27: Operationsfolgen

Es sei $n \coloneqq 5$ und $m \coloneqq 2$. Dann ist

$$\lambda = (\lambda_1,\ \lambda_2) \coloneqq (3,\ 2) \in \Lambda_n^{(m)}$$

eine moegliche Operationsfolge, nach Def. **??**, der Laenge 2.
Denn es gilt $\lambda_1 \in \{1,\ 2,\ \cdots\ ,\ 5\} = \{1,\ 2,\ \cdots\ ,\ 5 - 1 + 1\}$, und $\lambda_2 \in \{1,\ 2,\ \cdots\ ,\ 4\} = \{1,\ 2,\ \cdots\ ,\ 5 - 2 + 1\}$.
Weiter sei

$$\pi \coloneqq \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 3 & 2 & 4 & 5 & 1 \end{pmatrix} \in S_5.$$

Dann ist

$$\psi_{\lambda}^{(n)}(\pi) = \psi_{\lambda_2}^{(n,\ 1)}\left(\psi_{\lambda_1}^{(n,\ 0)}(\pi)\right) = \psi_2^{(5,\ 1)}\left(\psi_3^{(5,\ 0)}(\pi)\right)$$

Nach Bsp. 21 folgt:

$$\begin{aligned}&\psi_2^{(5,\ 1)}\left(\psi_3^{(5,\ 0)}(\pi)\right) \\ &= \psi_2^{(5,\ 1)}\left(\psi_3^{(5,\ 0)}\begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 3 & 2 & 4 & 5 & 1 \end{pmatrix}\right) \\ &= \psi_2^{(5,\ 1)}\begin{pmatrix} 1 & 2 & 3 & 4 \\ 2 & 3 & 5 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 2 & 3 \\ 2 & 5 & 1 \end{pmatrix}\end{aligned}$$

Bemerkung 28: Operationsfolgen - Notation

Es sei $n \in \mathbb{N}$ und $\pi \in S_n$ eine Permutation, bzw. ein Pfannkuchenstapel, sowie $m \in \mathbb{N}_0$ mit $0 \leq m \leq n - 1$ und

$$\lambda = (\lambda_1,\ \ldots\ ,\ \lambda_m) \in \Lambda_n^{(m)}.$$

eine Operationsfolge der Laenge m. Weiter sei

$$\sigma \coloneqq \psi_{\lambda}^{(n)}(\pi).$$

Dann wird im Folgenden auch gesagt, dass π durch λ in σ ueberfuehrt wird.
Auszerdem wird dafuer ebenfalls folgende Schreibweise verwendet:

$$\pi \xrightarrow{\lambda_1} \cdots \xrightarrow{\lambda_m} \sigma$$

Definition 29: Sortierter Pfannkuchenstapel

Es seien $n \in \mathbb{N}$, $k \in \mathbb{N}_0$ mit $0 \leq k \leq n - 1$ und $\pi \in W_n^{(k)}$, wobei

$$\pi = \begin{pmatrix} 1 & \ldots & n-k \\ \pi(1) & \ldots & \pi(n-k) \end{pmatrix}.$$

Dann nennt man π sortiert genau dann, wenn

$$\Big(\forall i \in \{1,\ 2,\ \ldots n - k - 1\}\Big)\Big[\ \pi(i) < \pi(i+1)\ \Big].$$

D.h., auch, wenn $k = n - 1$ gilt.

Beispiel 30: Sortierter Pfannkuchenstapel

Fuer $n \in \mathbb{N}$ und $k = 0$ ist aus der Menge $W_n^{(0)} = S_n$ nur das Neutralelement

$$\mathrm{id} \coloneqq \begin{pmatrix} 1 & 2 & \dots & n \\ 1 & 2 & \dots & n \end{pmatrix} \in S_n$$

sortiert, denn fuer jedes $i \in \{1,\ 2,\ \dots\ ,\ n-1\}$ ist $\pi(i) < \pi(i+1)$.
Fuer $n = 5$ und $k = 2$ ist beispielsweise

$$\begin{pmatrix} 1 & 2 & 3 \\ 2 & 3 & 5 \end{pmatrix} \text{ oder } \begin{pmatrix} 1 & 2 & 3 \\ 3 & 4 & 5 \end{pmatrix}$$

sortiert.

Satz 31:

Ist $n \in \mathbb{N}$ und $\pi \in S_n$ eine Permutation, bzw. ein Pfannkuchenstapel, sowie

$$\lambda = \left(\lambda_1,\ \dots\ ,\ \lambda_m\right) \in \Lambda_n^{(m)}.$$

eine Operationsfolge der Laenge $m = n - 1$, dann wird π durch λ immer in einen sortierten Stapel ueberfuehrt.

Beweis:
Durch eine Operationsfolge der Laenge $n-1$ wird π nach Def. 28 stets in einen Stapel der Menge $W_n^{(n-1)}$ ueberfuehrt. Somit besitzt dieser Stapel die Groesze $n - (n-1) = 1$ und ist nach Def. **??** immer sortiert.
Dies gilt insbesondere auch fuer $n = 1$.

Definition 32: $A(\pi)$ fuer $\pi \in S_n$

Es sei $n \in \mathbb{N}$ und $\pi \in S_n$. Dann ist

$$A(\pi) \coloneqq \min\left\{ m \in \{0,\ 1,\ \dots\ ,\ n-1\} : \left(\exists \lambda \in \Lambda_n^{(m)}\right)\left[\psi_\lambda^{(n)}(\pi)\ \text{ ist sortiert }\right] \right\}$$

die kleinste Zahl $m \in \{0,\ 1,\ \dots\ ,\ n-1\}$, sodass eine Operationsfolge λ dieser groesze existiert, durch die π in einen sortierten Stapel $\psi_\lambda^{(n)}(\pi)$ ueberfuehrt wird. Die Menge der Zahlen $\{0,\ 1,\ \dots\ ,\ n-1\}$ reicht dabei nach Satz 31 offensichtlich aus.

Definition 33: $P(n)$ fuer $n \in \mathbb{N}$

Es sei $n \in \mathbb{N}$. Dann ist

$$P(n) \coloneqq \max\left\{ A(\pi) : \pi \in S_n \right\}$$

die groeszte Zahl, die $A(\pi)$ fuer eine beliebige Permutation (Pfannkuchenstapel) der Groesze n erreicht.

Definition 34: Das Problem der Stapelsortierung (MWO-Problem) - (a)

Das Problem der ersten Teilaufgabe (a) wird nun formal definiert.
Es soll fuer einen Pfannkuchenstapel $\pi \in S_n$ mit $n \in \mathbb{N}$ das zuvor definierte $A(\pi) =: m$ bestimmt werden und eine dazugehoerige Operationsfolge der Laenge m, die π tatsaechlich in einen sortierten Stapel ueberfuehrt:

$$\lambda \in \Lambda_n^{(m)} \text{ mit } \psi_\lambda^{(n)}\left(\pi\right) \text{ ist sortiert.}$$

Definition 35: Das Problem der PWUE-Zahlen (PWZ-Problem) - (b)

Das Problem der zweiten Teilaufgabe (n) wird nun formal definiert.
Es soll fuer $n \in \mathbb{N}$ das zuvor definierte $P(n) =: m$ bestimmt werden und ein Beispielpfannkuchenstapel $\pi \in S_n$ angegeben werden, welcher tatsaechlich mindestens eine Operationsfolge der Laenge m benoetigt, um in einen sortierten Stapel ueberfuehrt zu werden. D.h.,

$$A(\pi) = P(n)$$

3. Loesungsvorschlaege I - Originale WUE-Operation

Um die zuvor definierte WUE-Operation auf einem Pfannkuchenstapel am Index i, umzusetzen kann aus der Definition 19 einfach ein iterativer Algorithmus definiert werden. Im Folgenden wird allerdings immer die zuvor definierte Abbildung im Pseudocode verwendet werden.
Dafuer interpretiert man den Stapel als Liste von Zahlen der Laenge n und definiert dann eine neue Liste, die das Ergebnis sein wird. Nun wird mit Hilfe einer Schleife durch die Indizes x von 1 bis $n-1$ iteriert und nach Definition 19 das naechste Element der Ergebnisliste bestimmt. Dazu wird unterschieden zwischen dem Fall, dass $x \geq i$ und dem, dass $x \leq i$ is. Im ersten Fall ist das naechste Element der Liste $\pi[x+1]$ und im zweiten Fall $\pi[i-x]$. Zum Schluss wird die Ergebnisliste zurueckgegeben.

Algorithmus 1 : WUE-Operation

Input : Stapel $\pi = (\pi_1, \ \dots \ , \ \pi_n)$ der Laenge $n \in \mathbb{N}$ mit $n \geq 2$ und Index $i \in \mathbb{N}$ mit $1 \leq i \leq n$
Output : $\pi' = (\pi'_1, \ \dots \ , \ \pi'_{n-1})$

```
1 π' ← ( ) // Leere Liste
2 for x ← 1 to n − 1 do
3 |   if x ≥ i then
4 |   |   π'.add(π(x + 1))
5 |   else
6 |   |   π'.add(π(i − x))
7 |   end if
8 end for
9 return π'
```

Um weiter eine Operationsfolge Anzuwenden (siehe Def. 28), kann der soeben definierte Algorithmus iterativ mehrmals angewendet werden:

Algorithmus 2 : Operationsfolge Anwenden

Input : Stapel $\pi = (\pi_1, \ \dots \ , \ \pi_n)$ der Laenge $n \in \mathbb{N}$ und Operationsfolge $\lambda = (\lambda_1, \ \dots \ , \ \lambda_m) \in \Lambda_n^{(m)}$ mit $m \in \mathbb{N}_0$ und $0 \leq m \leq n-1$
Output : $\pi' = (\pi'_1, \ \dots \ , \ \pi'_{n-m})$

```
1 π' ← π // Kopie!
2 for i ← 1 to m do
3 |   π' ← ψ_{λ_i}^{(n, i−1)}(π')
4 end for
5 return π'
```

3.1. Stapelsortierung / MWO-Problem (Aufgabenteil a)

Nach dem das Problem der ersten Teilaufgabe (a) nun in Definition 34 und Definition 35 formal definiert wurde, lassen es sich jetzt relativ einfach Algorithmen finden, die das MWO-Problem (Min-WUE-Operations-Problem) loesen koennen.

Es sei $n \in \mathbb{N}$ und $\pi \in S_n$ ein Pfannkuchenstapel der Groesze n. Nunn soll ein Algorithmus, der das Problem loesst diesen Pfannkuchenstapel π entgegen nehmen, und eine Operationsfolge $\lambda \in \Lambda_n^{(m)}$ finden, die π in einen sortierten Pfannkuchenstapel $\pi' \in W_n^{(m)}$ der Laenge $n-m$ ueberfuehrt. Dabei ist $m = A(\pi)$. D.h., die Operationsfolge λ hat die minimale Laenge, die gebraucht wird, um π mit der Hilfe von WUE-Operationen in einen sortierten Stapel zu ueberfuehren.

Der erste Algorithmus (Algorithmus 3) geht so vor, dass durch alle moeglichen $m \in \{0,\ 1,\ \dots\ ,\ n-1\}$ iteriert wird und fuer jedes dieser moeglichen m je durch alle moeglichen Operationsfolgen $\lambda \in \Lambda_n^{(m)}$ iteriert. Fuer die aktuelle Operationsfolge λ wird jeweils $\sigma = \psi_\lambda^{(n)}$ bestimmt, um anschlieszend zu pruefen, ob der entstandene Stapel σ sortiert ist. Also ob

$$\Big(\forall i \in \{1,\ 2,\ \dots\ ,\ n-m-1\}\Big)\Big[\sigma(i) < \pi(i+1)\Big]$$

erfuellt ist (vergleiche Definition 29).
Ist dies der Fall, so wird die aktuelle Operationsfolge zurueckgegeben. In der Tat wird dadurch die minimale Groesze fuer den Rueckgabewert λ gefunden, da **alle** moeglichen Groeszen der Operationsfolge aufsteigend ausprobiert werden. D.h., sobald eine Operationsfolge gefunden wird (also nach Satz 48 spaetestens fuer $m = n-1$), muessen alle vorherigen Operationsfolgen fuer kleinere m den gegebenen Stapel **nicht** in einen sortierten ueberfuert haben.

Algorithmus 3 : Iterativer Algorithmus (MWO-Problem)

Input : Permutation $\pi \in S_n$ der Laenge $n \in \mathbb{N}$
Output : $\lambda = (\lambda_1,\ \dots\ ,\ \lambda_m)$ mit $m \in \{0\ ,1,\ \dots\ n-1\}$

```
1 for m ← 0 to n − 1 do
2 |  for λ ∈ Λ_n^(m) do
3 |  |  σ ← ψ_λ^(n)(π)
4 |  |  // Pruefe, ob σ sortiert ist
5 |  |  if (∀i ∈ {1, 2, ...n − m − 1}) [ σ(i) < σ(i + 1) ] then
6 |  |  |  return λ
7 |  |  end if
8 |  end for
9 end for
```

Um in der Praxis tatsaechlich durch alle Elemente der Menge $\Lambda_n^{(m)}$ zu iterieren, wobei $n \in \mathbb{N}$ und $m \in \mathbb{N}_0$ mit $0 \leq m \leq n-1$, kann ein rekursiver Algorithmus genutzt werden (siehe Algorithmus 4, der die Operationsfolgen rekursiv erzeugt. Dazu wird zunaechst die Abbruchbedingung geprueft: $|\lambda| = m$. In diesem Fall ist die Operationsfolge bereits fertig und kann ausgeben (bzw. getestet werden). Ansonsten wird durch alle moeglichen Indizes fuer $\lambda_{|\lambda|+1}$ (nach Def. 23) die Elemente der Menge $\{1,\ \dots\ ,\ n-|\lambda|\}$) iteriert. Fur jeden moeglichen Index i wird dieser ans Ende der Operationsfolge (Interpretation als Liste!) angefuegt und die Funktion wird rekursiv mit der Liste λ als Referenz(!) als Uebergabeparameter aufgerufen, sodass anschlieszend alle Operationsfolgen ausgegeben werden, die mit der aktuellen Teiloperationsfolge anfangen. Danach wird das letzte Element der List (Operationsfolge) λ wieder entfernt.
Um die gesamte rekursive Funktion zu starten, wird am Anfang der Funktion mit einer leeren Liste ( ) und m als Uebergabeparameter aufgerufen.

Algorithmus 4 : Rekursiver Algorithmus zum Ausgeben der Elemente der Menge $\Lambda_n^{(m)}$

Input : $n \in \mathbb{N}$ und $m \in \mathbb{N}_0$ mit $0 \leq m \leq n-1$

```
1  printAll(( ), m)
2  Function printAll(λ, m):
3      if |λ| = m then
4          PRINT(λ) // Aktuelle Operationsfolge ausgeben
5          return
6      end if
7      for i ← 1 to (n − |λ|) do
8          λ.add(i)
9          printAll(λ, m)
10         λ.removeLastElement() // Remove element that was added last (line 8)
11     end for
```

Algorithmus 5 : Implementierung des Algorithmus 3 mit Hilfe des Algorithmus 4 (MWO-Problem)

Input : Permutation $\pi \in S_n$ der Laenge $n \in \mathbb{N}$

Output : $\lambda = (\lambda_1, \ldots, \lambda_m)$ mit $m \in \{0, 1, \ldots n-1\}$

```
1  for m ← 0 to n − 1 do
2      λ_result ← NULL
3      if checkAll(( ), m) then
4          return λ_result
5      end if
6  end for
7  Function checkAll(λ, m):
8      if |λ| = m then
9          σ ← ψ_λ^(n)(π)
10         // Pruefe, ob σ sortiert ist
11         if (∀i ∈ {1, 2, ...n − m − 1}) [ σ(i) < σ(i + 1) ] then
12             λ_result ← λ // Kopie!
13             return TRUE
14         else
15             return FALSE
16         end if
17     end if
18     for i ← 1 to (n − |λ|) do
19         λ.add(i)
20         if checkAll(λ, m) then
21             return TRUE
22         end if
23         λ.removeLastElement() // Remove element that was added last (line 19)
24     end for
25     return FALSE
```

Mit Hilfe des Algorithmus 4 kann nun der erste Algorithmus (Algorithmus 3) sinnvoll, bzw. konkret implementiert werden (siehe Algorithmus 5). Dazu wird erneut jedes moegliche m von 0 aufsteigend zu $n-1$ ausprobiert. Fuer jedes m wird nun eine globale Variable λ_{result} mit $NULL$ initialisiert. D.h., diese wird auch in der, unter der Schleife, definierten Funktion *CheckAll* benutzbar sein. Anschlieszend wird die Funktion checkAll mit den Uebergabeparametern ( ) (leere Liste) und dem aktuellen m ausgefuehrt und falls diese **TRUE** zurueckgibt, wird λ_{result} als Ergebnis zurueckgegeben.
Die Funktion checkAll funktioniert aehnlich, wie die printAll Funktion des letzte Algorithmus.
Sie probiert rekursiv alle moeglichen Operationsfolgen und testet dann, wie in Algorithmus 3 , ob die aktuelle Operationsfolge das gegebene π in einen sortierten Stapel ueberfuehrt. Ist dies der Fall, wird λ_{result} via Kopie auf den Wert der aktuellen Operationsfolge λ gesetzt und **TRUE** zurueckgegeben. Andernfalls wird fuer die aktuelle Opertionsfolge **FALSE** zurueckgegeben. Auszerdem wird in der Schleife, die alle moeglichen Indizes fuer die naechste Stelle der Operationsfolge durchiteriert, beim rekursiven Aufrufen der Funkton, geprueft, ob diese **TRUE** zurueckgegeben hat. Ist dies der Fall wird ebenfalls **TRUE** zurueckgegeben. Dadurch wird, nachdem eine Operationsfolge gefunden wurde, die π in einen sortierten Stapel ueberfuehrt, keine weitere mehr ausprobiert.

Bemerkung fuer $m = 0$ oder $m = n - 1$:
Fuer den Fall, dass $m = 0$ gilt, wird bereits beim ersten Aufruf der Funktion der Abbruchfall erreicht, sodass sofort ausprobiert wird, ob die leere Liste (Operationsfolge) π sortiert. Nach Definition 28, ist σ dann gleich π, sodass nur **TRUE** zurueckgegeben wird, wenn $\pi = \text{id}$ gilt.

Weiter wird fuer den Fall $m = n - 1$ wird zunaechst eine Operationsfolge vollstaendig erzeugt, woraufhin nach Satz 48 das Ergebnis der WUE-Operationen σ auf jeden Fall sortiert ist, so dass keine weitere Operationsfolge erzeugt wird, und die gefundene Operationsfolge zurueckgegeben wird.

Die zweite Moeglichkeit, das MWO-Problem zu loesen ist es, den Algorithmus (bzw. das Problem) rekursiv zu formulieren. Fuer einen Stapel urspruenglichen Stapal $\pi \in S_n$ der Laenge n kann eine rekursive Funktion definiert werden, die einen Stapel τ entgegen nimmt:
Im Basisfall, dass der gegebene Stapel τ ist sortiert ist (insbesondere fuer $|\tau| = 1$), wird eine leere Liste (Operationsfolge) zurueckgegeben, da keine WUE-Operatione benoetigt werden, um τ zu sortieren.
Sonst kann man alle moeglichen WUE-Operationen index $= 1,\ 2,\ \dots\ ,\ |\tau|$ ausprobieren, und die fuer den entstehenden Stapel $\sigma = \psi_{\text{index}}^{(n,\ n-|\tau|)}(\tau)$ den Algorithmus rekursiv anwenden. Die Operationsfolge nur bestehend aus dem index gefolgt von der, die fuer σ zurueckgegeben wurde, besitzt nun die kleinst moegliche Laenge fuer eine Operationsfolge, die τ sortiert und mit index anfaengt. Nimmt man nun die kleinste Operatiosfolge, die fuer einen Startindex gefunden wurde, so hat man auch die insgesamt kleinste Operatonsfolge gefunden, die τ sortiert.

Dieses Verfahren wurde in Algorithmus 6 mit der Funktion *Foo* umgesetzt. Dabei wird mit Hilfe einer vor der Schleife definierten Variable die bisher beste gespeichert und geupdated.
Ruft man also *Foo* mit π als Uebergabeparameter auf, erhaelt man die gesuchte Operationsfolge.

Algorithmus 6 : Rekursive Loesung des MWO-Problems

Input : Permutation $\pi \in S_n$ der Laenge $n \in \mathbb{N}$
Output : $\lambda = (\lambda_1,\ \dots\ ,\ \lambda_m)$ mit $m \in \{0\ ,1,\ \dots\ n-1\}$

```
return Foo(π)
Function Foo(τ):
    // Pruefe, ob τ sortiert ist
    if (∀i ∈ {1, 2, ...|τ| − 1}) [ τ(i) < τ(i + 1) ] then
        return ( )
    end if
    (λ_result, m) ← (NULL, ∞)
    for i ← 1 to |τ| do
        σ ← ψ_i^(n, n−|τ|)(τ)
        λ' ← ( i ).add(Foo(σ))
        if |λ'| < m then
            (λ_result, m) ← (λ', |λ'|)
        end if
    end for
    return λ_result
```

3.2. PWUE-Zahlen / PWZ-Problem (Aufgabenteil b)

Um im Folgenden das PWZ-Problem zu loesen (D.h., $P(n)$ fuer $n \in \mathbb{N}$ bestimmen und ein dazugehoeriges Beispiel $\pi \in S_n$ bestimmen das tatsaechlich mindestens $P(n)$ WUE-Operationen braucht, um sortiert zu werden) werden zunaechst zwei Verfahren vorgestellt, die $A(\pi)$ bestimmen.

Das erste Verfahren, um $A(\pi)$ fuer $\pi \in S_N$ zu bestimmen, funktioniert analog zu Algorithmus 5. Wobei nun fuer jedes moegliche $m \in \{0,\ 1,\ \ldots\ ,\ n-1\}$ nur noch geprueft wird, ob fuer die aktuelle Groesze m irgendeine Operationsfolge existiert, die π in einen sortierten Stapel ueberfuehren kann. Ist dies der Fall, wird das aktuelle m zurueckgegeben (erneut: spaetestens fuer $m = n - 1$).
Um zu pruefen, ob eine Operationsfolge der Laenge m existiert, die π in einen sortierten Stapel ueberfuehrt, werden durch das rekursive Verfahren des Algorithmus 4 wieder alle moeglichen Operationsfolgen der aktuellen Laenge m ausprobiert und fuer jede geprueft, ob sie π in einen sortierten Stapel ueberfuehrt. Ist dies der Fall, so wird TRUE zurueckgegeben und andernfalls FALSE.

Algorithmus 7 : Algorithmus um $A(\pi)$ zu bestimmen

Input : $\pi \in S_n$ mit $n \in \mathbb{N}$
Output : $A(\pi)$

```
1  Function getA(π):
2      for m ← 0 to n − 1 do
3          if checkA( ), m) then
4              return m
5          end if
6      end for
7  Function checkA(λ, m):
8      if |λ| = m then
9          σ ← ψ_λ^(n)(π)
10         // Pruefe, ob σ sortiert ist
11         if (∀i ∈ {1, 2, ...n − m − 1}) [ σ(i) < σ(i + 1) ] then
12             return TRUE
13         else
14             return FALSE
15         end if
16     end if
17     for i ← 1 to (n − |λ|) do
18         λ.add(i)
19         if checkA(λ, m) then
20             return TRUE
21         end if
22         λ.removeLastElement() // Remove element that was added last (line 17)
23     end for
24     return FALSE
```

Ein weiteres Verfahren, um $A(\pi)$ fuer $\pi \in S_n$ ($n \in \mathbb{N}$) zu bestimmen, analog zu Algorithmus 6, besteht darin, den Algorithmus (oder das Problem) rekursiv zu definieren (siehe Algorithmus 8).
Das heiszt konkret, dass der Basisfall weiterhin besteht, wenn der gegebene Stapel τ sortiert ist. In diesem Fall werden 0 WUE-Operationen benoetigt, um den Stapel zu sortieren. Sonst laesst sich $A(\tau)$ erneut ermitteln, indem alle moeglichen Indizes (von 1 bis zur Laenge $|\tau|$ des aktuellen Stapels) ausprobiert werden und jeweis fuer das Ergebnis σ der jeweiligen WUE-Operation rekursiv $A(\sigma)$ bestimmt wird. Denn dadurch laesst sich die kleinste Anzahl an WUE-Operationen des aktuellen Stapels τ bestimmen, indem das Minum aller rekursiv besimmter $A(\sigma)$'s plus eins genommen wird. Das $+1$ ist dabei notwendig, da bereits eine WUE-Operation ausgefuehrt wurde.

Algorithmus 8 : Rekursiver Algorithmus um $A(\pi)$ zu bestimmen

Input : $\pi \in S_n$ mit $n \in \mathbb{N}$
Output : $A(\pi)$

```
return getA(π)
Function getA(τ):
    // Pruefe, ob τ sortiert ist
    if (∀i ∈ {1, 2, ... |τ| − 1}) [ τ(i) < τ(i + 1) ] then
        return 0
    end if
    min ← ∞
    for i ← 1 to |τ| do
        σ ← ψ_i^(n, n−|τ|)(τ)
        A ← getA(σ) + 1
        min ← min{min, A}
    end for
    return min
```

Um nun tatsaechlich das PWZ-Problem zu loesen, wird schlicht fuer alle $\pi \in S_n$ mit Hilfe einem der zwei vorherigen Algorithmen $A(\pi)$ bestimmt. Zuvor werden Variablen initialisiert, die die aktuelle schlechteste Permutation (groesztes $A(\pi)$) speichern und das dazugehoerige $A(\pi)$.

Algorithmus 9 : Iterativer Algorithmus um $P(n)$ und Bsp. zu bestimmen (PWZ-Problem)

Input : $n \in \mathbb{N}$
Output : $P(n)$ und Beispiel $\pi \in S_N$

```
π_worst ← NULL
max ← 0
for π ∈ S_n do
    A ← getA(π)
    if A > max then
        max ← A
        π_worst ← π
    end if
end for
return (π_worst, max)
```

Um die Permutationen in der Praxiz effizient zu generieren, kann beispielsweise der Heap-Algorithmus (s. [10]) nach B.R. Heap aus dem Jahr 1963 benutzt werden. Dieser nimmt eine Liste P der Laenge N entgegen und gibt alle Permutationen der Liste aus. Weiter kann dieser auch nicht-rekursiv formuliert werden[10] und wurde wie folgt verwendet, um Algorithmus 9 umzusetzen[11]:

[10] Siehe auch `https://de.wikipedia.org/wiki/Heap-Algorithmus` und `https://sedgewick.io/wp-content/uploads/2022/03/2002PermGeneration.pdf` (Seite 16)
[11] Pseudocode von `https://de.wikipedia.org/wiki/Heap-Algorithmus`

Algorithmus 10 : Algorithmus um $P(n)$ und Bsp. mit Hilfe des Heap-Algorithmus zu bestimmen (PWZ-Problem)

Input : $n \in \mathbb{N}$
Output : $P(n)$ und Beispiel $\pi \in S_N$

```
1  π_worst ← NULL
2  max ← 0
3  P ← (1, 2, ... , n) // 0-basiert indexierte Liste
4  C = (c_0, c_1 ... , c_{n-1}) ← (0, 0, ... , 0)
5  j ← 1
6  while j < N do
7  |  if c_j < j then
8  |  |  if j ist grade then
9  |  |  |  Tausche 0-tes und j-tes Element von P
10 |  |  else
11 |  |  |  Tausche c_j-tes und j-tes Element von P
12 |  |  end if
13 |  |  // Aktuelle Permutation pruefen
14 |  |  π ← Die Permutation π, die aktuell durch P dargestellt wird. // Aus formalen Gruenden
15 |  |  A ← getA(π)
16 |  |  if A > max then
17 |  |  |  max ← A
18 |  |  |  π_worst ← π
19 |  |  end if
20 |  |  c_j ← c_j + 1
21 |  |  j ← 1
22 |  else
23 |  |  c_j ← 0
24 |  |  j ← j + 1
25 |  end if
26 end while
27 return (π_worst, max)
```

4. Vereinfachte Modellierung der WUE-Operation und Definition der Probleme

Um die gegebenen Probleme zu vereinfachen wird nun eine einfachere Definition der WUE-Operation angegeben, welche analog zur vorherigen genutzt werden kann, um die Probleme zu definieren.
Dabei ist die Abbildung, die die WUE-Operation modelliert nicht mehr zwischen $W_n^{(k)}$ und $W_n^{(k+1)}$ (fuer $n \in \mathbb{N}$, $k \in \mathbb{N}_0$ und $0 \leq k \leq n-2$), sondern zwischen S_n und S_{n-1} (fuer $n \in \mathbb{N}$ mit $n \geq 2$). Dies hat zu Folge, dass die Loesungen zwar identischen (TODO - ?) zu den vorherigen sind, jedoch deutlich schneller funktionieren.

Bemerkung 36: Pfannkuchenstapel

Im Folgenden wird gemaesz Definition 12 eine Permutation $\pi \in S_n$ fuer $n \in \mathbb{N}$ als Modellierung fuer einen Pfannkuchenstapel gesehen und die Worte werden teilweise gleichbedeutend genutzt.

Definition 37: Wende-und-Ess-Operation II

Nun wird eine Abbildung definiert die aus einem urspruenglichen (echten) Pfannkuchenstapel (vergl. Bemerkung 16) der Groesze $n \in \mathbb{N}$ mit $n \geq 2$ wieder einen urspruenglichen (echten) Pfannkuchenstapel. D.h., eine wirkliche Permutations der Zahlen $1,\ 2,\ \ldots\ ,\ n-1$.

Dazu wird analog zu Def. 19 eine Abbildung

$$\varphi_i^{(n)} : S_n \longrightarrow S_{n-1}$$

definiert, wobei $i,\ n \in \mathbb{N}$ und $1 \leq i \leq n$.

$$\varphi_i^{(n)} : S_n \ni \pi \longmapsto \left(\pi' : \{1,\ \ldots\ ,\ n-1\} \ni x \longmapsto \begin{cases} \pi(x+1), & \text{falls } x \geq i \text{ und } \pi(x+1) < \pi(i), \\ \pi(x+1)-1, & \text{falls } x \geq i \text{ und } \pi(x+1) > \pi(i), \\ \pi(i-x), & \text{falls } x < i \text{ und } \pi(i-x) < \pi(i), \\ \pi(i-x)-1, & \text{falls } x < i \text{ und } \pi(i-x) > \pi(i) \end{cases} \in \{1,\ 2,\ \ldots\ ,\ n-1\}\right) \in S_{n-1}$$

Bemerkung 38: WUE-Operation

In der Tat gilt fuer $i,\ n \in \mathbb{N}$, $n \geq 2$ und $1 \leq i \leq n$, dass

$$\varphi_i^{(n)}(\pi) \in S_{n-1}.$$

da fuer alle Indizes $x = 1,\ 2,\ \ldots\ ,\ n-1$ wird je genau einmal auf die Element der Menge $\{1,\ 2,\ \ldots\ ,\ n-1\}$ abegebildet. Denn sollte ein $\pi(x+1)$ bzw. $\pi(i-x)$ einmal groeszer sein als $\pi(i)$ wird es um eins verkleinert.

Beispiel 39: WUE-Operation

Es sei $n \coloneqq 5$ und $i \coloneqq 3$. Weiter sei

$$\pi \coloneqq \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 3 & 2 & 4 & 5 & 1 \end{pmatrix} \in S_5.$$

Dann ist $\pi(i) = \pi(3) = 4$ und

$$\varphi_i^{(n)}(\pi) = \begin{pmatrix} 1 & 2 & 3 & 4 \\ \pi(2) & \pi(1) & \pi(4)-1 & \pi(5) \end{pmatrix} = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 2 & 3 & 4 & 1 \end{pmatrix} =: \sigma.$$

Denn fuer

1. $\sigma(1)$ gilt: $\sigma(1) = \pi(i-1) = \pi(2) = 2$, da $1 < 3 = i$ und $\pi(2) = 2 < 4 = \pi(3) = \pi(i)$
2. $\sigma(2)$ gilt: $\sigma(2) = \pi(i-2) = \pi(1) = 3$, da $2 < 3 = i$ und $\pi(1) = 3 < 4 = \pi(3) = \pi(i)$
3. $\sigma(3)$ gilt: $\sigma(3) = \pi(3+1)-1 = \pi(4)-1 = 4$, da $3 \geq 3 = i$ und $\pi(4) = 5 > 4 = \pi(3) = \pi(i)$
4. $\sigma(4)$ gilt: $\sigma(4) = \pi(4+1) = \pi(5) = 1$, da $4 \geq 3 = i$ und $\pi(5) = 1 > 4 = \pi(3) = \pi(i)$

Satz 40: Eigenschaften der Abbildung $\varphi_i^{(n)}$

Es sei $n \in \mathbb{N}$ mit $n \geq 2$ und $i \in \mathbb{N}$ mit $1 \leq i \leq n$. Dann ist

$$\varphi_i^{(n)} : S_n \longrightarrow S_{n-1}$$

surjektiv, aber nicht injektiv.

Beweis:
Es sei $n \in \mathbb{N}$ mit $n \geq 2$ und $i \in \mathbb{N}$ mit $1 \leq i \leq n$.

1. $\varphi_i^{(n)}$ ist surjektiv.
   D.h.,
   $$\big(\forall \sigma \in S_{n-1}\big)\big(\exists \pi \in S_n\big)\big[\ \varphi_i^{(n)}(\pi) = \sigma\ \big]$$
   Es sei $\sigma \in S_{n-1}$ mit
   $$\sigma = \begin{pmatrix} 1 & \dots & i-1 & i & i+1 & \dots & n-1 \\ \sigma(1) & \dots & \sigma(i-1) & \sigma(i) & \sigma(i+1) & \dots & \sigma(n-1) \end{pmatrix}.$$
   Weiter sei
   $$\pi = \begin{pmatrix} 1 & \dots & i-1 & i & i+1 & i+2 & \dots & n \\ \sigma(i-1) & \dots & \sigma(1) & n & \sigma(i) & \sigma(i+1) & \dots & \sigma(n-1) \end{pmatrix},$$
   sodass
   $$\varphi_i^{(n)}(\pi) = \begin{pmatrix} 1 & \dots & i-1 & i & i+1 & \dots & n-1 \\ \sigma(1) & \dots & \sigma(i-1) & \sigma(i) & \sigma(i+1) & \dots & \sigma(n-1) \end{pmatrix} = \sigma$$
   ist.
   Somit existiert in der Tat fuer jedes $\sigma \in S_{n-1}$ ein $\pi \in S_n$, sodass $\varphi_i^{(n)}(\pi) = \sigma$.
2. $\varphi_i^{(n)}$ ist nicht injektiv.
   Da die Maechtigkeit der Definitionsmenge S_n nach Bem. 9 $n!$, und die Maechtigkeit der Bildmenge $(n-1)!$ ist, wobei fuer $n \geq 2$ stets $n! \neq (n-1)!$ gilt folgt daraus, dass $\varphi_i^{(n)}$ surjektiv ist, sofort, dass $\varphi_i^{(n)}$ nicht injektiv ist.

Somit ist $\varphi_i^{(n)}$ surjektiv, aber nicht injektiv. □

Definition 41: Anwendung von Operationsfolgen

Da sich fuer die moeglichen Indizes der (neuen) WUE-Operations nichts aendert wird im Folgenden die Menge $\Lambda_n^{(m)}$ nach Def. 23 verwendet, um moegliche Operationsfolgen λ zu beschreiben.

Es seien $n \in \mathbb{N}$ und $m \in \mathbb{N}_0$ mit $n \geq 2$ und $0 \leq m \leq n-1$. Weiter sei

$$\lambda = (\lambda_1,\ \dots\ ,\ \lambda_n) \in \Lambda_n^{(m)}$$

eine Operationsfolge. Nun wird die Abbildung

$$\varphi_\lambda^{(n)} : S_n \longrightarrow S_{n-m},$$

die mehrere die Anwendung mehrere WUE-Operationen darstell, definiert:

$$\varphi_\lambda^{(n)} : S_n \ni \pi \mapsto \Big(\varphi_{\lambda_m}^{(n-m+1)}(\pi) \circ \varphi_{\lambda_{m-1}}^{(n-m+2)}(\pi) \circ \dots \circ \varphi_{\lambda_2}^{(n-1)}(\pi) \circ \varphi_{\lambda_1}^{(n)}\Big)\Big(\pi\Big) \in S_{n-m}$$

Fuer $m = 0$ muss erneut $\lambda = (\ )$ gelten und fuer $\pi \in S_n$ gilt

$$\varphi_\lambda^{(n)}(\pi) = \varphi_{()}^{(n)}(\pi) \coloneqq \pi \in S_n.$$

Bemerkung 42: Operationsfolgen

Wie in Bem. 28 gilt auch fuer die neue Definition der WUE-Operation:
Ist $n \in \mathbb{N}$, $\pi \in S_n$ und $m \in \mathbb{N}_0$ mit $0 \leq m \leq n-1$, sowie $\lambda \in \Lambda_n^{(m)}$ und

$$\sigma \coloneqq \varphi_\lambda^{(n)}(\pi),$$

dann sagt man, dass π durch λ in σ ueberfuehrt wird und schreibt

$$\pi \xrightarrow{\lambda_1} \dots \xrightarrow{\lambda_m} \sigma$$

wobei stets darauf hingewiesen werden wird, wenn die neue Definition φ, anstatt der originalen Definition ψ der WUE-Operation gemeint ist.

Definition 43: Sortierte Pfannkuchenstapel

Analog zu Def. 29 laesst sich die Sortiertheit von originalen Pfannkuchenstapeln $\pi \in S_n$ sehr einfach definieren. Denn es existiert nach Beispiel 30 nur ein sortierter Pfannkuchenstapel bzw. Permutation der Menge S_n. Dies ist das Neutralelement

$$\mathrm{id} \coloneqq \begin{pmatrix} 1 & 2 & \dots & n \\ 1 & 2 & \dots & n \end{pmatrix} \in S_n.$$

Bemerkung 44:

Analog zu Satz 31 gilt auch fuer die neue Definition der WUE-Operation, dass ein $\pi \in S_n$ fuer $n \in \mathbb{N}$ durch eine beliebige Operationsfolge der Laenge $n-1$ in einen sortierten Stapel ueberfuehrt wird. Naemlich genau in das Neutralelement der Menge S_1, welches nach Definition 43 sortiert ist:

$$\mathrm{id} \coloneqq \begin{pmatrix} 1 \\ 1 \end{pmatrix} \in S_1$$

Bemerkung 45: Neue Definition der Probleme

Nachdem nun auf die vereinfachte Definition der WUE-Operations als Abbildung von S_n nach S_{n-1} eingegangen wurde, lassen sich die orginalen Probleme neu, aber aequivalent definieren, indem die neue WUE-Operations an Stelle der originalen genutzt wird.
Dies ist moeglich, weil es bei den Problemen (bzw. $A(\pi)$ und $P(n)$) nur um die relative Reihenfolge der Pfannkuchen im Pfannkuchenstapel geht. Das heiszt, dass die Pfannkuchen

$$\begin{pmatrix} 1 & 2 & 3 & 4 \\ 4 & 3 & 6 & 5 \end{pmatrix}, \text{ und } \begin{pmatrix} 1 & 2 & 3 & 4 \\ 3 & 2 & 5 & 4 \end{pmatrix} \in W_6^{(2)}$$

gleichbedeutend zum Stapel

$$\begin{pmatrix} 1 & 2 & 3 & 4 \\ 2 & 1 & 4 & 3 \end{pmatrix} \in S_4$$

sind, da die relative Reihenfolge der Elemente immer dieselbe ist. Diese Eigenschaft wird bei der neuen Definition der WUE-Operation beruecksichtigt.

Definition 46: $A(\pi)$ fuer $\pi \in S_n$

Es sei $n \in \mathbb{N}$ und $\pi \in S_n$. Dann ist

$$A(\pi) \coloneqq \min\left\{ m \in \{0,\ 1,\ \dots\ ,\ n-1\} : \left(\exists \lambda \in \Lambda_n^{(m)}\right)\left[\varphi_\lambda^{(n)}(\pi) = \mathrm{id} \in S_{n-m}\right] \right\}$$

die kleinste Zahl $m \in \{0,\ 1,\ \dots\ ,\ n-1\}$, sodass eine Operationsfolge λ dieser groesze existiert, durch die π in einen sortierten Stapel $\mathrm{id} \in S_{n-m}$ ueberfuehrt wird. Die Menge der Zahlen $\{0,\ 1,\ \dots\ ,\ n-1\}$ reicht dabei nach Satz 44 offensichtlich aus.

Definition 47: $P(n)$ fuer $n \in \mathbb{N}$

Es sei $n \in \mathbb{N}$. Dann ist

$$P(n) \coloneqq \max\left\{ A(\pi) : \pi \in S_n \right\}$$

erneut die groeszte Zahl, die $A(\pi)$ fuer eine beliebige Permutation (Pfannkuchenstapel) der Groesze n erreicht.

Definition 48: Das Problem der Stapelsortierung (MWO-Problem) - (a)

Das Problem der ersten Teilaufgabe (a) besteht nun darin fuer einen Pfannkuchenstapel $\pi \in S_n$ mit $n \in \mathbb{N}$ $A(\pi) =: m$ zu bestimmen und eine dazugehoerige Operationsfolge der Laenge m anzugeben, die π tatsaechlich in den sortierten Stapel id ueberfuehrt:

$$\varphi_\lambda^{(n)}(\pi) = \mathrm{id} \in S_{n-m}$$

Definition 49: Das Problem der PWUE-Zahlen (PWZ-Problem) - (b)

Das Problem der zweiten Teilaufgabe (b) besteht ebenso weiterhin darin, fuer ein $n \in \mathbb{N}$ das zuvor definierte $P(n) =: m$ zu bestimmen und einen Beispielpfannkuchenstapel $\pi \in S_n$ anzugeben, der tatsaechlich mindestens eine Operationsfolge der Laenge m benoetigt, um in einen sortierten Stapel ueberfuehrt zu werden. D.h.,

$$A(\pi) = P(n)$$

Satz 50:

Es sei $n \in \mathbb{N}$ und $k \in \mathbb{N}_0$ mit $0 \leq k \leq n-1$. Nun werden die Maechtigkeiten der Mengen S_{n-k} und $W_n^{(k)}$ verglichen. Beide Mengen Stellen - einmal nach der originalen, und einmal nach der vereinfachten Definition - Pfannkuchenstapel der Laenge $n-k$ der urspruenglichen Laenge n dar, die durch k WUE-Operationen veraendert wurden.

S_{n-k} hat nach Bemerkung 9 $(n-k)!$ Elemente, waehrend $W_n^{(k)}$ nach Satz 18 $\frac{n!}{k!}$ Elemente.
Weiter gilt

$$\frac{\frac{n!}{k!}}{(n-k)!} = \frac{n!}{k!} \cdot \frac{1}{(n-k)!} = \frac{n!}{k! \cdot (n-k)!} = \binom{n}{k}$$

Demnach ist die Menge $W_n^{(k)}$ um einen Faktor $\binom{n}{k}$ groeszer als die Menge S_{n-k}.

5. Loesungsvorschlaege II - Vereinfachte WUE-Operation

Um die neu definierte WUE-Operation auf einem Pfannkuchenstapel am Index i umzusetzen kann mit Hilfe der Definition 37 einfach ein iterativer Algorithmus definiert werden. Im Folgenden wird allerdings, wie bei der vorherigen WUE-Operation auch, immer die zuvor definierte Abbildung im Pseudocode verwendet werden. Dafuer interpretiert man den Stapel wieder als Liste von Zahlen der Laenge n und definiert dann eine neue Liste, die das Ergebnis sein wird.

Algorithmus 11 : Iterativer Algorithmus (MWO-Problem)

Input : Stapel $\pi = (\pi_1, \ \dots \ , \ \pi_n)$ der Laenge $n \in \mathbb{N}$ mit $n \geq 2$ und Index $i \in \mathbb{N}$ mit $1 \leq i \leq n$
Output : $\pi' = (\pi'_1, \ \dots \ , \ \pi'_{n-1})$

```
1  π' ← ( ) // Leere Liste
2  for x ← 1 to n − 1 do
3  |  if x ≥ i then
4  |  |  if π(x + 1) < π(i) then
5  |  |  |  π'.add(π(x + 1))
6  |  |  else
7  |  |  |  π'.add(π(x + 1) − 1)
8  |  |  end if
9  |  else
10 |  |  if π(i − x) < π(i) then
11 |  |  |  π'.add(π(i − x))
12 |  |  else
13 |  |  |  π'.add(π(i − x) − 1)
14 |  |  end if
15 |  end if
16 end for
17 return π'
```

Um weiter eine Operationsfolge Anzuwenden (siehe Def. 41), kann der soeben definierte Algorithmus iterativ mehrmals angewendet werden:

Algorithmus 12 : Iterativer Algorithmus (MWO-Problem)

Input : Stapel $\pi = (\pi_1, \ \dots \ , \ \pi_n)$ der Laenge $n \in \mathbb{N}$ und Operationsfolge $\lambda = (\lambda_1, \ \dots \ , \ \lambda_m) \in \Lambda_n^{(m)}$ mit $m \in \mathbb{N}_0$ und $0 \leq m \leq n-1$
Output : $\pi' = (\pi'_1, \ \dots \ , \ \pi'_{n-m})$

```
1  π' ← π // Kopie!
2  for i ← 1 to m do
3  |  π' ← φ_{λ_i}^{(n)}(π')
4  end for
5  return π'
```

5.1. Stapelsortierung / MWO-Problem (Aufgabenteil a)

Mit Hilfe der vereinfachten Variante der WUE-Operations lassen sich die zuvor definierten Algorithmen noch etwas vereinfachen und die Laufzeit wird sich in der Praxis verbessern:
Der folgende Algorithmus 13 funktioniert genau wir Algorithmus 5 mit dem Unterschied, dass nun die neue Definition $\varphi_i^{(n)}$ der WUE-Operation genutzt wird und das pruefen der Sortiertheit gemassz Definition 43 vereinfacht wurde.

Algorithmus 13 : Vereinfachung des Algorithmus 5 (MWO-Problem)

Input : Permutation $\pi \in S_n$ der Laenge $n \in \mathbb{N}$
Output : $\lambda = (\lambda_1, \ldots, \lambda_m)$ mit $m \in \{0, 1, \ldots n-1\}$

```
1  for m ← 0 to n − 1 do
2      λ_result ← NULL
3      if checkAll(( ), m) then
4          return λ_result
5      end if
6  end for
7  Function checkAll(λ, m):
8      if |λ| = m then
9          σ ← φ_λ^(n)(π)
10         // Pruefe, ob σ sortiert ist
11         if σ = id then
12             λ_result ← λ // Kopie!
13             return TRUE
14         else
15             return FALSE
16         end if
17     end if
18     for i ← 1 to (n − |λ|) do
19         λ.add(i)
20         if checkAll(λ, m) then
21             return TRUE
22         end if
23         λ.removeLastElement() // Remove element that was added last (line 19)
24     end for
25     return FALSE
```

Mit dem gleichen Prinzip wurde Algorithmus 14 basierend auf Algorithmus 6 wie folgt definiert:

Algorithmus 14 : Vereinfachung des Algorithmus 6 (MWO Problem)

Input : Permutation $\pi \in S_n$ der Laenge $n \in \mathbb{N}$
Output : $\lambda = (\lambda_1, \ \dots \ , \ \lambda_m)$ mit $m \in \{0 \ , 1, \ \dots \ n-1\}$

```
1  return Foo(π)
2  Function Foo(σ):
3      // Pruefe, ob σ sortiert ist
4      if σ = id then
5          return ( )
6      end if
7      (λ_result, m) ← (NULL, ∞)
8      for i ← 1 to |σ| do
9          σ' ← φ_i^(|σ|)(σ)
10         λ' ← ( i ).add(Foo(σ'))
11         if |λ'| < m then
12             (λ_result, m) ← (λ', |λ'|)
13         end if
14     end for
15     return λ_result
```

5.2. PWUE-Zahlen / PWZ-Problem (Aufgabenteil b)

Auch die bereits vorgestellten Loesungsvorschlaege fuer das PWZ-Problem lassen sich mit der neuen Definition der WUE-Operation etwas vereinfachen. Auch hier wird sich die praktische Laufzeit deutlich verbessert.

Zunaechst laesst sich Algorithmus 7 mit Hilf der neuen WUE-Operation wie folgt vereinfachen:

Algorithmus 15 : $A(\pi)$ bestimmen - Vereinfachung des Algorithmus 7

Input : $\pi \in S_n$ mit $n \in \mathbb{N}$
Output : $A(\pi)$

```
1  for m ← 0 to n − 1 do
2  |  if getA( ), m, n) then
3  |  |  return m
4  |  end if
5  end for
6  Function getA(λ, m, n):
7  |  if |λ| = m then
8  |  |  σ ← φ_λ^(n)(π)
9  |  |  // Pruefe, ob σ sortiert ist
10 |  |  if σ = id then
11 |  |  |  return TRUE
12 |  |  else
13 |  |  |  return FALSE
14 |  |  end if
15 |  end if
16 |  for i ← 1 to (n − |λ|) do
17 |  |  λ.add(i)
18 |  |  if getA(λ, m, n) then
19 |  |  |  return TRUE
20 |  |  end if
21 |  |  λ.remove(i)
22 |  end for
23 |  return FALSE
```

So wie beim vorherigen Algorithmus, laesst sich auch Algorithmus 8 mit der neu definierten WUE-Operation vereinfachen (siehe Algorithmus 16).

Mit Algorithmus 10 laesst sich nun weiterhin $P(n)$ berechnen, wenn man eine eine der soeben vorgestellten Algortihmen nutzt, um $A(\pi)$ zu bestimmem.

Algorithmus 16 : $A(\pi)$ rekursiv bestimmen - Vereinfachung des Algorithmus 8

Input : $\pi \in S_n$ mit $n \in \mathbb{N}$
Output : $A(\pi)$

```
1  return getA(π)
2  Function getA(τ):
3      // Pruefe, ob τ sortiert ist
4      if τ = id then
5          return 0
6      end if
7      min ← ∞
8      for i ← 1 to |τ| do
9          σ ← φ_i^(|τ|)(τ)
10         A ← getA(σ) + 1
11         min ← min{min, A}
12     end for
13     return min
```

6. Optimierung - Dynamische Programmierung

Die rekursiven Algorithmen 14 (zum loessen des MWO-Problems) und 16 (zum bestimmen von $A(\pi)$ als Bestandteil des PWZ-Problems) lassen sich mit Hilfe der Dynamischen Programmierung stark verbessern. Insbesondere die Laufzeit der Loesung des PWZ-Problems laesst sich dadurch in der Praxis deutlich verbessern.

Dabei ist die Idee, dass man bei einem Problem, welches man in Teilprobleme zerlegen (mit einem kleinsten Problem, Basisfall) und dann rekursiv loesen kann, wobei einige Teilprobleme mehrfach geloest werden muessen[12], einfach die Loesungen der Teilprobleme speichert.

Genau dieses Verfahren kann beim rekursiven loesen des MWO-Problems (Algorithmen 14) und dem rekursiven bestimmen des Wertes $A(\pi)$ (Algorithmus 16) benutzt werden, da die kleinen Instanzen des Problems eindeutig mehrfach bestimmt werden muessen und bei Algorithmus 16 in der Tat alle, mehrfach.

6.1. MWO-Problem

Um Algorithmus 14 mit Hilfe der Dynamischen Programmierung zu verbessern, wird bereits vor dem ersten Aufruf der rekursiven Funktion *Foo* eine Hashtabelle erstellt, die bereits berechneten Rueckgabewerte dieser Funktion speichern wird. Somit kann beim Aufruf der Funktion, nach der Ueberpruefung der Sortiertheit, abgefragt werden, ob die Eingabepermutation σ bereits berechnet wurde (also in der Hashtabelle ist). Ist dies der Fall, so wird der bereits berechnete Wert direkt zurueckgegeben.
Auszerdem muss vor der Rueckgabe des Ergebnis in der Funktion, der Wert, der zurueckgegeben werden wird, in die Hashtabelle eingetragen werden.

Algorithmus 17 : Algorithmus 14 mit DP (MWO-Problem)

Input : Permutation $\pi \in S_n$ der Laenge $n \in \mathbb{N}$
Output : $\lambda = (\lambda_1, \ \dots \ , \ \lambda_m)$ mit $m \in \{0 \ , 1, \ \dots \ n-1\}$

```
1  memo ← [ ] // Hashtabelle
2  return Foo(π)
3  Function Foo(σ):
4      // Pruefe, ob σ sortiert ist
5      if σ = id then
6          return ( )
7      end if
8      if memo.contains(σ) then
9          return memo.get(σ)
10     end if
11     (λ_result, m) ← (NULL, ∞)
12     for i ← 1 to |σ| do
13         σ' ← φ_i^(|σ|)(σ)
14         λ' ← ( i ).add(Foo(σ'))
15         if |λ'| < m then
16             (λ_result, m) ← (λ', |λ'|)
17         end if
18     end for
19     memo.put(σ, λ_result)
20     return λ_result
```

[12] Siehe auch `https://de.wikipedia.org/wiki/Dynamische_Programmierung`; Einfaches Beispie: Die Fibonacci-Folge

6.2. PWZ-Problem

Dasselbe Prinzip wie zuvor wird nun auch auf Algorithmus 16 (Bestimmung von $A(\pi)$) angewendet (siehe Algorithmus 18), sodass Algorithmus 9 (PWZ-Problem) anschlieszend mit Hilfe der Dynamischen Programmierung deutlich verbessert wird (siehe Algorithmus 19).

Algorithmus 18 : $A(\pi)$ rekursiv bestimmen - Algorithmus 18 mit Dynamischer Programmierung

Input : $\pi \in S_n$ mit $n \in \mathbb{N}$
Output : $A(\pi)$

```
1  return getA(π)
2  Function getA(τ):
3      // Pruefe, ob τ sortiert ist
4      if τ = id then
5          return 0
6      end if
7      // Pruefe, ob A(τ) bereits berechnet wurde
8      if memo.contains(σ) then
9          return memo.get(σ)
10     end if
11     min ← ∞
12     for i ← 1 to |τ| do
13         σ ← φ_i^(|τ|)(τ)
14         A ← getA(σ) + 1
15         min ← min{min, A}
16     end for
17     // Trage das Ergebnis in die HashMap ein
18     memo.put(τ, min)
19     return min
```

Algorithmus 19 : Iterativer Algorithmus um $P(n)$ und Bsp. zu bestimmen - mit DP (PWZ-Problem)

Input : $n \in \mathbb{N}$
Output : $P(n)$ und Beispiel $\pi \in S_N$

```
1  π_worst ← NULL
2  max ← 0
3  memo ← [ ] // Hashtabelle
4  for π ∈ S_n do
5      A ← getA(π)
6      if A > max then
7          max ← A
8          π_worst ← π
9      end if
10 end for
11 return (π_worst, max)
```

Ein weiteres Verfahren, das PWZ-Problem zu loesen, besteht darin, fuer jede Groesze der Permutationen j angefangen bei 1 bis zu n alle Werte $A(\pi)$ in einer Hashtabelle zu speichern und mit dessen Hilfe je $A(\pi)$ fuer alle Permutationen, die um eins groeszer sind, zu finden. Diese Ergebnisse werden dann in einer neuen Hashtabelle gepeichert und das Verfahren wird iterativ wiederholt. Bei jedem iterieren durch alle Permutationen wird der (fuer die aktuelle Groesze) groeszte Wert von $A(\pi)$ und ein dazugehoeriges Beispiel gespeichert, sodass dieses am Ende auch fuer die Groesze n der Permutationen berechnet wurde.
Angefangen wird dabei mit der Groesze 1, wofuer nur ein Element manuell zur ersten Hashtabelle hinzugefuegt werden muss (mit dem Wert 0).

Algorithmus 20 : Iterativer Algorithmus um $P(n)$ und Bsp. zu bestimmen - mit DP (PWZ-Problem)

Input : $n \in \mathbb{N}$
Output : $P(n)$ und Beispiel $\pi \in S_N$

```
1  memo ← [ ] // Hashtabelle
2  id_1 ← id ∈ S_1
3  memo.put(id_1, 0)
4  (max_result, example_result) ← (0, NULL)
5  for j ← 2 to n do
6  |  memo_next ← [ ] // Hashtabelle
7  |  max ← 0 example ← 0
8  |  for π ∈ S_j do
9  |  |  A ← getA(π)
10 |  |  if A > max then
11 |  |  end if
12 |  |  memo_next.put(π, A)
13 |  end for
14 |  memo ← memo_next
15 |  (max_result, example_result) ← (max, example)
16 end for
17 return (max_result, example_result)
18 // Algorithmus um A(π) zu bestimmen
19 Function getA(π):
20 |  // Pruefe, ob π sortiert ist
21 |  if π = id then
22 |  |  return 0
23 |  end if
24 |  min ← ∞
25 |  for i ← 1 to |π| do
26 |  |  σ ← φ_i^(|π|)(π)
27 |  |  A ← memo.get(σ)
28 |  |  min ← min{min, A}
29 |  end for
30 |  return min
```

6.3. Hashing von Permutationen

Damit die HashMap richtig funktionieren kann, muss ein Hashverfahren fuer Permutationen gegeben sein, das Permutationen einen moeglichst eindeutigen Wert zuweist.

Das erste Hash-Verfahren ist ein standartmaessiges Verfahren, um Listen an ganzen Zahlen zu hashen:

Algorithmus 21 : Hash Verfahren fuer Permutationen I

Input : $\pi \in S_n$ fuer $n \in \mathbb{N}$
Output : Hash $h \in \mathbb{Z}$

```
1 integer h ← 1
2 for i ← 1 to n do
3 |   h ← 31 · h + π(i)
4 end for
5 return h
```

Da h ein integer ist, ist gewollt, dass ein Integer-Overflow entstehen kann, sodass am Ende

$$-2^{31} \leq h \leq 2^{31} - 1$$

gilt.

Das zweite Hash-Verfahen interpretiert eine Permutation als Zahl zur Basis n und konvertiert diese in eine Dezimmalzahl. Dadurch ist sichergestellt, dass jede Permutation dieser Groesze einen eindeutigen Wert hat.

Algorithmus 22 : Hash Verfahren fuer Permutationen II

Input : $\pi \in S_n$ fuer $n \in \mathbb{N}$
Output : Hash $h \in \mathbb{N}$

```
1 h ← 0
2 for i ← 1 to n do
3 |   h ← h + n^(i−1) · π(i)
4 end for
5 return h
```

Hier wird also

$$h = \sum_{n}^{i=1} n^{i-1} \cdot \pi(i)$$

berechnet.
In der Realitaet, wenn h beispielsweise ein Long in Java (64 Bits) ist, ist dieses Verfahen nur fuer n einer gewissen Maximalgroesze geeignet. Fuer den Fall des Long aus dem Wertebereich $[-2^{63}; 2^{63} - 1]$ ist dieser Bereich ab $n = 16$ ueberschritten.
Denn fuer die Permutationen id $\in S_{15}$ und id $\in S_{16}$ gilt, dass

$$\sum_{i=1}^{n} n^{i-1} \cdot \pi(i) = \sum_{n}^{i=1} n^{i-1} \cdot i$$

und es gilt

$$2^{63} - 1 > \sum_{i=1}^{15} 15^{i-1} \cdot i$$

aber

$$2^{63} - 1 < \sum_{i=1}^{16} 16^{i-1} \cdot i.$$

(siehe Wolframalpha)

7. Implementierung

Nun wird auf die Implementierung der Loesungsideen eingegangen.
Die Loesungsideen wurden alle in einem Java 8 Programm implementiert (sowohl Teilaufgabe (a) als auch Teilaufgabe (b)).

Dabei wurde objektorientiert vorgegangen, sodass eine Instanz einen Problems (MWO-Problem oder PWZ-Problem) je durch eine Instanz einer Klasse geloest wird. Wobei es wiederum zwei abstrakte Klassen *PancakeSort* und *PZSolver* gibt, die als Eltern-Klassen der Klassen fungieren, die die Probleme wirklichen loesen koennen. Dadurch ist das benutzen unterschiedlicher konkreter Implemtierungen einfacher zu handhaben und zu testen.

Die Pfannkuchenstapel wurden in allen Loesungsimplementierungen als Integer-Arrays dargestellt werden, bzw. durch eine Wrapper-Klasse dieser. Diese Wrapper-Klasse *IntArray* fuer ein Integer-Array wird fuer einige Implementierung genutzt, die die Dynamische Programmierung verwenden, da dadurch das Hashing von Integer-Arrays vereinfacht wird. Logischerweise sind die Arrays 0-basiert indexiert, sodass dementspraechend auch die Pfannkuchen so behandelt werden, als waere der erste Pfannkuchen an Index 0, usw. Deswegen sind auch die Ausgaben 0- und nicht 1-basiert indexiert.

Die Algorithmen, die die Dynamische Programmierung verwenden, wurden allesamt mit Hilfe von *HashMaps* realisiert, die entweder von *IntArray* auf *Integer* abbilden, oder von *Long* auf *Integer* (fuer das manuele Hash-Verfahren).

Zudem gibt es einige Funktionen, die von vielen Loesungsimplementierungen benoetigt werden. Diese wurden als oeffentliche, statische Funktionen in die Klasse *Utils* ausgelagert. Beispielsweise, die Funktionen, um die WUE-Operationen anzuwenden, oder zu ueberpruefen, ob ein Pfannkuchenstapel (also in Integer-Array) sortiert ist.

8. Beispiele

8.1. Stapelsortierung / MWO-Problem (Aufgabenteil a)

Nun werden die Ergebnisse der Impementierung der Algorithmen in Java für die Beispieleingaben der BwInf-Website und zwei eigenden, sowie die dazugehoerige Laufzeit auf meinem Rechner vorgestellt. Dafuer werden zunaechst zwei Tabellen als Uebersicht ueber die Ergennisse dargelegt und anschlieszend die konkreten Loesungen. Dazu wurde immer eine mathematische Notation und bei den Beispielen 0 bis 3 der BwInf-Website auch eine bessere, visuelle Darstellung der WUE-Operationen angegeben.
Dabei ist zu beachten, dass im Folgenden eine einsbasierte Indexierung genutzt wird. In den Beispieldatein, die mitgeschickt wurden, ist hingegen eine nullbasierte Indexierung verwendet worden, da das Java-Programm ebenfalls nullbasiert arbeitet.

Beispiel	1	2	3	4	5	6	7	8	9	10	11
Groesze	5	7	8	11	13	14	15	16	14	17	18
Laufzeit	1ms	1ms	1ms	9ms	56ms	109ms	551ms	2,412s	151ms	9,744s	45,501s

Tabelle 1: Uebersicht ueber die Beispieleingaben

B.	n	Eingabe	Ausgabe
1	5	$S = (3, 2, 4, 5, 1)$	$\lambda = (5,\ 4)$
2	7	$S = (6,\ 3,\ 1,\ 7,\ 4,\ 2,\ 5)$	$\lambda = (3,\ 4,\ 5)$
3	8	$S = (8,\ 1,\ 7,\ 5,\ 3,\ 6,\ 4,\ 2)$	$\lambda = (2,\ 3,\ 3,\ 5)$
4	11	$S = (5,\ 10,\ 1,\ 11,\ 4,\ 8,\ 2,\ 9,\ 7,\ 3,\ 6)$	$\lambda = (1,\ 1,\ 7,\ 2,\ 7,\ 3)$
5	13	$S = (7,\ 4,\ 11,\ 5,\ 10,\ 6,\ 1,\ 13,\ 12,\ 9,\ 3,\ 8,\ 2)$	$\lambda = (1,\ 1,\ 2,\ 5,\ 6,\ 8,\ 4)$
6	14	$S = (4,\ 13,\ 10,\ 8,\ 2,\ 3,\ 7,\ 9,\ 14,\ 1,\ 12,\ 6,\ 5,\ 11)$	$\lambda = (1,\ 13,\ 7,\ 9,\ 5,\ 6)$
7	15	$S = (14,\ 8,\ 4,\ 12,\ 13,\ 2,\ 1,\ 15,\ 7,\ 11,\ 3,\ 9,\ 5,\ 10,\ 6)$	$\lambda = (1,\ 2,\ 6,\ 11,\ 2,\ 3,\ 6,\ 8)$
8	16	$S = (8,\ 5,\ 10,\ 15,\ 3,\ 7,\ 13,\ 6,\ 2,\ 4,\ 12,\ 9,\ 1,\ 14,\ 16,\ 11)$	$\lambda = (1,\ 3,\ 8,\ 4,\ 9,\ 11,\ 10,\ 6)$
9	14	$S = (8,\ 13,\ 3,\ 11,\ 1,\ 12,\ 9,\ 7,\ 4,\ 6,\ 14,\ 10,\ 5,\ 2)$	$\lambda = (4,\ 7,\ 3,\ 7,\ 8,\ 9,\ 4)$
10	17	$S = (14,\ 9,\ 13,\ 10,\ 6,\ 2,\ 12,\ 15,\ 3,\ 1,\ 4,\ 16,\ 5,\ 7,\ 11,\ 17,\ 8)$	$\lambda = (2,\ 6,\ 7,\ 8,\ 9,\ 10,\ 11,\ 10)$
11	18	$S = (13,\ 5,\ 9,\ 18,\ 11,\ 1,\ 17,\ 14,\ 4,\ 7,\ 12,\ 10,\ 8,\ 16,\ 6,\ 15,\ 3,\ 2)$	$\lambda = (1,\ 2,\ 14,\ 12,\ 7,\ 9,\ 7,\ 3,\ 10)$

Tabelle 2: Uebersicht ueber die folgenden Ergebnisse

8.1.1 Beispiel 1 - „pancake0.txt“ (Groesze 5)

Der Pfannkuchenstapel

$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 3 & 2 & 4 & 5 & 1 \end{pmatrix}$$

kann durch Anwendung der WUE-Operationsfolge $\lambda := (5,\ 4)$ wie folgt in einen sortierten Stapel ueberfuehrt werden:

$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 3 & 2 & 4 & 5 & 1 \end{pmatrix} \xrightarrow{5} \begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 4 & 2 & 3 \end{pmatrix} \xrightarrow{4} \begin{pmatrix} 1 & 2 & 3 \\ 2 & 4 & 5 \end{pmatrix}$$

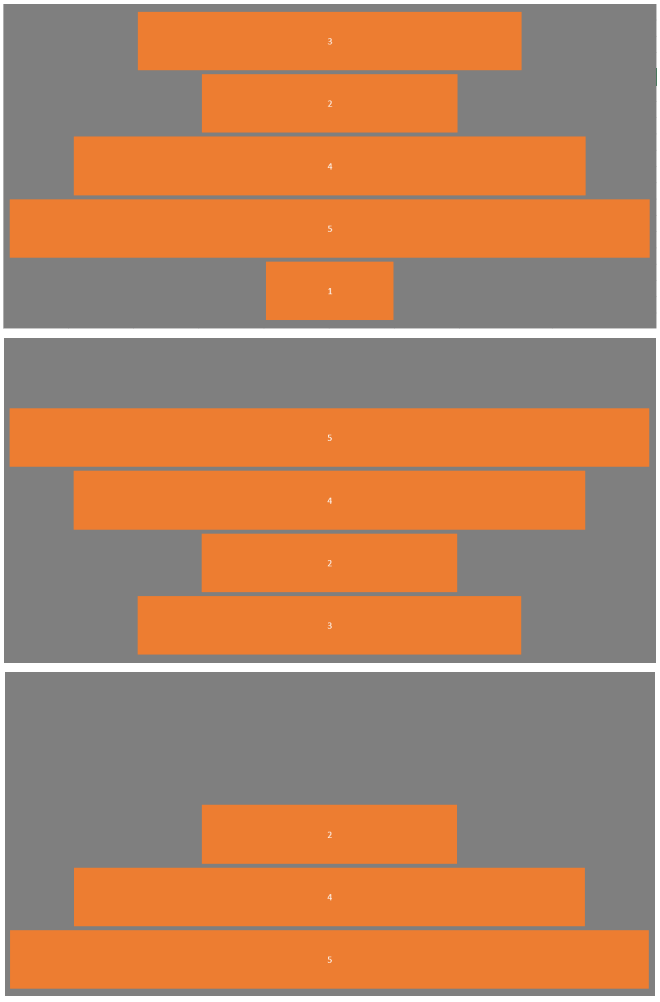
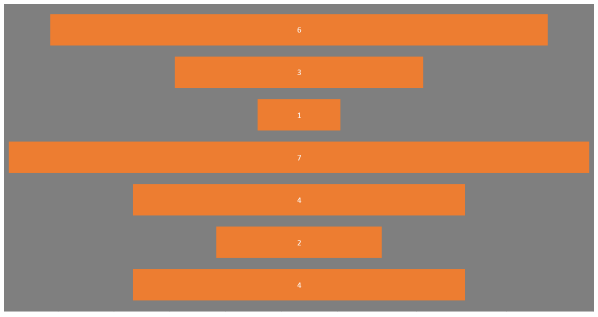


Abbildung 1: Keine Operationen, 1. Operation, 2. Operation

8.1.2 Beispiel 2 - „pancake1.txt“ (Groesze 7)

Der Pfannkuchenstapel

$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ 6 & 3 & 1 & 7 & 4 & 2 & 5 \end{pmatrix}$$



kann durch Anwendung der WUE-Operationsfolge $\lambda := (3,\ 4,\ 5)$ wie folgt in einen sortierten Stapel ueberfuehrt werden (Berechnungszeit 1ms):

$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ 6 & 3 & 1 & 7 & 4 & 2 & 5 \end{pmatrix} \xrightarrow{3} \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 3 & 6 & 7 & 4 & 2 & 5 \end{pmatrix}$$

$$\xrightarrow{4} \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 7 & 6 & 3 & 2 & 5 \end{pmatrix}$$

$$\xrightarrow{5} \begin{pmatrix} 1 & 2 & 3 & 4 \\ 2 & 3 & 6 & 7 \end{pmatrix}$$

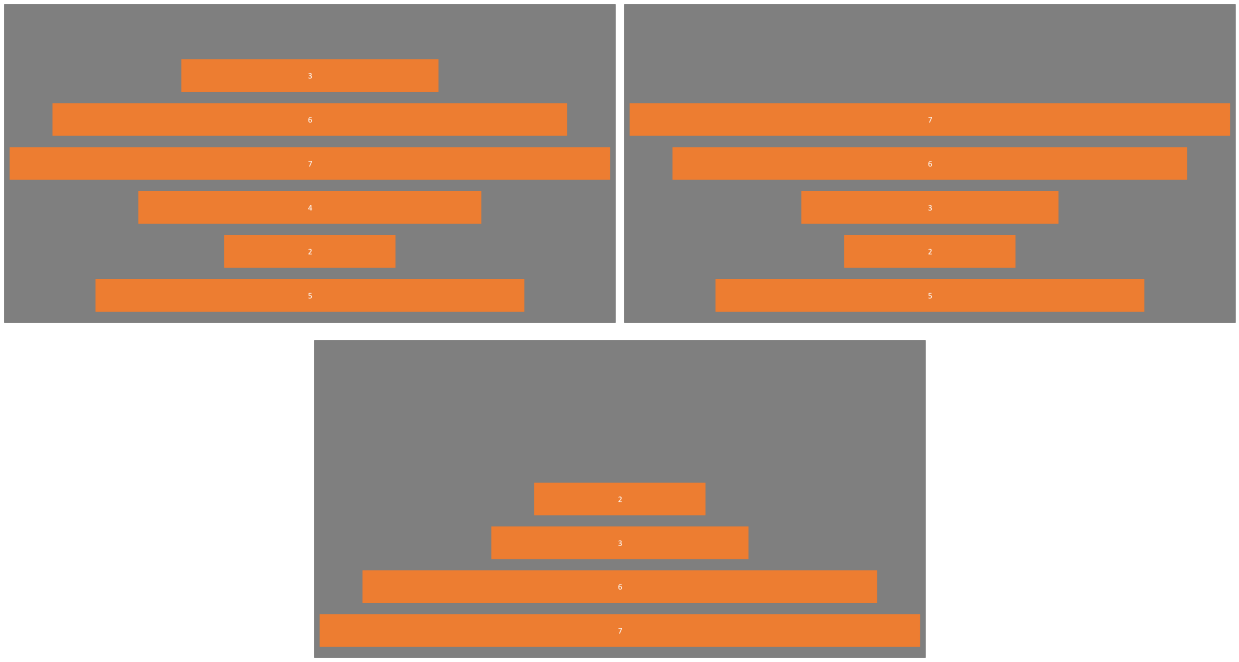
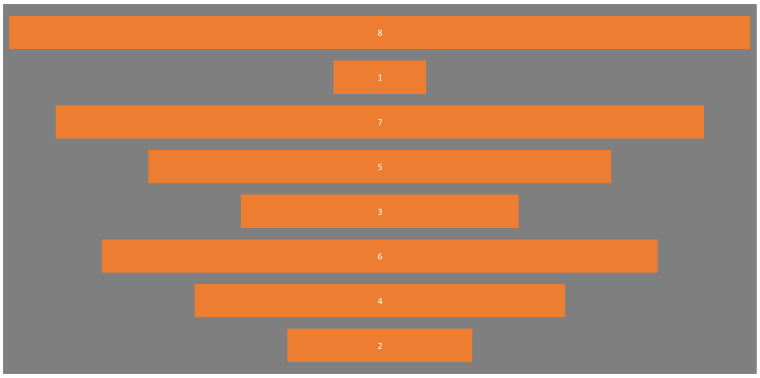


Abbildung 2: 1. Operationen, 2. Operation, 3. Operation

8.1.3 Beispiel 3 - „pancake2.txt“ (Groesze 8)

Der Pfannkuchenstapel

$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ 8 & 1 & 7 & 5 & 3 & 6 & 4 & 2 \end{pmatrix}$$



kann durch Anwendung der WUE-Operationsfolge $\lambda := (2,\ 3,\ 3,\ 5)$ wie folgt in einen sortierten Stapel ueberfuehrt werden:

$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ 8 & 1 & 7 & 5 & 3 & 6 & 4 & 2 \end{pmatrix} \xrightarrow{2} \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ 8 & 7 & 5 & 3 & 6 & 4 & 2 \end{pmatrix}$$
$$\xrightarrow{3} \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 7 & 8 & 3 & 6 & 4 & 2 \end{pmatrix}$$
$$\xrightarrow{3} \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 8 & 7 & 6 & 4 & 2 \end{pmatrix}$$
$$\xrightarrow{5} \begin{pmatrix} 1 & 2 & 3 & 4 \\ 4 & 6 & 7 & 8 \end{pmatrix}$$

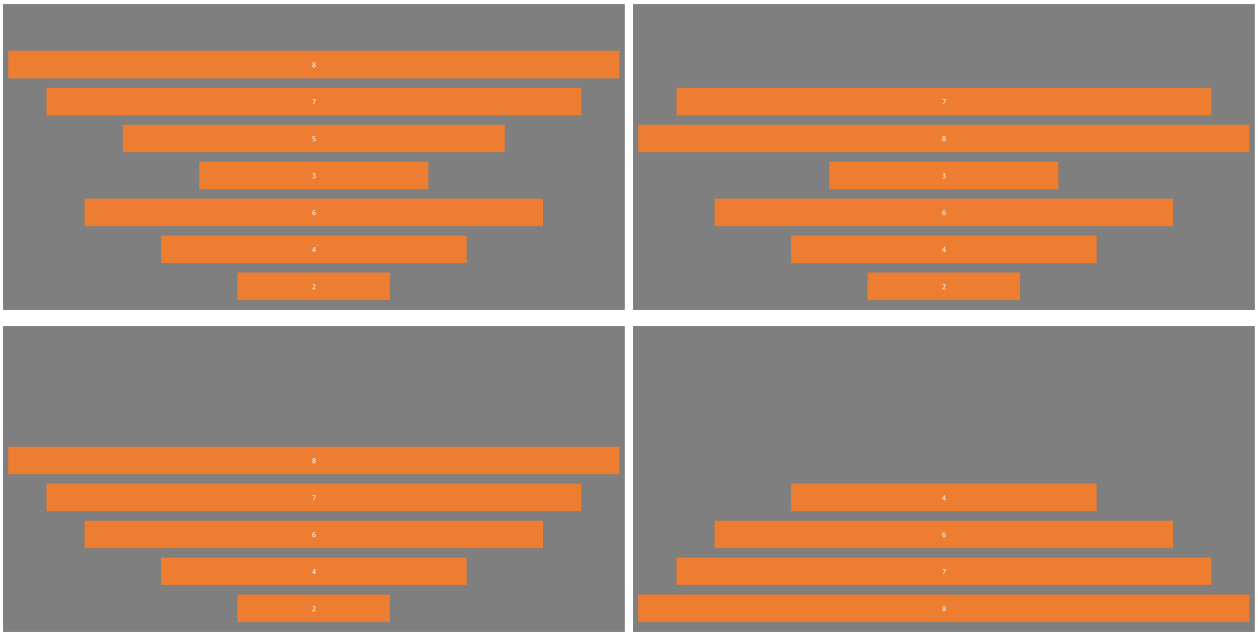
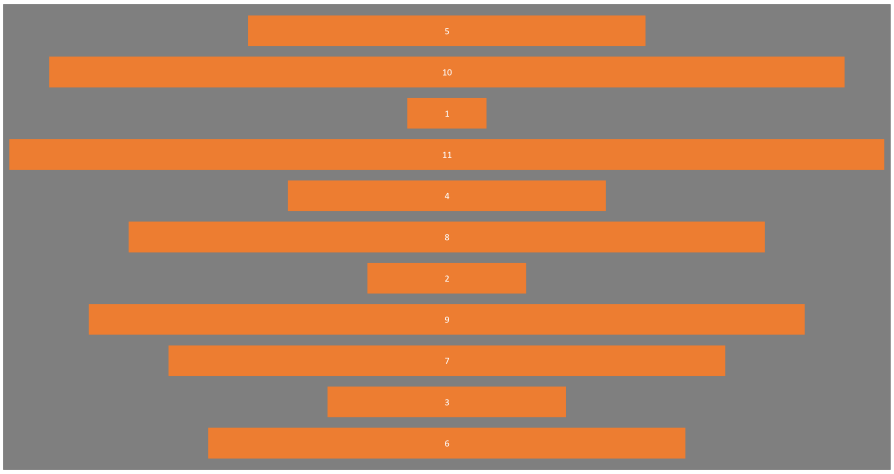


Abbildung 3

8.1.4 Beispiel 4 - „pancake3.txt“ (Groesze 11)

Der Pfannkuchenstapel

$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 \\ 5 & 10 & 1 & 11 & 4 & 8 & 2 & 9 & 7 & 3 & 6 \end{pmatrix}$$



kann durch Anwendung der WUE-Operationsfolge $\lambda := (1,\ 1,\ 7,\ 2,\ 7,\ 3)$ wie folgt in einen sortierten Stapel ueberfuehrt werden:

$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 \\ 5 & 10 & 1 & 11 & 4 & 8 & 2 & 9 & 7 & 3 & 6 \end{pmatrix} \xrightarrow{1} \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\ 10 & 1 & 11 & 4 & 8 & 2 & 9 & 7 & 3 & 6 \end{pmatrix}$$
$$\xrightarrow{1} \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\ 1 & 11 & 4 & 8 & 2 & 9 & 7 & 3 & 6 \end{pmatrix}$$
$$\xrightarrow{7} \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ 9 & 2 & 8 & 4 & 11 & 1 & 3 & 6 \end{pmatrix}$$
$$\xrightarrow{2} \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ 9 & 8 & 4 & 11 & 1 & 3 & 6 \end{pmatrix}$$
$$\xrightarrow{7} \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 3 & 1 & 11 & 4 & 8 & 9 \end{pmatrix}$$
$$\xrightarrow{3} \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 1 & 3 & 4 & 8 & 9 \end{pmatrix}$$

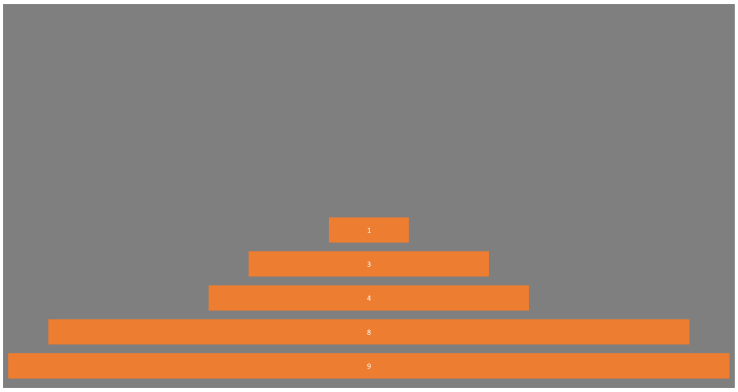


Abbildung 4: Ergebnis der Operationsfolge

8.1.5 Beispiel 5 - „pancake4.txt“ (Groesze 13)

Der Pfannkuchenstapel

$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 \\ 7 & 4 & 11 & 5 & 10 & 6 & 1 & 13 & 12 & 9 & 3 & 8 & 2 \end{pmatrix}$$

kann durch Anwendung der WUE-Operationsfolge $\lambda := (1,\ 1,\ 2,\ 5,\ 6,\ 8,\ 4)$ wie folgt in einen sortierten Stapel ueberfuehrt werden:

$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 \\ 7 & 4 & 11 & 5 & 10 & 6 & 1 & 13 & 12 & 9 & 3 & 8 & 2 \end{pmatrix} \xrightarrow{1} \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 \\ 4 & 11 & 5 & 10 & 6 & 1 & 13 & 12 & 9 & 3 & 8 & 2 \end{pmatrix}$$
$$\xrightarrow{1} \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 \\ 11 & 5 & 10 & 6 & 1 & 13 & 12 & 9 & 3 & 8 & 2 \end{pmatrix}$$
$$\xrightarrow{2} \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\ 11 & 10 & 6 & 1 & 13 & 12 & 9 & 3 & 8 & 2 \end{pmatrix}$$
$$\xrightarrow{5} \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\ 1 & 6 & 10 & 11 & 12 & 9 & 3 & 8 & 2 \end{pmatrix}$$
$$\xrightarrow{6} \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ 12 & 11 & 10 & 6 & 1 & 3 & 8 & 2 \end{pmatrix}$$
$$\xrightarrow{8} \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ 8 & 3 & 1 & 6 & 10 & 11 & 12 \end{pmatrix}$$
$$\xrightarrow{4} \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 1 & 3 & 8 & 10 & 11 & 12 \end{pmatrix}$$

8.1.6 Beispiel 6 - „pancake5.txt“ (Groesze 14)

Der Pfannkuchenstapel

$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 \\ 4 & 13 & 10 & 8 & 2 & 3 & 7 & 9 & 14 & 1 & 12 & 6 & 5 & 11 \end{pmatrix}$$

kann durch Anwendung der WUE-Operationsfolge $\lambda := (1,\ 13,\ 7,\ 9,\ 5,\ 6)$ wie folgt in einen sortierten Stapel ueberfuehrt werden:

$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 \\ 4 & 13 & 10 & 8 & 2 & 3 & 7 & 9 & 14 & 1 & 12 & 6 & 5 & 11 \end{pmatrix} \xrightarrow{1} \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 \\ 13 & 10 & 8 & 2 & 3 & 7 & 9 & 14 & 1 & 12 & 6 & 5 & 11 \end{pmatrix}$$
$$\xrightarrow{13} \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 \\ 5 & 6 & 12 & 1 & 14 & 9 & 7 & 3 & 2 & 8 & 10 & 13 \end{pmatrix}$$
$$\xrightarrow{7} \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 \\ 9 & 14 & 1 & 12 & 6 & 5 & 3 & 2 & 8 & 10 & 13 \end{pmatrix}$$
$$\xrightarrow{9} \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\ 2 & 3 & 5 & 6 & 12 & 1 & 14 & 9 & 10 & 13 \end{pmatrix}$$
$$\xrightarrow{5} \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\ 6 & 5 & 3 & 2 & 1 & 14 & 9 & 10 & 13 \end{pmatrix}$$
$$\xrightarrow{6} \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ 1 & 2 & 3 & 5 & 6 & 9 & 10 & 13 \end{pmatrix}$$

8.1.7 Beispiel 7 - „pancake6.txt“ (Groesze 15)

Der Pfannkuchenstapel

$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 \\ 14 & 8 & 4 & 12 & 13 & 2 & 1 & 15 & 7 & 11 & 3 & 9 & 5 & 10 & 6 \end{pmatrix}$$

kann durch Anwendung der WUE-Operationsfolge $\lambda := (1,\ 2,\ 6,\ 11,\ 2,\ 3,\ 6,\ 8)$ wie folgt in einen sortierten Stapel ueberfuehrt werden:

$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 \\ 14 & 8 & 4 & 12 & 13 & 2 & 1 & 15 & 7 & 11 & 3 & 9 & 5 & 10 & 6 \end{pmatrix}$$

$$\xrightarrow{1} \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 \\ 8 & 4 & 12 & 13 & 2 & 1 & 15 & 7 & 11 & 3 & 9 & 5 & 10 & 6 \end{pmatrix}$$

$$\xrightarrow{2} \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 \\ 8 & 12 & 13 & 2 & 1 & 15 & 7 & 11 & 3 & 9 & 5 & 10 & 6 \end{pmatrix}$$

$$\xrightarrow{6} \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 \\ 1 & 2 & 13 & 12 & 8 & 7 & 11 & 3 & 9 & 5 & 10 & 6 \end{pmatrix}$$

$$\xrightarrow{11} \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 \\ 5 & 9 & 3 & 11 & 7 & 8 & 12 & 13 & 2 & 1 & 6 \end{pmatrix}$$

$$\xrightarrow{2} \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\ 5 & 3 & 11 & 7 & 8 & 12 & 13 & 2 & 1 & 6 \end{pmatrix}$$

$$\xrightarrow{3} \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\ 3 & 5 & 7 & 8 & 12 & 13 & 2 & 1 & 6 \end{pmatrix}$$

$$\xrightarrow{6} \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ 12 & 8 & 7 & 5 & 3 & 2 & 1 & 6 \end{pmatrix}$$

$$\xrightarrow{8} \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ 1 & 2 & 3 & 5 & 7 & 8 & 12 \end{pmatrix}$$

8.1.8 Beispiel 8 - „pancake7.txt“ (Groesze 16)

Der Pfannkuchenstapel

$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 & 16 \\ 8 & 5 & 10 & 15 & 3 & 7 & 13 & 6 & 2 & 4 & 12 & 9 & 1 & 14 & 16 & 11 \end{pmatrix}$$

kann durch Anwendung der WUE-Operationsfolge $\lambda := (1,\ 3,\ 8,\ 4,\ 9,\ 11,\ 10,\ 6)$ wie folgt in einen sortierten Stapel ueberfuehrt werden:

$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 & 16 \\ 8 & 5 & 10 & 15 & 3 & 7 & 13 & 6 & 2 & 4 & 12 & 9 & 1 & 14 & 16 & 11 \end{pmatrix}$$

$$\xrightarrow{1} \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 \\ 5 & 10 & 15 & 3 & 7 & 13 & 6 & 2 & 4 & 12 & 9 & 1 & 14 & 16 & 11 \end{pmatrix}$$

$$\xrightarrow{3} \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 \\ 10 & 5 & 3 & 7 & 13 & 6 & 2 & 4 & 12 & 9 & 1 & 14 & 16 & 11 \end{pmatrix}$$

$$\xrightarrow{8} \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 \\ 2 & 6 & 13 & 7 & 3 & 5 & 10 & 12 & 9 & 1 & 14 & 16 & 11 \end{pmatrix}$$

$$\xrightarrow{4} \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 \\ 13 & 6 & 2 & 3 & 5 & 10 & 12 & 9 & 1 & 14 & 16 & 11 \end{pmatrix}$$

$$\xrightarrow{9} \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 \\ 9 & 12 & 10 & 5 & 3 & 2 & 6 & 13 & 14 & 16 & 11 \end{pmatrix}$$

$$\xrightarrow{11} \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\ 16 & 14 & 13 & 6 & 2 & 3 & 5 & 10 & 12 & 9 \end{pmatrix}$$

$$\xrightarrow{10} \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\ 12 & 10 & 5 & 3 & 2 & 6 & 13 & 14 & 16 \end{pmatrix}$$

$$\xrightarrow{6} \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ 2 & 3 & 5 & 10 & 12 & 13 & 14 & 16 \end{pmatrix}$$

8.1.9 Beispiel 9 - Eigenes Beispiel I (Groesze 14)

Der Pfannkuchenstapel

$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 \\ 8 & 13 & 3 & 11 & 1 & 12 & 9 & 7 & 4 & 6 & 14 & 10 & 5 & 2 \end{pmatrix}$$

kann durch Anwendung der WUE-Operationsfolge $\lambda := (4,\ 7,\ 3,\ 7,\ 8,\ 9,\ 4)$ wie folgt in einen sortierten Stapel ueberfuehrt werden:

$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 \\ 8 & 13 & 3 & 11 & 1 & 12 & 9 & 7 & 4 & 6 & 14 & 10 & 5 & 2 \end{pmatrix}$$
$$\xrightarrow{4} \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 \\ 3 & 13 & 8 & 1 & 12 & 9 & 7 & 4 & 6 & 14 & 10 & 5 & 2 \end{pmatrix}$$
$$\xrightarrow{7} \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 \\ 9 & 12 & 1 & 8 & 13 & 3 & 4 & 6 & 14 & 10 & 5 & 2 \end{pmatrix}$$
$$\xrightarrow{3} \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 \\ 12 & 9 & 8 & 13 & 3 & 4 & 6 & 14 & 10 & 5 & 2 \end{pmatrix}$$
$$\xrightarrow{7} \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\ 4 & 3 & 13 & 8 & 9 & 12 & 14 & 10 & 5 & 2 \end{pmatrix}$$
$$\xrightarrow{8} \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\ 14 & 12 & 9 & 8 & 13 & 3 & 4 & 5 & 2 \end{pmatrix}$$
$$\xrightarrow{9} \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ 5 & 4 & 3 & 13 & 8 & 9 & 12 & 14 \end{pmatrix}$$
$$\xrightarrow{4} \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ 3 & 4 & 5 & 8 & 9 & 12 & 14 \end{pmatrix}$$

8.1.10 Beispiel 10 - Eigenes Beispiel II (Groesze 17)

Der Pfannkuchenstapel

$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 & 16 & 17 \\ 14 & 9 & 13 & 10 & 6 & 2 & 12 & 15 & 3 & 1 & 4 & 16 & 5 & 7 & 11 & 17 & 8 \end{pmatrix}$$

kann durch Anwendung der WUE-Operationsfolge $\lambda := (2,\ 6,\ 7,\ 8,\ 9,\ 10,\ 11,\ 10)$ wie folgt in einen sortierten Stapel ueberfuehrt werden:

$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 & 16 & 17 \\ 14 & 9 & 13 & 10 & 6 & 2 & 12 & 15 & 3 & 1 & 4 & 16 & 5 & 7 & 11 & 17 & 8 \end{pmatrix}$$
$$\xrightarrow{2} \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 & 16 \\ 14 & 13 & 10 & 6 & 2 & 12 & 15 & 3 & 1 & 4 & 16 & 5 & 7 & 11 & 17 & 8 \end{pmatrix}$$
$$\xrightarrow{6} \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 \\ 2 & 6 & 10 & 13 & 14 & 15 & 3 & 1 & 4 & 16 & 5 & 7 & 11 & 17 & 8 \end{pmatrix}$$
$$\xrightarrow{7} \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 \\ 15 & 14 & 13 & 10 & 6 & 2 & 1 & 4 & 16 & 5 & 7 & 11 & 17 & 8 \end{pmatrix}$$
$$\xrightarrow{8} \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 \\ 1 & 2 & 6 & 10 & 13 & 14 & 15 & 16 & 5 & 7 & 11 & 17 & 8 \end{pmatrix}$$
$$\xrightarrow{9} \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 \\ 16 & 15 & 14 & 13 & 10 & 6 & 2 & 1 & 7 & 11 & 17 & 8 \end{pmatrix}$$
$$\xrightarrow{10} \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 \\ 7 & 1 & 2 & 6 & 10 & 13 & 14 & 15 & 16 & 17 & 8 \end{pmatrix}$$
$$\xrightarrow{11} \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\ 17 & 16 & 15 & 14 & 13 & 10 & 6 & 2 & 1 & 7 \end{pmatrix}$$
$$\xrightarrow{10} \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\ 1 & 2 & 6 & 10 & 13 & 14 & 15 & 16 & 17 \end{pmatrix}$$

8.1.11 Beispiel 11 - Eigenes Beispiel III (Groesze 18)

Der Pfannkuchenstapel

$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 & 16 & 17 & 18 \\ 13 & 5 & 9 & 18 & 11 & 1 & 17 & 14 & 4 & 7 & 12 & 10 & 8 & 16 & 6 & 15 & 3 & 2 \end{pmatrix}$$

kann durch Anwendung der WUE-Operationsfolge $\lambda := (1,\ 2,\ 14,\ 12,\ 7,\ 9,\ 7,\ 3,\ 10)$ wie folgt in einen sortierten Stapel ueberfuehrt werden:

$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 & 16 & 17 & 18 \\ 13 & 5 & 9 & 18 & 11 & 1 & 17 & 14 & 4 & 7 & 12 & 10 & 8 & 16 & 6 & 15 & 3 & 2 \end{pmatrix}$$

$$\xrightarrow{1} \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 & 16 & 17 \\ 5 & 9 & 18 & 11 & 1 & 17 & 14 & 4 & 7 & 12 & 10 & 8 & 16 & 6 & 15 & 3 & 2 \end{pmatrix}$$

$$\xrightarrow{2} \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 & 16 \\ 5 & 18 & 11 & 1 & 17 & 14 & 4 & 7 & 12 & 10 & 8 & 16 & 6 & 15 & 3 & 2 \end{pmatrix}$$

$$\xrightarrow{14} \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 \\ 6 & 16 & 8 & 10 & 12 & 7 & 4 & 14 & 17 & 1 & 11 & 18 & 5 & 3 & 2 \end{pmatrix}$$

$$\xrightarrow{12} \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 \\ 11 & 1 & 17 & 14 & 4 & 7 & 12 & 10 & 8 & 16 & 6 & 5 & 3 & 2 \end{pmatrix}$$

$$\xrightarrow{7} \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 \\ 7 & 4 & 14 & 17 & 1 & 11 & 10 & 8 & 16 & 6 & 5 & 3 & 2 \end{pmatrix}$$

$$\xrightarrow{9} \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 \\ 8 & 10 & 11 & 1 & 17 & 14 & 4 & 7 & 6 & 5 & 3 & 2 \end{pmatrix}$$

$$\xrightarrow{7} \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 \\ 14 & 17 & 1 & 11 & 10 & 8 & 7 & 6 & 5 & 3 & 2 \end{pmatrix}$$

$$\xrightarrow{3} \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\ 17 & 14 & 11 & 10 & 8 & 7 & 6 & 5 & 3 & 2 \end{pmatrix}$$

$$\xrightarrow{10} \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\ 3 & 5 & 6 & 7 & 8 & 10 & 11 & 14 & 17 \end{pmatrix}$$

8.2. PWUE-Zahlen / PWZ-Problem (Aufgabenteil b)

In diesem Kapitel werden zunaechst einige Grafiken bezgl. der PWUE-Zahlen von $P(1)$ bis $P(12)$ gezeigt. Dabei werden zuerst die eigentliche PWUE-Zahlen, sowie deren Berechnungszeit auf meinem Rechner in einer Tabelle angegeben. Zusaetzlich wird der Wert $n - P(n)$ dargestellt. Dieser entspricht der minimalen Groesze einen Pfannkuchenstapels der Groesze n nachdem er durch eine minimale Anzahl an WUE-Operationen sortiert wurde. Diese PWUE-Zahlen werden ebenfalls in einem Balkendiagramm gezeigt. Anschlieszend wird genauer auf die Verteilung der PWUE-Zahlen eingegangen. In dem Sinne, dass gezeigt wird, wie viele Pfannkuchenstapel einer konkreten Groesze wie viele WUE-Operationen mindestens(!) benoetigen, um in einen sortierten Stapel ueberfuehrt zu werden.
Zuletzt wird fuer jede PWUE-Zahl von $P(1)$ bis $P(12)$ ein konkretes Beispiel gezeigt, welches tatsaechlich mindestens diese Anzahl an WUE-Operationen benoetigt.

n	1	2	3	4	5	6	7	8	9	10	11	12
P(n)	0	1	2	2	3	3	4	5	5	6	6	7
Laufzeit	1ms	1ms	1ms	1ms	1ms	2ms	7ms	39ms	254ms	4.601ms	74,4s	16min
$n - P(n)$	1	1	1	2	2	3	3	3	4	4	5	5

Tabelle 3: Uebersicht ueber die PWUE-Zahlen

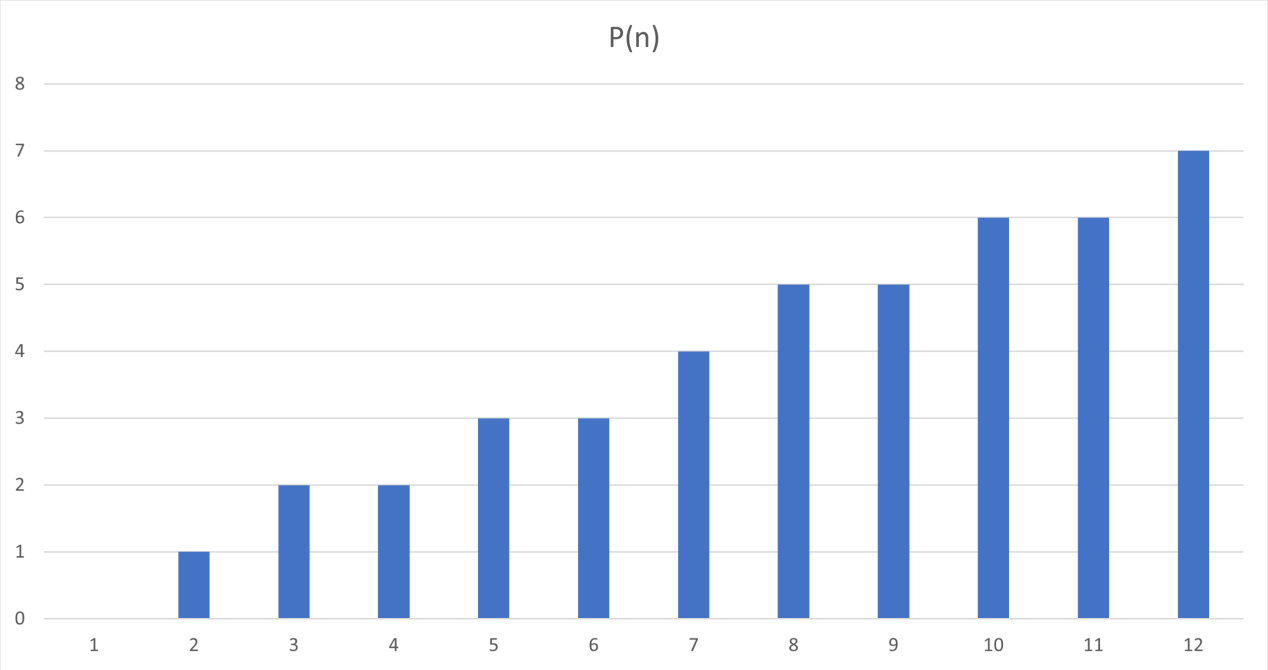


Abbildung 5: Uebersicht ueber die PWUE-Zahlen

Auszerdem folgt aus den Beispielen fuer $n \in \{13,\ 14,\ 15,\ 16,\ 17,\ 18\}$, dass $P(n)$ je einen bestimmten Mindestwert hat:

n	13	14	15	16	17	18
$P(n)$ mind.	7	7	8	8	8	9
Vergl. Bsp.	5	9	7	8	10	11

Tabelle 4: Uebersicht ueber die PWUE-Zahlen II

In Abbildung 6 ist fuer jedes $n = 1, 2, \ldots, 12$ angegeben, wie viele Pfannkuchenstapel (Permutationen der Groesze n) wie viele WUE-Operationen brauchen (mindestens!), um in einen sortierten Stapel ueberfuehrt zu werden. Dadurch laesst sich jeweils direkt die PWUE-Zahl $P(n)$ ablesen, indem man in der Zeile n die groeszte Anzahl an Operationen fuer irgendeinen Pfannkuchen abliesst.

n \ k	0 Operationen	1 Operation	2 Operationen	3 Operationen	4 Operationen	5 Operationen	6 Operationen	7 Operationen	8 Operationen	9 Operationen	10 Operationen	11 Operationen
1	1											
2	1	1										
3	1	4	1									
4	1	10	13	0								
5	1	18	87	14	0							
6	1	28	332	359	0	0						
7	1	40	851	3721	427	0	0					
8	1	54	1774	19816	18645	30	0	0				
9	1	70	3257	67716	262745	29091	0	0	0			
10	1	88	5480	179633	1859546	1581505	2547	0	0	0		
11	1	108	8647	406321	8176623	28346544	2978556	0	0	0	0	
12	1	130	12986	822636	26838446	254035998	196940779	350624	0	0	0	0

Abbildung 6: Uebersicht ueber die Anzahl an Stapeln pro Anzahl an noetigen WUE-Operationen

Weiter ist in Abbildung 7 dargestellt, wie sich die Laufzeich der Berechnung der PWUE-Zahlen (auf meinem Rechner) in Millisekunden verhaelt (vergleiche Tabelle 3). Dabei wurde eine logarithmische Darstellung (Basis 10) gewaehlt. Dort ist in orange die Laufzeit der Tabelle 3 in Millisekunden zu sehen und in grau die Grade $x \mapsto 10^{x-5}$. Da der Graph der Laufzeit ab ab $n = 5$ nahe zu linear verlauft, ist klar erkennbar, dass die Berechnung der PWUE-Zahlen eine exponentielle Laufzeit benoetigt hat.

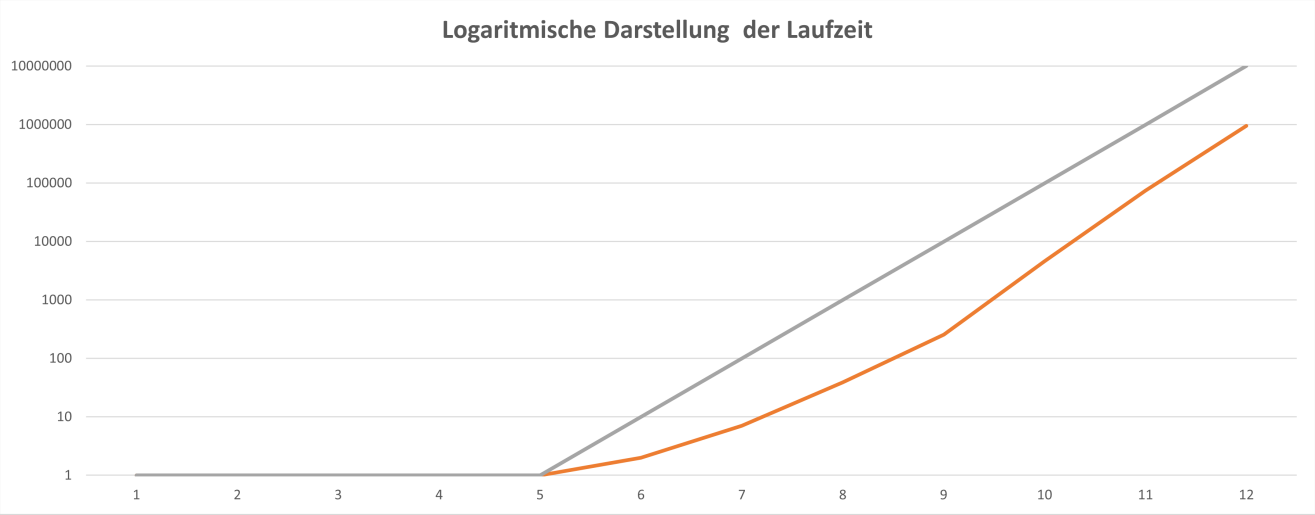


Abbildung 7: Logarithmische Darstellung der Laufzeit der Berechnung von $P(n)$

Im Folgenden wird nun fuer alle PWUE-Zahlen $P(n)$ fuer $n = 1, 2, \ldots, 12$ ein konkretes Beispiel angegeben, welches die angegebene Anzahl $P(n)$ an WUE-Operationen mindestens benoetigt, um in einen sortierten Stapel ueberfuehrt zu werden. Eine Operationsfolge, die den Beispiel-Stapel in einen sortierten ueberfuehrt wird jeweils angegeben und in mathamtischer Schreibweise durchgefuehrt. Dabei ist erneut zu beachten, dass die Indizes immer einsbasiert sind. D.h., der erste Pfannkuchen hat den Index 1, usw. - so, wie es die zweizeilen Schreibweise auch nahe legt.

1. $$P(1) = 0$$

   Dazugehoeriges Beispiel:

   $$\pi = \begin{pmatrix} 1 \\ 1 \end{pmatrix} \in S_1$$

   Offensichtlich existiert nur dieser Stapel der Groesze 1 und dieser ist bereits sortiert.

2. $$P(2) = 1$$

   Dazugehoeriges Beispiel:

   $$\pi = \begin{pmatrix} 1 & 2 \\ 2 & 1 \end{pmatrix} \in S_2$$

   Offensichtlich existieren nur zwei moegliche Stapel von denen nur dieser unsortiert ist, und durch eine beliebige WUE-Operation in einen sortierten ueberfuehrt werden kann. Zum Beispiel am Index 1:

   $$\begin{pmatrix} 1 & 2 \\ 2 & 1 \end{pmatrix} \xrightarrow{1} \begin{pmatrix} 1 \\ 1 \end{pmatrix}$$

3. $$P(3) = 2$$

   Dazugehoeriges Beispiel:

   $$\pi = \begin{pmatrix} 1 & 2 & 3 \\ 2 & 3 & 1 \end{pmatrix} \in S_3$$

   Dieser Stapel benoetigt mind. 2 WUE-Operationen, um in einen sortierten Stapel ueberfuehrt zu werden. Denn durch eine einzelne koennen nur

   $$\begin{pmatrix} 1 & 2 \\ 3 & 1 \end{pmatrix}, \begin{pmatrix} 1 & 2 \\ 2 & 1 \end{pmatrix}, \text{ oder } \begin{pmatrix} 1 & 2 \\ 3 & 2 \end{pmatrix}$$

   entstehen. Und durch eine beliebige, weitere entsteht ein stets sortierter Stapel der Groesze 1. Beispielsweise durch die Operationsfolge $\lambda = (1, 1)$:

   $$\begin{pmatrix} 1 & 2 & 3 \\ 2 & 3 & 1 \end{pmatrix} \xrightarrow{1} \begin{pmatrix} 1 & 2 \\ 3 & 1 \end{pmatrix} \xrightarrow{1} \begin{pmatrix} 1 \\ 1 \end{pmatrix}$$

4. $$P(4) = 2$$

   Dazugehoeriges Beispiel (vergleiche mit Beispiel 3!):

   $$\pi = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 2 & 3 & 1 & 4 \end{pmatrix} \in S_4$$

   Fuer diesen Stapel sind auch mindestens 2 WUE-Operationen notwendig, um ihn einen sortierten Stapel zu ueberfuehren. Dies kann wieder durch die Operationsfolge $\lambda = (1, 1)$ geschehen:

   $$\begin{pmatrix} 1 & 2 & 3 & 4 \\ 2 & 3 & 1 & 4 \end{pmatrix} \xrightarrow{1} \begin{pmatrix} 1 & 2 & 3 \\ 3 & 1 & 4 \end{pmatrix} \xrightarrow{1} \begin{pmatrix} 1 & 2 \\ 1 & 4 \end{pmatrix}$$

5. $$P(5) = 3$$

   Dazugehoeriges Beispiel:

   $$\pi = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 3 & 1 & 5 & 4 & 2 \end{pmatrix} \in S_5$$

   Dieser Stapel benoetigt mindestens 3 WUE-Operationen, um in einen sortierten Stapel ueberfuehrt zu werden. Beispielsweise durch die Operationsfolge $\lambda = (1, 1, 3)$:

   $$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 3 & 1 & 5 & 4 & 2 \end{pmatrix} \xrightarrow{1} \begin{pmatrix} 1 & 2 & 3 & 4 \\ 1 & 5 & 4 & 2 \end{pmatrix} \xrightarrow{1} \begin{pmatrix} 1 & 2 & 3 \\ 5 & 4 & 2 \end{pmatrix} \xrightarrow{3} \begin{pmatrix} 1 & 2 \\ 4 & 5 \end{pmatrix}$$

6.

$$P(6) = 3$$

Dazugehoeriges Beispiel (vergleiche mit Beispiel 5!):

$$\pi = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 3 & 1 & 5 & 4 & 2 & 6 \end{pmatrix} \in S_6$$

Dieser Stapel benoetigt ebenfalls mindestens 3 WUE-Operationen, um in einen sortierten Stapel ueberfuehrt zu werden. Dies kann, wie in Beispiel 5, durch die Operationsfolge $\lambda = (1,\ 1,\ 3)$ geschehen:

$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 3 & 1 & 5 & 4 & 2 & 6 \end{pmatrix} \xrightarrow{1} \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 1 & 5 & 4 & 2 & 6 \end{pmatrix} \xrightarrow{1} \begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 4 & 2 & 6 \end{pmatrix} \xrightarrow{3} \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$$

7.

$$P(7) = 4$$

Dazugehoeriges Beispiel:

$$\pi = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ 5 & 2 & 7 & 3 & 6 & 1 & 4 \end{pmatrix} \in S_7$$

Dieser Stapel benoetigt mindestens 4 WUE-Operationen, um in einen sortierten Stapel ueberfuehrt zu werden. Beispielsweise durch die Operationsfolge $\lambda = (1,\ 1,\ 2,\ 4)$:

$$\begin{aligned} \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ 5 & 2 & 7 & 3 & 6 & 1 & 4 \end{pmatrix} &\xrightarrow{1} \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 2 & 7 & 3 & 6 & 1 & 4 \end{pmatrix} \\ &\xrightarrow{1} \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 7 & 3 & 6 & 1 & 4 \end{pmatrix} \\ &\xrightarrow{2} \begin{pmatrix} 1 & 2 & 3 & 4 \\ 7 & 6 & 1 & 4 \end{pmatrix} \\ &\xrightarrow{4} \begin{pmatrix} 1 & 2 & 3 \\ 1 & 6 & 7 \end{pmatrix} \end{aligned}$$

8.

$$P(8) = 5$$

Dazugehoeriges Beispiel:

$$\pi = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ 6 & 1 & 8 & 5 & 3 & 7 & 4 & 2 \end{pmatrix} \in S_8$$

Dieser Stapel benoetigt mindestens 5 WUE-Operationen, um in einen sortierten Stapel ueberfuehrt zu werden. Zum Beispiel mit Hilfe der Operationsfolge $\lambda = (1,\ 1,\ 2,\ 2,\ 4)$:

$$\begin{aligned} \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ 6 & 1 & 8 & 5 & 3 & 7 & 4 & 2 \end{pmatrix} &\xrightarrow{1} \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ 1 & 8 & 5 & 3 & 7 & 4 & 2 \end{pmatrix} \\ &\xrightarrow{1} \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 8 & 5 & 3 & 7 & 4 & 2 \end{pmatrix} \\ &\xrightarrow{2} \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 8 & 3 & 7 & 4 & 2 \end{pmatrix} \\ &\xrightarrow{2} \begin{pmatrix} 1 & 2 & 3 & 4 \\ 8 & 7 & 4 & 2 \end{pmatrix} \\ &\xrightarrow{4} \begin{pmatrix} 1 & 2 & 3 \\ 4 & 7 & 8 \end{pmatrix} \end{aligned}$$

9.

$$P(9) = 5$$

Dazugehoeriges Beispiel (vergleiche mit Beispiel 8!):

$$\pi = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\ 6 & 1 & 8 & 5 & 3 & 7 & 4 & 2 & 9 \end{pmatrix} \in S_9$$

Dieser Stapel benoetigt mindestens 5 WUE-Operationen, um in einen sortierten Stapel ueberfuehrt zu werden. Beispielsweise (wie in Beispiel 8) durch die Operationsfolge $\lambda = (1,\ 1,\ 2,\ 2,\ 4)$:

$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\ 6 & 1 & 8 & 5 & 3 & 7 & 4 & 2 & 9 \end{pmatrix} \xrightarrow{1} \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ 1 & 8 & 5 & 3 & 7 & 4 & 2 & 9 \end{pmatrix}$$
$$\xrightarrow{1} \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ 8 & 5 & 3 & 7 & 4 & 2 & 9 \end{pmatrix}$$
$$\xrightarrow{2} \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 8 & 3 & 7 & 4 & 2 & 9 \end{pmatrix}$$
$$\xrightarrow{2} \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 8 & 7 & 4 & 2 & 9 \end{pmatrix}$$
$$\xrightarrow{4} \begin{pmatrix} 1 & 2 & 3 & 4 \\ 4 & 7 & 8 & 9 \end{pmatrix}$$

10.

$$P(10) = 6$$

Dazugehoeriges Beispiel:

$$\pi = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\ 6 & 4 & 7 & 5 & 8 & 1 & 9 & 3 & 10 & 2 \end{pmatrix} \in S_{10}$$

Dieser Stapel benoetigt mindestens 6 WUE-Operationen, um in einen sortierten Stapel ueberfuehrt zu werden. Beispielsweise durch die Operationsfolge $\lambda = (1,\ 1,\ 4,\ 7,\ 2,\ 5)$:

$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\ 6 & 4 & 7 & 5 & 8 & 1 & 9 & 3 & 10 & 2 \end{pmatrix} \xrightarrow{1} \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\ 4 & 7 & 5 & 8 & 1 & 9 & 3 & 10 & 2 \end{pmatrix}$$
$$\xrightarrow{1} \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ 7 & 5 & 8 & 1 & 9 & 3 & 10 & 2 \end{pmatrix}$$
$$\xrightarrow{4} \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ 8 & 5 & 7 & 9 & 3 & 10 & 2 \end{pmatrix}$$
$$\xrightarrow{7} \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 10 & 3 & 9 & 7 & 5 & 8 \end{pmatrix}$$
$$\xrightarrow{2} \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 10 & 9 & 7 & 5 & 8 \end{pmatrix}$$
$$\xrightarrow{5} \begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 7 & 9 & 10 \end{pmatrix}$$

11.

$$P(11) = 6$$

Dazugehoeriges Beispiel (vergleiche mit Beispiel 10!):

$$\pi = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 \\ 6 & 4 & 7 & 5 & 8 & 1 & 9 & 3 & 10 & 2 & 11 \end{pmatrix} \in S_{11}$$

Dieser Stapel benoetigt ebenfalls mindestens 6 WUE-Operationen, um in einen sortierten Stapel ueberfuehrt zu werden. Beispielsweise durch die Operationsfolge $\lambda = (1,\ 1,\ 4,\ 7,\ 2,\ 5)$, analog zu Beispiel 10:

$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 \\ 6 & 4 & 7 & 5 & 8 & 1 & 9 & 3 & 10 & 2 & 11 \end{pmatrix} \xrightarrow{1} \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\ 4 & 7 & 5 & 8 & 1 & 9 & 3 & 10 & 2 & 11 \end{pmatrix}$$
$$\xrightarrow{1} \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\ 7 & 5 & 8 & 1 & 9 & 3 & 10 & 2 & 11 \end{pmatrix}$$
$$\xrightarrow{4} \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ 8 & 5 & 7 & 9 & 3 & 10 & 2 & 11 \end{pmatrix}$$
$$\xrightarrow{7} \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ 10 & 3 & 9 & 7 & 5 & 8 & 11 \end{pmatrix}$$
$$\xrightarrow{2} \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 10 & 9 & 7 & 5 & 8 & 11 \end{pmatrix}$$
$$\xrightarrow{5} \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 5 & 7 & 9 & 10 & 11 \end{pmatrix}$$

12.

$$P(12) = 7$$

Dazugehoeriges Beispiel:

$$\pi = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 \\ 7 & 2 & 8 & 6 & 4 & 10 & 5 & 3 & 9 & 12 & 1 & 11 \end{pmatrix} \in S_{12}$$

Dieser Stapel benoetigt mindestens 7 WUE-Operationen, um in einen sortierten Stapel ueberfuehrt zu werden. Beispielsweise durch die Operationsfolge $\lambda = (1,\ 1,\ 3,\ 3,\ 4,\ 5,\ 5)$:

$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 \\ 7 & 2 & 8 & 6 & 4 & 10 & 5 & 3 & 9 & 12 & 1 & 11 \end{pmatrix} \xrightarrow{1} \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 \\ 2 & 8 & 6 & 4 & 10 & 5 & 3 & 9 & 12 & 1 & 11 \end{pmatrix}$$
$$\xrightarrow{1} \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\ 8 & 6 & 4 & 10 & 5 & 3 & 9 & 12 & 1 & 11 \end{pmatrix}$$
$$\xrightarrow{3} \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\ 6 & 8 & 10 & 5 & 3 & 9 & 12 & 1 & 11 \end{pmatrix}$$
$$\xrightarrow{3} \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ 8 & 6 & 5 & 3 & 9 & 12 & 1 & 11 \end{pmatrix}$$
$$\xrightarrow{4} \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ 5 & 6 & 8 & 9 & 12 & 1 & 11 \end{pmatrix}$$
$$\xrightarrow{5} \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 9 & 8 & 6 & 5 & 1 & 11 \end{pmatrix}$$
$$\xrightarrow{5} \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 5 & 6 & 8 & 9 & 11 \end{pmatrix}$$

9. Quellcode

Nun folgt der Quellcode der wichtigsten Teile der Loesungsvorschlaege.

9.1. Allgemein

Simple Algorithmen, um zu pruefen, ob ein array bzw. eine Permutation sortiert sind:
Diese beide Methoden befinden sich in der Klasse *Utils* und dienen dazu, zu ueberpruefen, ob ein gegebenes Array an Integern sortiert ist. Die Funktion *isSorted* prueft dabei, ob

$$A[1] < A[2] < \cdots < A[n]$$

gilt, waehrend die Funktion *isSortedPermution* prueft, ob

$$A[i] = i,\ \forall i \in \{1,\ 2,\ \cdots\ ,\ n\}$$

gilt. Die Funktionen werden also dazu genutzt, um fuer einen durch WUE-Operationen entstandenen Stapel zu pruefen, ob dieser sortiert ist, wobei bei der ersten Variante die normalen und bei der zweiten Variante die vereinfachten WUE-Operationen genutzt wurden.

```
1   /**
     * Public static function to check weather a given array is sorted,
     * e.i. A[1] <= A[2] <= ... <= A[n].
     *
     * @param array The array
6    * @return If the array is sorted
     */
    public static boolean isSorted(int[] array) {
        // Arrays of the length 0 or 1 are always sorted
        if (array.length <= 1) return true;
11
        // Check for all pairs (A[i], A[i+1]) if A[i+1] is greater
        // that A[i]. If so, return false
        for (int i = 0; i < array.length - 1; i++) {
            if (array[i] > array[i + 1]) return false;
16      }

        // Finally, return true, since the array must be sorted
        return true;
    }
21
    /**
     * Public static function to check weather a given permutation is sorted,
     * e.i. A[1] = 1, A[2] = 2, ... A[n] = n.
     *
26   * @param array The permutation given as an array of integers
     * @return If the array is sorted
     */
    public static boolean isSortedPermutation(int[] array) {
        // Check for each index i if the current element of the
31      // array is not i+1. If so, return false
        for (int i = 0; i < array.length; i++) {
            if (array[i] != i + 1) return false;
        }

36      // Finally, return true, since the array must be sorted
        return true;
    }
```

Implementierung der normalen (standart) und der vereinfachen (improved) WUE-Operation: Diese beiden Methoden befinden sich ebenfalls in der Klasse Utils und werden benutzt um die normale und die vereinfachte WUE-Operation durchzufuehren. Dabei funktioniert die Funktion *standartWUEOperation* getreu der Definition 19, wie der Algorithmus 11, waehrend die zweite aus Gruenden der Einfachheit zunaechst die erste Funktion verwendet und daraufhin das Ergebnis in eine korrekte Permutation ueberfuehrt, indem bei allen Elementen 1 abgezogen wird, die groeszer sind als das entfernte Element *removed*, anstatt das Verfahren des Algorithmus 12 zu nutzen.

```
/**
 * Public static function doing the WUE-Operation on a given array by
 * creating a new one (not changing the given array) given the 0-based index.
 *
 * @param array The array (won't be changes!)
 * @param index The index for the WUE-Operation
 * @return A new array containing the resulting stack when doing the
 * WUE-Operation on the given array
 * @throws IllegalArgumentException If the given index is invalid, e.i. not in the range [0; n-1]
 */
public static int[] standardWUEOperation(int[] array, int index) {
    // Check if the given index is valid and create the result array
    if (index >= array.length || index < 0) throw new IllegalArgumentException();
    int[] result = new int[array.length - 1];

    // Fill up the result array using the definition of the (normal) WUE-Operations
    for (int i = 0; i < array.length - 1; i++) {
        if (i >= index) {
            result[i] = array[i + 1];
        } else {
            result[i] = array[index - i - 1];
        }
    }

    // Return the result
    return result;
}


/**
 * Public static function doing the improved WUE-Operation on a given array by
 * creating a new one (not changing the array) given the 0-based index.
 *
 * @param array The array
 * @param index The index for the WUE-Operation
 * @return The resulting stack when doing improved the WUE-Operation on the given array
 */
public static int[] improvedWUEOperation(int[] array, int index) {
    // First, get which element of the given array will be removed
    int removed = array[index];

    // Second, compute the normal WUE-Operation on the array (not changing the array)
    int[] result = standardWUEOperation(array, index);

    // Next, make sure the result is actually a permutation,
    // assuming the given array is one.
    for (int i = 0; i < result.length; i++) {
        if (result[i] >= removed) {
            result[i]--;
        }
    }

    // Finally, return the result
    return result;
}
```

Ueberblick ueber die Klasse IntArray:
Wie bereits erwaehnt, wird diese Klasse verwendet, um als Wrapper fuer Integer-Array zu fungieren, um das Hashing zu vereinfachen.
Um diese Klasse uebersichtlich darzustellen, wird im Folgenden bei einigen Funktionen der Rumpf und die Kommentierung weggelassen.

```
/**
 * This class is an Int-Array wrapper, providing some useful methods.
 * E.g., checking if the array is sorted, swapping elements and
 * especially computing the array's hashcode.
 *
 */
public class IntArray implements Cloneable {

    /**
     * The internal int array
     */
    private final int[] array;

    /**
     * Public constructor taking an int array and set it to be this classes
     * internal int-array, not(!) copying the given array.
     *
     * @param array The array that will be used as this classes internal array, not being copied!
     */
    public IntArray(int[] array) {
        this.array = array;
    }

    public boolean isSorted() {  }

    public void swap(int i, int j) {  }

    public Object clone() { $\dots$ }

    public boolean equals(Object o) {  }

    public int hashCode() {
        if (this.array == null) return 0;

        int result = 1;
        for (int element : this.array) {
            result = 31 * result + element;
        }

        return result;
    }

    // Getter und Setter ohne ihre Implementierungen:
    public String toString() {  }

    public int get(int index) {  }

    public void set(int index, int value) {  }

    public int getLength() {  }

    public int[] getArray() {  }
}
```

9.2. Stapelsortierung / MWO-Problem (Aufgabenteil a)

Abstrakte Eltern-Klasse PancakeSort:
Wie bereits erwaehnt, existieren fuer die Klassen, die die Probleme loesen koennen, je eine Oberklasse. Fuer die Klassen, die das MWO-Problem loesen koennen, ist dies die Klasse PancakeSort, die wie folgt aussieht:

```
/**
 * Abstract parent class of classes solving the PWZ-Problem of finding P(n).
 *
 */
public abstract class PZSolver {

    /**
     * The value of n, as a protected variable such that subclasses can use it.
     */
    protected final int n;

    /**
     * Public constructor only taking in the value of n, such that P(n) can be computed.
     *
     * @param n The value n, such that P(n) can be computed.
     */
    public PZSolver(int n) {
        this.n = n;
    }

    /**
     * Abstract methode that should return a P(N) and a corresponding example, e.i.,
     * a stack of the size n that needs at least P(N) WUE-Operation to be sorted.
     *
     * @return P(n) and the corresponding example stack
     */
    public abstract Pair<Integer, int[]> solve();

}
```

Implementierung des Algorithmus 5:
Die Implementierung des Algorithmus 5 befindet sich in der Klasse *PancakeSort1* und erbt von der Klasse *PancakeSort*. Sie ist sehr nah am Pseudocode mit dem Hauptunterschied, dass in dieser Implementierung nicht die Indizes von 1 bis $n - |\lambda|$, sondern die von 0 bis $n - |\lambda| - 1$, probiert werden. Die Operationsfolge wird dabei durch eine Liste an Integern representiert.

```
    /**
     * Implementation of the solve-methode solving the MWO-Problem as described above.
     * @return An int array containing the indexes (0-based) for the WUE-Operations
     */
    @Override
    public int[] solve() {
        // The size of the given stack (array)
        int size = this.getArray().length;

        // Go through all possible sizes of index series.
        // Note that the size (size = n-1) always creates a sorted stack.
        for (int m = 0; m < size; m++) {
            // Initialize the current list of indexes
            List<Integer> list = new ArrayList<>();

            // Recursively test out every possible list of indexes of the current size m.
            // If the methode returns true, a list of the current size m was found (bestList).
            if (solveRecursively(list, m, size)) {
                break;
            }
        }

        // Return the best list (of indexes) as an int array
        return Utils.toArray(this.bestList);
    }
```

```java
/**
 * Private methode for recursively trying out every possible list of indexes
 * of a given size m used for the WUE-Operation and return weather any list of
 * this size can sort the given stack (permutation) when used as a series of WUE-Operations
 * @param list The current list of indexes
 * @param m    The wanted size for the list of indexes
 * @param n    The size of the given stack
 * @return true, if a list of indexes of the given size m, converting
 * the given stack into a sorted one, exists. False otherwise.
 */
private boolean solveRecursively(List<Integer> list, int m, int n) {
    // If the current list of indexes has reached the wanted size,
    // check if converts the given stack into a sorted one by
    // applying the normal WUE-Operation.
    if (list.size() == m) {
        // Copy the given stack (array)
        int[] result = this.getArray().clone();
        // Apply the WUE-Operations given by the current list of indexes
        for (int index : list) {
            result = Utils.standardWUEOperation(result, index);
        }

        // Check if the resulting stack is sorted.
        // If so, set the bestList field to the current list and return true.
        if (Utils.isSorted(result)) {
            this.bestList = new ArrayList<>(list);
            return true;
        }

        // Otherwise, return false.
        return false;
    }
    // Iterate through all possible indexes (0 to n - list.size() - 1)
    for (int index = 0; index < n - list.size(); index++) {
        // Add the current index to the list of indexes
        list.add((Integer) index);

        // Recursively use this methode to create lists of indexes
        // starting with the current one (or check the current one)
        if (solveRecursively(list, m, n)) {
            return true;
        }
        // Remove the last element of the list
        list.remove(list.size() - 1);
    }
    return false;  }
```

Implementierung des Algorithmus 6:
Die Implementierung des Algorithmus 6 befindet sich in der Klasse *PancakeSort2* und funktioniert fast genauso, wie der Pseudocode. Mit dem erneuen Unterschied, dass die Indizes (wie zuvor) von 0 bis $n-1$ probiert werden, und die eigentlichen Operationsfolgen (aus erstem dem aktuellen Index und dann den rekursiv gefundenen) wird nur erstellt, wenn die gefundene Operationsfolge tatsaechlich besser ist, als eine vorherige (oder die erste ist).

```java
/**
 * Implementation of the solve-methode solving the MWO-Problem as described above.
 *
 * @return An int array containing the indexes (0-based) for the WUE-Operations
 */
@Override
public int[] solve() {
    return Utils.toArray(solve(this.getArray()));
}

/**
 * Recursive private methode for finding the smallest list of indexes converting the
 * given stack (array) into a sorted one
 *
 * @param array The current stack
 * @return The smallest possible list of 0-based indexes that can convert the given
 * stack in a sorted one using WUE-Operations.
 */
private List<Integer> solve(int[] array) {
    // Check if the given array (stack) is sorted. If so return an empty list.
    if (Utils.isSorted(array)) return new ArrayList<>();

    // Keep track of the best list (of indexes for the WUE-Operations) using this list
    List<Integer> bestList = null;

    // Iterate through all possible indexes for WUE-Operations on the current stack
    for (int index = 0; index < array.length; index++) {
        // Apply the normal WUE-Operation on the given array (stack) using the current index
        // and solve the problem recursively.
        int[] currentArray = Utils.standardWUEOperation(array, index);
        List<Integer> recursiveResult = solve(currentArray);

        // If the best list is still null or the current list (recursiveResult.size() + 1!)
        // si smaller than the best list, update the best list to the current one
        if (bestList == null || recursiveResult.size() + 1 < bestList.size()) {
            // Create the current list by first adding the current index and then adding the recursive
            bestList = new ArrayList<>();
            bestList.add(index);
            bestList.addAll(recursiveResult);
        }
    }

    // Finally, return the best list (of indexes)
    return bestList;
}
```

Implementierung des Algorithmus 13 und Algorithmus 14:
Diese beiden Algorithmen sind fast identisch zu den vorherigen zwei und befinden sich in den Klassen *PancakeSort3* und *PancakeSort4*. Die Unterschiede bestehen darin, dass nun in der Implementierung des Algorithmus 13

```
// Apply the WUE-Operations given by the current list of indexes
for (int index : list) {
    result = Utils.improvedWUEOperation(result, index);
}

// Check if the resulting stack is sorted.
// If so, set the bestList field to the current list and return true.
if (Utils.isSortedPermutation(result)) {
    this.bestList = new ArrayList<>(list);
    return true;
}
```

in der Methode *solveRecursively* genutzt wird, um eine moegliche Operationsfolge zu testen und in der Implementierung des Algorithmus 14 nun

```
// Apply the improved WUE-Operation on the given array (stack) using the current index
// and solve the problem recursively.
int[] currentArray = Utils.improvedWUEOperation(array, index);
List<Integer> recursiveResult = solve(currentArray);
```

benutzt wird, um die (vereinfachte) WUE-Operation anzuwenden und das Problem rekursiv zu loesen.

Implementierung des Algorithmus 17:
In der Implementierung des Algorithmus 17 (Klasse *PancakeSortDP*) wird zunaechst eine HashMap initialisiert:

```
    /**
     * HashMap mapping from the int-array wrapper class IntArray
     * to lists of integer. This hashmap contains the results of the
     * solveRecursively-methode, such that subproblems that have already been solved are
     * returned instantly.
     */
    private final Map<IntArray, List<Integer>> dp;
```

und anschlieszend die Methode rekursive *solveRecursively* implementiert:

```
    /**
     * Implementation of the solve-methode solving the MWO-Problem as described above.
     *
     * @return An int array containing the indexes (0-based) for the WUE-Operations
     */
    @Override
    public int[] solve() {
        // Convert the given int array into an instance of the wrapper class IntArray
        IntArray arr = new IntArray(this.getArray());
        return Utils.toArray(solveRecursively(arr));
    }

    /**
     * Recursive private methode for finding the smallest list of indexes converting the
     * given stack (array) into a sorted one using dynamic programming.
     *
     * @param array The current stack
     * @return The smallest possible list of 0-based indexes that can convert the given
     * stack in a sorted one using WUE-Operations.
     */
    private List<Integer> solveRecursively(IntArray array) {
        // Check if the given array (stack) is sorted. If so, return an empty list.
        if (Utils.isSortedPermutation(array.getArray())) return new ArrayList<>();

        // Check if the current array has already been computed (e.i., is
        // contained in dp). If so, return the computed value.
        if (this.dp.containsKey(array)) return this.dp.get(array);

        // Keep track of the best list (of indexes for the WUE-Operations) using this list
        List<Integer> bestList = null;

        // Iterate through all possible indexes for WUE-Operations on the current stack
        for (int i = 0; i < array.getLength(); i++) {
            // Apply the improved WUE-Operation on the given array (stack) using the current index
            // and solve the problem recursively.
            IntArray currentArray = Utils.improvedWUEOperation(array, i);
            List<Integer> recursiveResult = solveRecursively(currentArray);

            // If the best list is still null or the current list (recursiveResult.size() + 1!)
            // is smaller than the best list, update the best list to the current one
            if (bestList == null || recursiveResult.size() + 1 < bestList.size()) {
                // Create the current list by first adding the current index and then adding the recursive
                bestList = new ArrayList<>();
                bestList.add(i);
                bestList.addAll(recursiveResult);
            }
        }

        // Finally, put the best list (of indexes) into the dp HashMap for the given array and return it.
        this.dp.put(array, bestList);
        return bestList;
    }
```

9.3. PWUE-Zahlen / PWZ-Problem (Aufgabenteil b)

Abstrakte Eltern-Klasse PZSolver:
Wie bereits erwaehnt, existieren fuer die Klassen, die die Probleme loesen koennen, je eine Oberklasse. Fuer die Klassen, die das PWZ-Problem loesen koennen, ist dies die Klasse PZSolver, die wie folgt aussieht:

```
/**
 * Abstract parent class of classes solving the PWZ-Problem of finding P(n).
 *
 */
public abstract class PZSolver {

    /**
     * The value of n, as a protected variable such that subclasses can use it.
     */
    protected final int n;

    /**
     * Public constructor only taking in the value of n, such that P(n) can be computed.
     *
     * @param n The value n, such that P(n) can be computed.
     */
    public PZSolver(int n) {
        this.n = n;
    }

    /**
     * Abstract methode that should return a P(N) and a corresponding example, e.i.,
     * a stack of the size n that needs at least P(N) WUE-Operation to be sorted.
     *
     * @return P(n) and the corresponding example stack
     */
    public abstract Pair<Integer, int[]> solve();
}
```

Implementierung des Algorithmus 10 mit Algorithmus 7 zum bestimmen der $A(\pi)$:
Die Implementierung des Algorithmus 10 mit Algorithmus 7 zum bestimmen der $A(\pi)$ befindet sich in der Klasse *PZSolver1*.
Bis auf die 0-basierte Indexierung und die Vereinfachung der Bestimmung der Elemete der aktuelle Permutation, die im Heap-Algorithmus getauscht werden, durch den ternaeren Operator (bzw. den conditional operator), ist diese Implementierung sehr exakte die des Pseudocodes.

```
/**
 * Implementation of the solve-methode solving the PWZ-Problem as described above.
 *
 * @return A pair containing P(n) and a corresponding example stack
 */
@Override
public Pair<Integer, int[]> solve() {
    // Create the start permutation containing 1, 2, ... , n
    int[] list = new int[this.n];
    for (int i = 1; i <= this.n; i++) {
        list[i - 1] = i;
    }

    // Keep track of the worst permutation found so far and the (smallest) amount of
    // WUE-Operations needed to solve it. That is, the stack (permutation) with the
    // greatest smallest amount of WUE-Operations needed to sort the stack.
    int[] worstPermutation = list.clone();
    int worstOperations = 0;

    // Use Heap's Algorithmus to iterate through all permutations of the numbers 1, 2, ... , n.
    int[] c = new int[this.n];
    int i = 1;

    while (i < this.n) {
        if (c[i] < i) {
            Utils.swap(list, i % 2 == 0 ? 0 : c[i], i);
            c[i]++;
            i = 1;

            // Check the current permutation S by first copying it and then
            // computing its value A(S)
            int currentNumber = getA(list);

            // If the smallest amount of WUE-Operations needed to sort the
            // current permutation (currentNumber) is greater than the
            // greatest found so far, update the worst permutation and its length.
            if (currentNumber > worstOperations) {
                worstPermutation = list.clone();
                worstOperations = currentNumber;
            }
        } else {
            c[i] = 0;
            i += 1;
        }
    }

    // Return the worst permutation and its length in a pair.
    return new Pair<>(worstOperations, worstPermutation);
}
```

```java
    /**
     * Recursive private methode for finding A(S) for a given stack S.
     *
     * @param permutation The given stack S
     * @return The value of A(S)
     */
    private int getA(int[] permutation) {
        // Go through all possible sizes of index series.
        // Note that the size (permutation.length - 1 = n-1) always creates a sorted stack.
        for (int m = 0; m < permutation.length - 1; m++) {
            // Initialize the current list of indexes
            List<Integer> list = new ArrayList<>();

            // Recursively test out every possible list of indexes of the current size m.
            // If the methode returns true, a list of the current size m was found, such
            // that m can be returned.
            if (solveRecursively(list, m, permutation)) {
                return m;
            }
        }

        // If none of the sizes before worked, return (permutation.length - 1 = n-1)
        // since this size always creates a sorted stack.
        return permutation.length - 1;
    }

    /**
     * Private methode for recursively trying out every possible list of indexes of a given size m
     * used for the WUE-Operation and return weather any
     * list of this size can sort a given stack (permutation)
     * when used as a series of WUE-Operations
     *
     * @param list        The current list of indexes
     * @param m           The wanted size for the list of indexes
     * @param permutation The
     * @return true, if a list of indexes of the given size m, converting
     * the given stack into a sorted one, exists. False otherwise.
     */
    private boolean solveRecursively(List<Integer> list, int m, int[] permutation) {
        // If the current list of indexes has reached the wanted size,
        // check if converts the given stack into a sorted one by
        // applying the normal WUE-Operation.
        if (list.size() == m) {
            // Copy the given stack (permutation)
            int[] result = permutation.clone();

            // Apply the WUE-Operations given by the current list of indexes
            for (int index : list) {
                result = Utils.standardWUEOperation(result, index);
            }

            // Return weather the current list of indexes could sort the stack (permutation)
            return Utils.isSorted(result);
        }

        // Iterate through all possible indexes (0 to n - list.size() - 1)
        for (int index = 0; index < n - list.size(); index++) {
            // Add the current index to the list of indexes
            list.add((Integer) index);

            // Recursively use this methode to create lists of indexes
            // starting with the current one (or check the current one)
            if (solveRecursively(list, m, permutation)) {
                return true;
            }

            // Remove the last element of the list
            list.remove(list.size() - 1);
        }
        return false;
    }
```

Implementierung des Algorithmus 10 mit Algorithmus 8 zum bestimmen der $A(\pi)$:
Diese Implementierung des Algorithmus 10 ist sehr aehnlich zur vorherigen, mit dem Unterschied, dass nun Algorithmus 8 genutzt wird, um $A(\pi)$ zu bestimmen. Auch diese Implementierung ist bis auf die zuvor erwaehnten Unterschiede sehr originalgetreu unf befindet sich in der Klasse *PZSolver2*

```
    /**
     * Implementation of the solve-methode solving the PWZ-Problem as described above.
     * @return A pair containing P(n) and a corresponding example stack
     */
    @Override
    public Pair<Integer, int[]> solve() {
        // Create the start permutation containing 1, 2, ... , n
        int[] list = new int[this.n];
        for (int i = 1; i <= this.n; i++) {
            list[i-1] = i;
        }
        // Keep track of the worst permutation found so far and the (smallest) amount of
        // WUE-Operations needed to solve it. That is, the stack (permutation) with the
        // greatest smallest amount of WUE-Operations needed to sort the stack.
        int[] worstPermutation = list.clone();
        int worstOperations = 0;
        // Use Heap's Algorithmus to iterate through all permutations of the numbers 1, 2, ... , n.
        int[] c = new int[this.n];
        int i = 1;

        while (i < this.n) {
            if (c[i] < i) {
                Utils.swap(list, i % 2 == 0 ? 0 : c[i], i);
                c[i]++;
                i = 1;
                // Check the current permutation S by first copying it and then
                // computing its value A(S)
                int currentNumber = getA(list);

                // If the smallest amount of WUE-Operations needed to sort the
                // current permutation (currentNumber) is greater than the
                // greatest found so far, update the worst permutation and its length.
                if (currentNumber > worstOperations) {
                    worstPermutation = list.clone();
                    worstOperations = currentNumber;
                }
            } else {
                c[i] = 0;
                i += 1;
            }
        }
        // Return the worst permutation and its length in a pair.
        return new Pair<>(worstOperations, worstPermutation);
    }
    /**
     * Recursive private methode for finding A(S) for a given stack S.
     * @param permutation The given stack S
     * @return The value of A(S)
     */
    private int getA(int[] permutation) {
        // Check if the permutation is sorted
        if (Utils.isSorted(permutation)) return 0;
        // Keep track of the smallest value for A(S) found recursively
        int min = -1;
        // Iterate through all possible indexes for WUE-Operations on the current stack
        for (int i = 0; i < permutation.length; i++) {
            // Apply the normal WUE-Operation on the given array (stack) using the current index
            // and solve the problem recursively.
            int[] currentPermutation = Utils.standardWUEOperation(permutation, i);
            int currentNumber = getA(currentPermutation) + 1;

            // If the smallest A(S) value min is still -1 or the current one is smaller
            // than min, update min to be the current value currentNum.
            if (min == -1 || currentNumber < min) {
                min = currentNumber;
            }
        }
        // Return the value of A(S)
        return min; }
```

Implementierung des Algorithmus 10 mit Algorithmus 15, bzw. Algorithmus 16 zum bestimmen der $A(\pi)$:
Auch diese beiden Algorithmen sind sehr aehnlich zu den letzten beiden.
Die Implementierung, die Algorithmus 15 benutzt, befindet sich in der Klasse *PZSolver3* testet nun die aktuelle Operationsfolge mit

```
            // Apply the WUE-Operations given by the current list of indexes
            for (int index : list) {
                result = Utils.improvedWUEOperation(result, index);
            }

            // Return weather the current list of indexes could sort the stack (permutation)
            return Utils.isSortedPermutation(result);
```

und die Implementierung mit Algorithmus 16 (Klasse *PZSolver4*) nutzt nun

```
            // Apply the improved WUE-Operation on the given array (stack) using the current index
            // and solve the problem recursively.
            int[] currentPermutation = Utils.improvedWUEOperation(permutation, i);
            int currentNumber = getA(currentPermutation) + 1;
```

um die (vereinfachte) WUE-Operation anzuwenden und das Problem rekursiv zu loesen (bzw., um $A(\pi)$ rekursiv zu bestimmen).

Implementierung des Algorithmus 18:
Auch hier (Klasse *PZSolverDP1*) wird zunaechst eine HashMap initialisiert:

```
    /**
     * HashMap mapping from the int-array wrapper class IntArray
     * to integers. This hashmap contains the results of the
     * getA-methode, such that subproblems that have already been solved are
     * returned instantly.
     */
    private final Map<IntArray, Integer> dp = new HashMap<>();
```

Und der restliche Code sieht wie folgt aus:

```
    /**
     * Implementation of the solve-methode solving the PWZ-Problem as described above.
     *
     * @return A pair containing P(n) and a corresponding example stack
     */
    @Override
    public Pair<Integer, int[]> solve() {
        // Create the start permutation containing 1, 2, ... , n
        IntArray list = new IntArray(this.n);
        for (int i = 1; i <= this.n; i++) {
            list.set(i - 1, i);
        }

        // Keep track of the worst permutation found so far and the (smallest) amount of
        // WUE-Operations needed to solve it. That is, the stack (permutation) with the
        // greatest smallest amount of WUE-Operations needed to sort the stack.
        IntArray worstPermutation = (IntArray) list.clone();
        int worstOperations = getA(worstPermutation);

        // Use Heap's Algorithmus to iterate through all permutations of the numbers 1, 2, ... , n.
        int[] c = new int[this.n];
        int i = 1;

        while (i < this.n) {
            if (c[i] < i) {
                list.swap(i % 2 == 0 ? 0 : c[i], i);
                c[i]++;
                i = 1;

                // Check current permutation S by first copying it
                // and then computing its value A(S)
                IntArray currentPermutation = (IntArray) list.clone();
                int currentNum = getA(currentPermutation);
```

```
                // If the smallest amount of WUE-Operations needed to sort the
                // current permutation (currentNumber) is greater than the
                // greatest found so far, update the worst permutation and its length.
                if (currentNum > worstOperations) {
                    worstPermutation = currentPermutation;
                    worstOperations = currentNum;
                }
            } else {
                c[i] = 0;
                i += 1;
            }
        }

        // Return the worst permutation and its length in a pair.
        return new Pair<>(worstOperations, worstPermutation.getArray());
    }

    /**
     * Recursive private methode for finding A(S) for a given stack S using dynamic programming.
     *
     * @param permutation The given stack S
     * @return The value of A(S)
     */
    private int getA(IntArray permutation) {
        // Check if the given array (stack) is sorted. If so, return an empty list.
        if (Utils.isSortedPermutation(permutation.getArray())) return 0;

        // Check if the current array has already been computed (e.i., is
        // contained in dp). If so, return the computed value.
        if (this.dp.containsKey(permutation)) return this.dp.get(permutation);

        // Keep track of the best number of operations
        int min = -1;

        // Iterate through all possible indexes for WUE-Operations on the current stack
        for (int i = 0; i < permutation.getLength(); i++) {
            // Apply the improved WUE-Operation on the given array (stack) using the current index
            // and solve the problem recursively.
            IntArray currentPermutation = Utils.improvedWUEOperation(permutation, i);
            int currentNumber = getA(currentPermutation) + 1;

            // If min is still -1 or the current result (currentNumber) is smaller
            // than min, update min to the currentNumber
            if (min == -1 || currentNumber < min) {
                min = currentNumber;
            }
        }

        // Finally, put the smallest number (min) into the dp HashMap
        // for the given permutation and return it.
        this.dp.put(permutation, min);
        return min;
    }
```

Implementierungen des Algorithmus 19:
Die erste Implementierungen des Algorithmus 19 in der Klasse *PZSolverDP2* verwendet das HashVerfahren, das in der Klasse *IntArray* definiert wurde:

```
    /**
     * Implementation of the solve-methode solving the PWZ-Problem as described above.
     *
     * @return A pair containing P(n) and a corresponding example stack
     */
    @Override
    public Pair<Integer, int[]> solve() {
        // Compute all values of A(S) where S has the size n-1
        // using dynamic programming, going from the bottom up to n-1
        Map<IntArray, Integer> dp = new HashMap<>();

        // Start with the size 1
        dp.put(new IntArray(new int[]{1}), 0);

        // Go bottom up, computing the next dp-table using the last one,
        // up to the dp-table for the S' of size n-1.
        for (int N = 2; N < this.n; N++) {
            dp = getNextDP(N, dp);
        }

        // Create the start permutation containing 1, 2, ... , n
        IntArray list = getStartPermutation(this.n);

        // Keep track of the worst permutation found so far and the (smallest) amount of
        // WUE-Operations needed to solve it. That is, the stack (permutation) with the
        // greatest smallest amount of WUE-Operations needed to sort the stack.
        IntArray worstPermutation = (IntArray) list.clone();
        int worstOperations = 0;

        // Use Heap's Algorithmus to iterate through all permutations of the numbers 1, 2, ... , n.
        int[] c = new int[this.n];
        int i = 1;

        while (i < this.n) {
            if (c[i] < i) {
                list.swap(i % 2 == 0 ? 0 : c[i], i);
                c[i]++;
                i = 1;

                // Check current permutation S by computing its value A(S) using the dp-table
                int currentNumber = getOperationNumber(list, dp);

                // If the smallest amount of WUE-Operations needed to sort the
                // current permutation (currentNumber) is greater than the
                // greatest found so far, update the worst permutation and its length.
                if (currentNumber > worstOperations) {
                    worstPermutation = (IntArray) list.clone();
                    worstOperations = currentNumber;
                }
            } else {
                c[i] = 0;
                i += 1;
            }
        }


        // Return the worst permutation and its length in a pair.
        return new Pair<>(worstOperations, worstPermutation.getArray());
    }
```

Bei dieser Implementierung werden zunaechst iterativ die Hashtabellen fuer die Permutationen der Groeszen 1 bis $n-1$ erstellt, ohne immer die Permutation mit dem groesten $A(\pi)$ Wert zu speichern, da dieser eigentlich nur fuer die Groesze n relevant ist. Zudem wird dadurch nicht eine zusaetzlich Hashtabelle (fuer die Groesze n) gefuellt, die eigentlich nicht gebraucht wird.
Auszerdem muss die Methode *getOperationNumber* nicht pruefen, ob die gegebene Permutation sortiert ist, weil die einzig sortierte Permutation nicht getestet sondern uebersprungen und vorher mit dem Wert 0 in die HashMap eingetragen wurde.

Und weiter:

```java
/**
 * Private methode computing the next dp-table, that will contain all A(S) for permutations
 * S of the size n, using the dp-table containing the results A(S') of all stacks of the size n-1
 *
 * @param n The size of the permutations
 * @param dp The dp-table containing the results A(S') of all stacks of the size n-1
 * @return The nex dp-table containing all A(S) for permutations of the size n
 */
private Map<IntArray, Integer> getNextDP(int n, Map<IntArray, Integer> dp) {
    // Initialize the next dp-table (HashMap)
    Map<IntArray, Integer> nextDP = new HashMap<>();

    // Put the first (sorted) permutation into the next dp-table
    IntArray list = getStartPermutation(n);

    IntArray firstPermutation = (IntArray) list.clone();
    nextDP.put(firstPermutation, 0);

    // Heap-Algorithmus to iterate through all permutations
    int[] c = new int[n];
    int i = 1;

    // Use Heap's Algorithmus to iterate through all permutations of the numbers 1, 2, ... , n.
    while (i < n) {
        if (c[i] < i) {
            list.swap(i % 2 == 0 ? 0 : c[i], i);
            c[i]++;
            i = 1;

            // Check current permutation S by first copying it
            // and then computing its value A(S) using the dp-table
            IntArray currentPermutation = (IntArray) list.clone();
            int currentNum = getOperationNumber(currentPermutation, dp);

            // Next put the result into the next dp-table
            nextDP.put(currentPermutation, currentNum);
        } else {
            c[i] = 0;
            i += 1;
        }
    }

    return nextDP;
}

/**
 * Private methode computing A(S) for a given permutation S of size n using the values
 * of a dp-table containing the A(S') results for permutations of size n-1
 *
 * @param permutation The given permutation S
 * @param dp          The dp-table containing the A(S') results for permutations of size n-1
 * @return The value A(S) for the given permutation S
 */
private int getOperationNumber(IntArray permutation, Map<IntArray, Integer> dp) {
    // Keep track of the best number of operations
    int min = -1;

    // Iterate through all possible indexes for WUE-Operations on the current stack
    for (int i = 0; i < permutation.getLength(); i++) {
        // Apply the improved WUE-Operation on the given array (stack) using the current index,
        // receiving a new permutation S' and get A(S') using the dp-table
        IntArray currentPermutation = Utils.improvedWUEOperation(permutation, i);
        int currentNumber = dp.get(currentPermutation) + 1;

        // If min is still -1 or the current result (currentNumber) is smaller
        // than min, update min to the currentNumber
        if (min == -1 || currentNumber < min) {
            min = currentNumber;
        }
    }

    // Finally, return min
    return min;
}
```

Die zweite Implementierung in der Klasse *PZSolverDP3* verwendet hingegen ein eigenes Hash-Verfahren, das fuer $n \leq 15$ funktioniert:

```
    /**
     * Custom hash function computing the hash value for a given permutation of a size n <= 16.
     *
     * @param list The given permutation
     * @return The permutation's hash value
     */
    private long hash(int[] list) {
        int n = list.length;

        // Make sure the permutation's size is smaller than 16.
        if (n >= 16) throw new IllegalArgumentException();

        // Compute the hash by seeing the given permutation as a number
        // with base n and convert it to a decimal number.
        // E.i., computing \sum_{i=0}^{n-1} n^i * A[i] for a permutation A.
        long hash = 0;
        long mul = 1;
        long base = n;

        for (int i : list) {
            hash += mul * i;
            mul *= base;
        }

        // Return the computed hash.
        return hash;
    }
```

Um dieses zu nutzen wird die HashMap wie folgt definiert:

```
        // Compute all values of A(S) where S has the size n-1
        // using dynamic programming, going from the bottom up to n-1
        Map<Long, Integer> dp = new HashMap<>();
```

und zu Einfuegen in die HashMaps wird wie folgt vorgegangen:

```
        // Start with the size 1
        dp.put(hash(new int[]{1}), 0);
```

und

```
        // Create the start permutation containing 1, 2, ... , n
        int[] list = getStartPermutation(n);

        // Put the first (sorted) permutation into the next dp-table
        int[] firstPermutation = list.clone();
        nextDP.put(hash(firstPermutation), 0);
```

Sowie

```
        // Next put the result into the next dp-table
        nextDP.put(hash(currentPermutation), currentNum);
```

Der Rest der Implementation ist identisch zur vorherigen.

10. Literatur

[1] Dweighter, Harry (Dez. 1975), «Elementary Problem E2569», American Mathematical Monthly, Vol. 82, S. 1010

[2] Dweighter, Harry; Garey, Michael R.; Johnson, David S.; Lin, Shen (Apr. 1977), «Elementary Problem E2569*», American Mathematical Monthly, Vol. 84, S. 296

[3] Bulteau, Laurent; Fertin, Guillaume; Rusu, Irena (Dez. 2015), «Pancake Flipping Is Hard», Journal of Computer and System Sciences, Vol. 81, Issue 8, S. 1556 - 1574

[4] Heydari, Mohammad Hossain; Sudborough, Ivan Hal (Okt. 1997), «On the Diameter of the Pancake Network», Journal of Algorithms, Vol. 25, Issue 1, S. 67-94

[5] Chitturi, Bhadrachalam; Fahle, W.; Meng, Z.; Morales, Linda; Shields, Charles O.; Sudborough, Ivan Hal; Voit, Walter E. (Aug. 2009), «An $(18/11)n$ upper bound for sorting by prefix reversals», Theoretical Computer Science, Vol. 410, Issue 36, S. 3372-3390

[6] Cibulka, Josef (Mar. 2011), «On average and highest number of flips in pancake sorting», Theoretical Computer Science, Vol. 412, Issues 8-10, S. 822-834

[7] Gates, William „Bill“ Henry; Papadimitriou, Christos H. (1979) «Bounds for sorting by prefix reversal», Discrete Mathematics, Vol. 27, Issue 1, S. 47-57

[8] Cauchy, Augustin-Louis (Jan. 1815) «Memoire Sur le Nombre des Valeurs qu'une Fonction peut acquerir, lorsqu'on y permute de toutes les manieres possibles les quantites qu'elle renferme» (Paper on the number of values that a function can return when the variables it contains are permuted in all possible ways), Journal de l'Ecole polytechnique, Vol. 10, S. 1-28

[9] Wussing, Hans (2007) «The Genesis of the Abstract Group Concept: A Contribution to the History of the Origin of Abstract Group Theory», Courier Dover Publications, S. 94

[10] Heap, B. R. (Nov. 1963), «Permutations by Interchanges», The Computer Journal, Vol. 6, Issue 3, S. 293-298