Aufgabe 3: Pancake Sort

Finn Rudolph

Teilnahme-ID: 67571

16. April 2023

Inhaltsverzeichnis

1 Lösungsidee

Pancake Sort ist ein bekanntes Problem, in dem es darum geht, mit einer geringstmöglichen Anzahl an Präfixumkehrungen zu sortieren. In dieser ursprünglichen Form wird der oberste Pfannkuchen allerdings nicht aufgegessen.

Ein Stapel von n Pfannkuchen wird als Permutation

$$p = p_0, p_1, p_2, \ldots, p_{n-1}$$

definiert, wobei $0 \leq p_i \leq n-1$ und $p_i \neq p_j$, für $0 \leq i, j \leq n-1$ und $i \neq j$. p_0 ist der oberste Pfannkuchen, p_{n-1} der unterste. Um eine Wende-und-Ess-Operation kompakt auszudrücken, wird der Wende-und-Ess-Operator γ_i eingeführt.

Definition 1. Sei p eine Permutation von Länge n. Der Wende-und-Ess-Operator γ_i ist für p wie folgt definiert.

$$\gamma_i p = p'_{i-1}, p'_{i-2}, \ldots, p'_1, p'_0, p'_{i+1}, p'_{i+2}, \ldots, p'_{n-1}$$

$$p'_j = \begin{cases} p_j & \text{wenn } p_j < p_i \\ p_j - 1 & \text{wenn } p_j > p_i \end{cases} \quad 0 \leq j \leq n-1, j \neq i$$

$\gamma_i p$ bezeichnet also die Permutation der Länge $n-1$, die man erhält, wenn die ersten $i+1$ Elemente von p umgekehrt werden und anschließend das erste entfernt wird. Um tatsächlich eine Permutation der Länge $n-1$ zu erhalten, werden durch γ_i außerdem alle Elemente von p, die größer als p_i sind, um 1 verkleinert. Das erhält die relative Ordung der Elemente, sodass die optimale Folge an Wende-und-Ess-Operationen unverändert bleibt. Zum Beispiel, wenn $p = 3, 0, 1, 2$, dann ist

$$\gamma_2 p = \gamma_2(3, 0, 1, 2) = 0, 3-1, 2-1 = 0, 2, 1$$

Um später das in Teilaufgabe a) vorliegende Problem klar benennen zu können, wird ihm der Name "γ_i-Pancake Sort" gegeben. I^n bezeichnet die identische Permutation der Länge n.

Definition 2 (γ_i-Pancake Sort)**.** Gegeben sei eine Permutation p der ersten n natürlichen Zahlen. Das Problem, eine kürzestmögliche Folge an γ_i-Operationen $\gamma_{i_0}, \gamma_{i_1}, \ldots \gamma_{i_{k-1}}$ zu finden, sodass

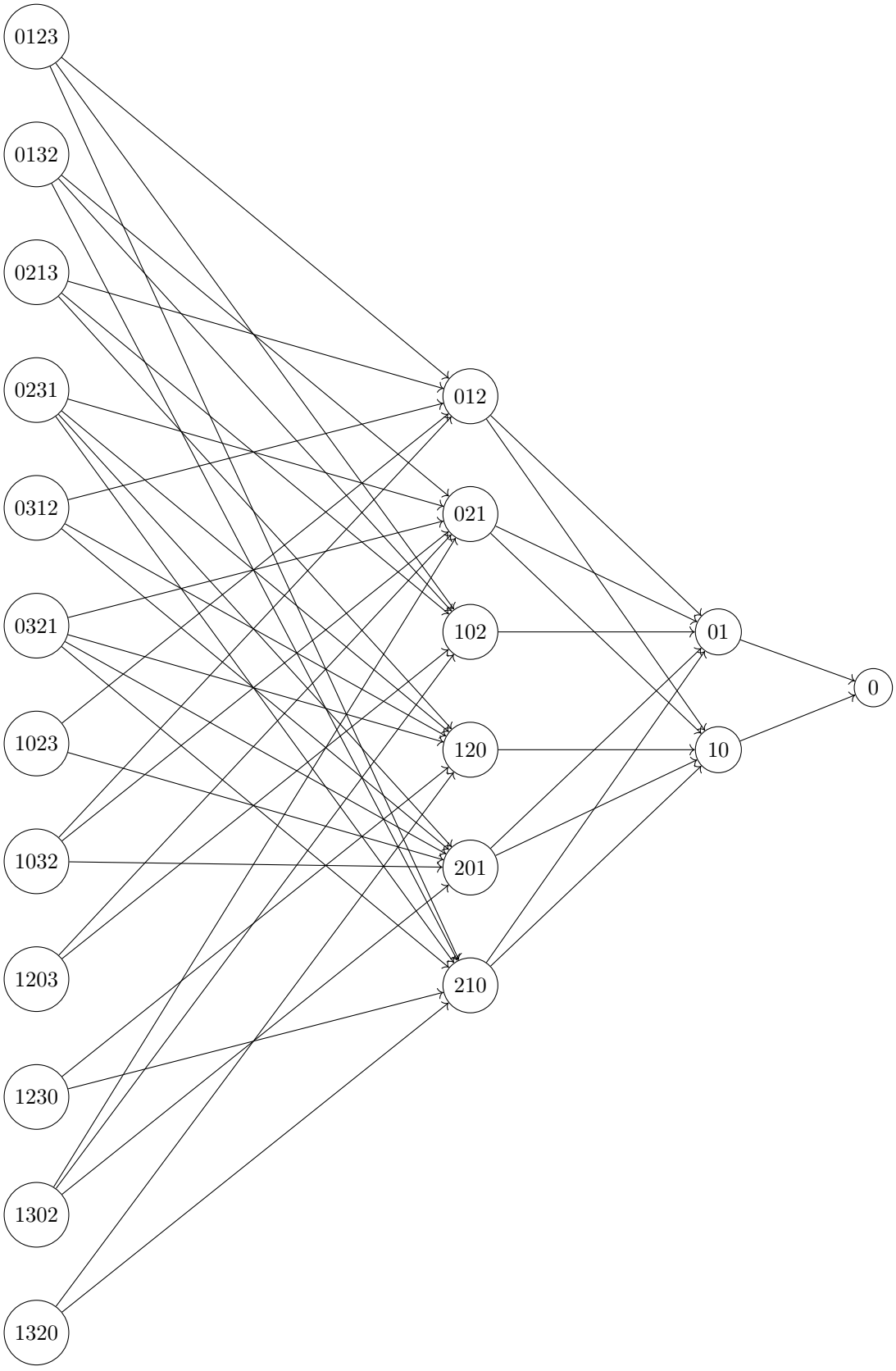
$$(\gamma_{i_{k-1}} \ldots (\gamma_{i_1}(\gamma_{i_0} p)) \ldots) = I^{n-k}$$

wird γ_i-Pancake Sort genannt. Das kleinstmögliche k wird als $A(p)$ bezeichnet.

1.1 Die Symmetrie des Pancake-Graphen

Im Kontext des ursprünglichen Pancake Sort-Problems ist der Pancake-Graph für Permutationen der Länge n wie folgt definiert: Für jede Permutation der Länge n gibt es genau einen Knoten, und zwischen den Knoten zweier Permutationen verläuft genau dann eine ungerichtete Kante, wenn sie durch das Umkehren eines Präfixes ineinander umgewandelt werden können. Analog dazu kann man einen Pancake-Graphen G_n für γ_i-Pancake Sort definieren: Jeder Permutation der Länge n oder kleiner wird ein Knoten zugeordnet, und zwischen zwei Permutationen p und q wird eine von p nach q gerichtete Kante eingefügt, wenn $\gamma_i p = q$, für irgendein $0 \leq i \leq |p| - 1$. G_n ist also ein gerichteter, azyklischer Graph (DAG) mit n "Ebenen", für jede Permutationslänge eine. Kanten verlaufen nur zur direkt folgenden Ebene, da γ_i die Länge um genau 1 reduziert. In Abbildung 1 ist G_4 dargestellt, aufgrund seiner Größe wurde er auf 2 Seiten aufgeteilt und Knoten der kleinern Permutationen dupliziert. Die Knoten sind jeweils in lexikographischer Ordung von oben nach unten angeordnet.

Betrachtet man die beiden Teile von G_4 gleichzeitig, fällt auf, dass der zweite Teil genau wie der erste Teil aussieht, nur vertikal gespiegelt. Auch die kleineren Pancake-Graphen G_3, G_2 und G_1, die Teilgraphen von G_4 sind, scheinen symmetrisch um ihre Mitte zu sein.


0123
0132
0213
0231
0312
0321
1023
1032
1203
1230
1302
1320
012
021
102
120
201
210
01
10
0

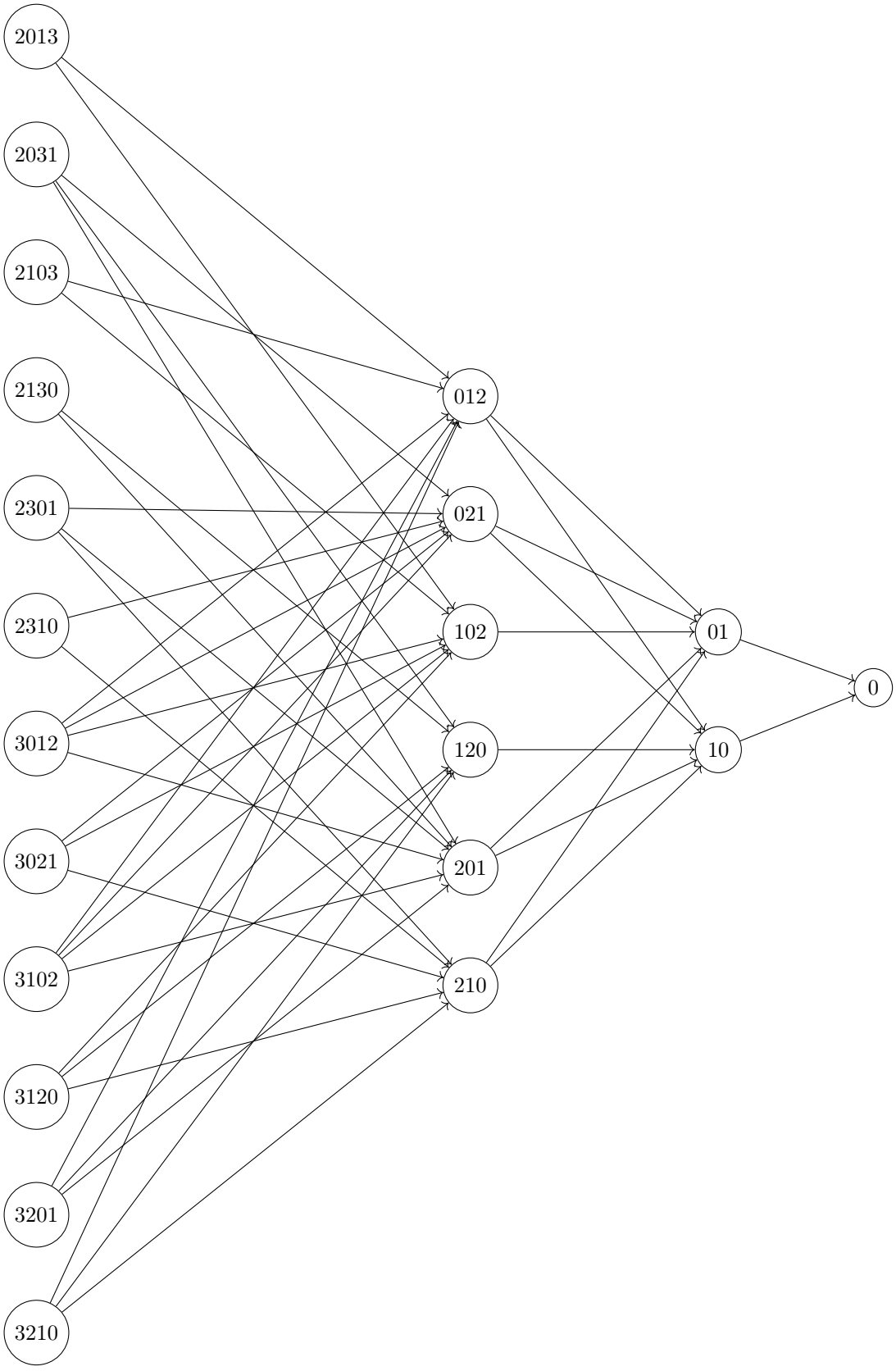

2013
2031
2103
2130
2301
2310
3012
3021
3102
3120
3201
3210
012
021
102
120
201
210
01
10
0

Abbildung 1: Der Pancake-Graph G_4. Von Knoten u verläuft genau dann eine gerichtete Kante nach Knoten v, wenn die zu u zugehörige Permutation durch ein γ_i in die zu v zugehörige Permutation umgewandelt werden kann.

Wenn man G_4 an einem Stück ohne Duplikation von Knoten zeichnen würde, wäre tatsächlich der gesamte Graph achsensymmetrisch um seine Mitte. Dass die Symmetrie bei G_1, G_2, G_3 und G_4 auftritt, lässt vermuten, dass sie für G_n, mit $n \geq 1$, allgemein gilt. Bei genauerer Betrachtung des Effekts von γ_i auf zwei zur Mitte symmetrisch liegende Permutationen fällt noch eine stärkere Eigenschaft auf. Wir betrachten als Beispiel die zwei symmetrisch liegenden Permutationen $p = 0, 2, 3, 1$ und $p^* = 3, 1, 0, 2$:

$$\begin{aligned} \gamma_0 p &= 1, 2, 0 & \gamma_0 p^* &= 1, 0, 2 \\ \gamma_1 p &= 0, 2, 1 & \gamma_1 p^* &= 2, 0, 1 \\ \gamma_2 p &= 2, 0, 1 & \gamma_2 p^* &= 0, 2, 1 \\ \gamma_3 p &= 2, 1, 0 & \gamma_3 p^* &= 0, 1, 2 \end{aligned}$$

Die Ergebnisse der jeweiligen Anwendung von γ_i liegen symmetrisch um die Mitte der Liste aller Permutationen von Länge 3. Tatsächlich ist das allgemein gültig, was im Folgenden bewiesen werden soll. Die Symmetrieeigenschaft erlaubt es, eine Optimierung bei der Lösung für Teilaufgabe b) vorzunehmen.

Für den Beweis sind allerdings einige weitere Mittel nötig. Zunächst wird das fakultätsbasierte Zahlensystem eingeführt, mithilfe dessen der γ_i-Operator aus einer anderen Perspektive betrachtet werden kann. Denn das fakultätsbasierte Zahlensystem erlaubt es, die Anzahl an Inversionen symmetrisch gelegener Paare von Permutationen miteinander in Verbindung zu bringen. Schließlich soll gezeigt werden, dass die Anwendung von γ_i auf zwei symmetrisch gelegene Permutationen wieder zu einem symmetrischen Paar führt.

Das fakultätsbasierte Zahlensystem. Im fakultätsbasierten Zahlensystem wird im Gegensatz zum Dezimal- oder Binärsystem eine unterschiedliche Basis für jede Ziffer verwendet. Die k-te Ziffer (mit 0 beginnend und von rechts gelesen) verwendet $k!$ als Basis und kann die Werte 0 bis k annehmen. Der Wert einer fakultätsbasiert geschriebenen Zahl ist die Summe der einzelnen Ziffern, multipliziert mit ihrer jeweiligen Basis. Beispielsweise ist

$$\begin{aligned} 17_{10} &= 2210_! = 2 \cdot 3! + 2 \cdot 2! + 1 \cdot 1! + 0 \cdot 0! \\ 24_{10} &= 10000_! = 1 \cdot 4! + 0 \cdot 3! + 0 \cdot 2! + 0 \cdot 1! + 0 \cdot 0! \\ 23_{10} &= 3210_! = 3 \cdot 3! + 2 \cdot 2! + 1 \cdot 1! + 0 \cdot 0! \end{aligned}$$

wobei das tiefgestellte „!" auf das fakultätsbasierte Zahlensystem hinweist. Eine Fakultät $n!$, geschrieben im fakultätsbasierten Zahlensystem, ist immer von der Form $1000\ldots$ (n Nullen). Die Ziffern von $n! - 1$ sind immer genau $n-1, n-2, n-3, \ldots, 0$. Da sich mit einer fakultätsbasierten Zahl mit n Ziffern genau $n!$ Zahlen darstellen lassen, können diese Zahlen zum Nummerieren von Permutationen der Länge n verwendet werden. Folgende zwei Arten der Nummerierung sind entscheidend für den Beweis der Symmetrie des Pancake-Graphen. Beide bilden eine Bijektion zwischen Permutationen der Länge n und ganzen Zahlen von 0 bis $n! - 1$.

Definition 3. Mit $\mu(p)$ wird der Index der Permutation p in einer *lexikographisch aufsteigend* sortierten Folge aller Permutationen der Länge $|p|$ bezeichnet. Mit $\mu(p)_i$ wird die i-te Ziffer (beginnend von links) von $\mu(p)$ bezeichnet, wenn $\mu(p)$ im fakultätsbasierten Zahlensystem geschrieben wird.

$\mu(p)$ ist auch als Lehmer-Code von p bekannt [6].

Definition 4. Mit $\nu(p)$ wird der Index der Permutation p in einer *kolexikographisch absteigend* sortierten Folge aller Permutationen der Länge $|p|$ bezeichnet. Mit $\nu(p)_i$ wird die i-te

Ziffer (beginnend von rechts) von $\nu(p)$ bezeichnet, wenn $\nu(p)$ im fakultätsbasierten Zahlensystem geschrieben wird.

Bei Sortierung nach kolexikographischer Ordung werden die Permutationen von rechts anstatt von links beginnend verglichen [7]. Man beachte, dass die Ziffern von $\mu(p)$ von links indexiert werden, wohingegen die Ziffern von $\nu(p)$ von rechts indexiert werden. Das vereinfacht die Notation später.

Für $\mu(p)$ und $\nu(p)$ gelten folgende Eigenschaften: $\mu(p)_i$ ist genau die Anzahl an kleineren Elementen rechts von p_i, und $\nu(p)_i$ die Anzahl an größeren Elementen links von p_i [6]. Mit $\mu(p)$ ist es nun möglich, die zu p symmetrisch gelegene Permutation zu definieren.

Definition 5. Sei p eine Permutation der Länge n. p^* bezeichnet die Permutation, sodass $\mu(p) + \mu(p^*) = n! - 1$.

p und p^* werden auch als symmetrisches Paar von Permutationen bezeichnet. Aus der Definition folgt direkt, dass $\mu(p)_i + \mu(p^*)_i = n - i - 1$. Ist das nicht der Fall, ist es leicht zu sehen, dass $\mu(p) + \mu(p^*)$ nicht $n! - 1$ sein kann. Zum Beweis der Symmetrie von G_n soll gezeigt werden, dass $\mu(\gamma_i p) + \mu(\gamma_i p^*) = (n-1)! - 1$ ist. Denn daraus folgt direkt, dass $\gamma_i p$ und $\gamma_i p^*$ wieder ein symmetrisches Paar von Permutationen ist. Nun wird folgender Zusammenhang zwischen $\nu(p)$ und $\nu(p^*)$ benötigt.

Lemma 1. $\nu(p) + \nu(p^*) = n! - 1$

Beweis. Die Aussage ist äquivalent zu $\nu(p)_j + \nu(p^*)_j = j$ für $0 \leq j \leq n-1$. Der Beweis geschieht durch Induktion über n. Für $n = 1$ stimmt die Aussage offensichtlich. Man nehme an, die Aussage stimmt für Permutationen der Länge k. Man nehme außerdem an, dass p eine Permutation der Länge k ist und die Zahl k vor Index i in p eingefügt wird. Durch Einfügen von k in p erhält man die Permutation

$$q = p_0, p_1, \ldots, p_{i-1}, k, p_i, p_{i+1}, \ldots, p_{k-1}$$

der Länge $k+1$. Wenn dieses Verfahren für alle Permutationen der Länge k und alle möglichen $0 \leq i \leq k$ durchgeführt wird, erhält man alle Permutationen der Länge $k+1$. Das Ziel ist es nun, zu zeigen, dass durch Einfügen von 0 in p^* und Erhöhen aller anderen Elemente um 1 q^* erzeugt wird, und dass nach dem Einfügen immer noch $\nu(q) + \nu(q^*) = (k+1)! - 1$ gilt. Auch hier ist es äquivalent zu zeigen, dass $\nu(q)_j + \nu(q^*)_j = j$ für alle $0 \leq j \leq k$ gilt.

Wir definieren die Permutation

$$x = p_0^* + 1, p_1^* + 1, \ldots, p_{i-1}^* + 1, 0, p_i^* + 1 p_{i+1}^* + 1, \ldots, p_{k-1}^* + 1$$

Für jedes j, sodass $i < j \leq k$, ist $\mu(x)_j = \mu(p^*)_{j-1}$, da p_j^* um eine Stelle verschoben wurde und sich die Anzahl rechts gelegener, kleinerer Elemente nicht ändert. Für $0 \leq j < i$ ist $\mu(x)_j = \mu(p^*)_j + 1$, da 0 rechts von j eingefügt wurde und sich die relative Ordnung sonst nicht ändert. Da $\mu(q)_i = k - i$, muss $\mu(q^*)_i = 0$ sein, denn $\mu(q)_i + \mu(q^*)_i = k - i$. $\mu(x)_i$ ist ebenfalls 0, da das Element 0 keine kleineren Elemente rechts von sich haben kann. Daher gilt für alle $0 \leq j \leq k$

$$\mu(q)_j + \mu(x)_j = k - j$$

und damit $x = q^*$.

Nun wird durch Fallunterscheidung gezeigt, dass $\nu(q)_j + \nu(q^*)_j = j$ für $0 \leq j \leq k$. Für $j = i$: Da $q_i = k$, ist $\nu(q)_i = 0$ (es gibt keine größeren, links gelegenen Elemente), und da $x = q^*$ ist $q_i^* = 0$ und damit $\nu(q^*)_i = i$, sodass $\nu(q)_i + \nu(q^*)_i = i$. Für $0 \leq j < i$ ist $\nu(q)_j = \nu(p)_j$ und $\nu(q^*)_j = \nu(p^*)_j$, da durch ν die größeren, links gelegenen Elemente gezählt werden, die für $0 \leq j < i$ gleich denen in p bzw. p^* sind. Damit gilt $\nu(q)_j + \nu(q^*)_j = j$ für $0 \leq j < i$ nach der Induktionsannahme. $\nu(q)_j = \nu(p)_{j-1} + 1$ für alle $i < j \leq k$, da die Elemente nach i durch das Einfügen von k um eine Stelle verschoben wurden und ein neues

größeres Element, k, links von ihnen liegt. Für $i < j \leq k$ ist $\nu(q^*)_j = \nu(p^*)_{j-1}$, da $q^*_i = 0$, sodass die Anzahl links gelegener, größerer Elemente unverändert bleibt. Folglich gilt auch für $i < j \leq k$

$$\nu(q)_j + \nu(q^*)_j = \nu(p)_{j-1} + 1 + \nu(p^*)_{j-1} = j - 1 + 1 = j$$

Aus $\nu(q)_j + \nu(q^*)_j = j$ für $0 \leq j \leq k$ folgt direkt $\nu(q) + \nu(q^*) = (k+1)! - 1$. □

Lemma 2. *Sei p eine Permutation der Länge n und p_i ein Element von p $(0 \leq i \leq n-1)$. Für jedes $0 \leq j \leq n-1, j \neq i$ ist entweder ($p_i < p_j$ und $p^*_i > p^*_j$) oder ($p_i > p_j$ und $p^*_i < p^*_j$).*

Beweis. In anderen Worten sagt das Lemma, dass die kleineren, rechts bzw. links gelegenen Elemente von p_i und p^*_i alle an unterschiedlichen Positionen liegen. Für jede Position j ist also entweder p_i und p_j oder p^*_i und p^*_j eine Inversion.

Der Beweis geschieht durch einen Widerspruch. Man nehme an, dass für irgendein $j_0 > i$ sowohl $p_{j_0} < p_i$, als auch $p^*_{j_0} < p^*_i$ gilt. Der Fall $j_0 < i$ funktioniert ähnlich. Auch der Fall $p_{j_0} > p_i$ und $p^*_{j_0} > p^*_i$ anstatt $p_{j_0} < p_i$ und $p^*_{j_0} < p^*_i$ kann mit der gleichen Methode bewiesen werden. Da $\nu(p)_{j_0} + \nu(p^*)_{j_0} = j_0$, muss für irgendein ein $j_1 < j_0$ gelten, dass $p_{j_1} < p_{j_0}$ und $p^*_{j_1} < p^*_{j_0}$. Andernfalls wäre es nicht möglich, auf insgesamt j_0 links gelegene, größere Elemente von p_{j_0} und $p^*_{j_0}$ zu kommen, denn sowohl p_i also auch p^*_i ist größer. Nun gibt es zwei Fälle:

1. $j_1 < i$: Rechts von j_1 liegen i und j_0. Das heißt, es muss zwei Indizes $j_2, j_3 > j_1$ geben, sodass $p_{j_2} < p_{j_1}$ und $p^*_{j_2} < p^*_{j_1}$, $p_{j_3} < p_{j_1}$ und $p^*_{j_3} < p^*_{j_1}$. Andernfalls wäre es nicht möglich, auf die nötigen $\mu(p)_{j_1} + \mu(p^*)_{j_1} = n - j_1 - 1$ nötigen, kleineren, rechts gelegenen Elemente zu kommen. Denn bereits zwei der $n - j_1 - 1$ rechts gelegenen Plätze sind sowohl in p als auch in p^* durch größere Zahlen besetzt, aber $n - j_1 - 1$ kleinere Elemente sind erforderlich.

2. $i < j_1 < j_0$: Da $p_{j_1} < p_{j_0}$ und $p^*_{j_1} < p^*_{j_0}$, muss es rechts von j_1 einen Index $j_2 > j_1$ geben, sodass $p_{j_2} < p_{j_1}$ und $p^*_{j_2} < p^*_{j_1}$. Auch links von j_1 muss es einen Index $j_3 < j_1$ geben, sodass $p_{j_3} < p_{j_1}$ und $p^*_{j_3} < p^*_{j_1}$. Erneut wäre andernfalls das Erreichen der nötigen $\mu(p)_{j_1} + \mu(p^*)_{j_1} = n - j_1 - 1$ rechts gelegenen, kleineren und der $\nu(p)_{j_1} + \nu(p^*)_{j_1} = j_1$ links gelegenen, größeren Elemente unmöglich.

Man sieht, dass immer kleinere Zahlen in p und p^* gezwungen werden. Um diese herum sind aber nur größere Elemente, wodurch wieder kleinere Zahlen zum Ausgleich nötig werden. Dieser Prozess endet niemals, da mit jedem Schritt immer noch kleinere Zahlen erzeugt werden. Das ist ein Widerspruch, da ganze Zahlen größer gleich 0, wie sie in einer Permutation vorkommen, nicht unendlich oft verringert werden können. Das zeigt, dass die Annahme $p_{j_0} < p_i$ und $p^*_{j_0} < p^*_i$ falsch war. □

Für eine Permutation p und $0 \leq i, j, k \leq n-1$ wird nun die Funktion $\eta(p, i, j, k)$ definiert.

$$\eta(p, i, j, k) = \sum_{h=j}^{k} \begin{cases} 1 & \text{wenn } p_i > p_h \\ 0 & \text{wenn } p_i \leq p_h \end{cases}$$

$\eta(p, i, j, k)$ zählt die Anzahl an Elementen mit Indizes in $[j, k]$, die kleiner als p_i sind. Lemma 2 ermöglicht es, die Summe von η für p und p^* einfach zu berechnen, denn für jeden Index

ist entweder das Element in p oder in p^* kleiner als p_i bzw. p_i^*.

$$\begin{aligned}\eta(p,i,j,k)+\eta(p^*,i,j,k) &= \sum_{h=j}^{k}\begin{cases}1 & \text{wenn } p_i > p_h\\ 0 & \text{wenn } p_i \le p_h\end{cases} + \sum_{h=j}^{k}\begin{cases}1 & \text{wenn } p_i^* > p_h^*\\ 0 & \text{wenn } p_i^* \le p_h^*\end{cases}\\ &= \sum_{h=j}^{k}\left(\begin{cases}1 & \text{wenn } p_i > p_h\\ 0 & \text{wenn } p_i \le p_h\end{cases} + \begin{cases}1 & \text{wenn } p_i^* > p_h^*\\ 0 & \text{wenn } p_i^* \le p_h^*\end{cases}\right)\\ &= \sum_{h=j}^{k} 1\\ &= k-j+1\end{aligned}$$

Damit kann nun gezeigt werden, dass das Umkehren von Präfixen zweier symmetrisch gelegener Permutationen erneut zu zwei symmetrisch gelegene Permutationen führt. Das ist fast das gewünschte Ergebnis, es muss lediglich noch das erste Element aus beiden Permutationen entfernt werden.

Lemma 3. *Seien x und y die Permutationen, die aus p und p^* durch Umkehrung des Präfixes bis einschließlich Index i hervorgehen. Es gilt $\mu(x)+\mu(y) = n!-1$.*

Beweis. Die Ziffern $\mu(p)$ und $\mu(p^*)$ geben in fakultätsbasierter Schreibweise die Anzahl an rechts gelegenen, kleineren Elementen an. Da an den Elementen p_j, p_j^* für $j > i$ nichts geändert wird, ändert sich auch nicht ihre Zahl rechts gelegener, kleinerer Elemente. Für das j-te Element mit $j \le i$ werden alle kleineren Elemente von Index $0 \dots j-1$ durch die Umkehrung auf die rechte Seite gebracht, die kleineren Elemente mit Indizes $i+1 \dots n-1$ bleiben. Der Index, zu dem das j-te Element durch die Umkehrung bewegt wird, ist $i-j$. Folglich ist die Anzahl rechts gelegener, kleinerer Elemente von x_{i-j} und y_{i-j} zusammen

$$\begin{aligned}\mu(x)_{i-j}+\mu(y)_{i-j} &= \eta(p,j,0,j-1)+\eta(p,j,i+1,n-1)\\ &\quad +\eta(p^*,j,0,j-1)+\eta(p^*,j,i+1,n-1)\\ &= \eta(p,j,0,j-1)+\eta(p^*,j,0,j-1)\\ &\quad +\eta(p,j,i+1,n-1)+\eta(p^*,j,i+1,n-1)\\ &= (j-1-0+1)+(n-1-(i+1)+1)\\ &= j+n-i-1\\ &= n-(i-j)-1\end{aligned}$$

Daraus folgt

$$\mu(x)_i+\mu(y)_i = n-i-1 \quad 0 \le i \le n-1$$

und damit

$$\mu(x)+\mu(y) = n!-1$$

□

Man sieht, dass das Umkehren eines Präfixes in einem symmetrischen Paar von Permutationen wieder zu einem symmetrischen Paar führt. Nun kann die Symmetrie des Pancake-Graphen, bzw. des γ_i-Operators als Satz festgehalten werden.

Satz 1. *Wenn p eine Permutation der Länge n und q eine Permutation der Länge $n-1$ ist, gilt*

$$\gamma_i p = q \iff \gamma_i p^* = q^* \quad 0 \le i \le n-1$$

Beweis. Wir nennen die Permutation, die man durch Umkehren des Präfixes bis i von p erhält, x. Wie in Lemma 3 gezeigt, ist x^* genau die Permutation, die man durch Umkehren des Präfixes bis i von p^* erhält. Da

$$\gamma_i p = x_1, x_2, \ldots, x_{n-1}$$
$$\gamma_i p^* = x_1^*, x_2^*, \ldots, x_{n-1}^*$$

entspricht die Anzahl rechts gelegener, kleinerer Elemente von $(\gamma_i p)_0, (\gamma_i p)_1, \ldots, (\gamma_i p)_{n-2}$ bzw. $(\gamma_i p^*)_0, (\gamma_i p^*)_1, \ldots, (\gamma_i p^*)_{n-2}$ genau der Anzahl rechts gelegener, kleinerer Elemente von $x_1, x_2, \ldots, x_{n-1}$ bzw. $x_1^*, x_2^*, \ldots, x_{n-1}^*$. Daher gilt

$$\mu(\gamma_i p)_j = \mu(x)_{j+1}$$
$$\mu(\gamma_i p^*)_j = \mu(x^*)_{j+1}$$

für $0 \leq j \leq n-2$ und folglich

$$\begin{aligned}\mu(\gamma_i p)_j + \mu(\gamma_i p^*)_j &= \mu(x)_{j+1} + \mu(x^*)_{j+1} \\ &= (n - (j+1) - 1) \\ &= (n-1) - j - 1\end{aligned}$$

Daraus folgt direkt, dass $\mu(\gamma_i p) + \mu(\gamma_i p^*) = (n-1)! - 1$. $\gamma_i p$ und $\gamma_i p^*$ ist also ein symmetrisches Paar von Permutationen der Länge $n-1$. $\gamma_i p = q$ gilt also genau dann, wenn $\gamma_i p^* = q^*$. □

1.2 Reduktion des Burnt Pancake-Problems auf γ_i-Pancake Sort

Die später vorgestellten Algorithmen zur Lösung beider Teilaufgaben werden mit der Annahme entwickelt, dass γ_i-Pancake Sort NP-schwer ist. Dafür wurde kein Beweis gefunden, aber es konnte gezeigt werden, dass γ_i-Pancake Sort unter Polynomialzeitreduktion mindestens so schwierig wie das Burnt Pancake-Problem ist. Dessen NP-Schwere ist nicht bewiesen, allerdings ist es seit der Vorstellung von Gates und Papadimitriou 1979 [4] niemandem gelungen, einen Algorithmus mit polynomieller Laufzeit dafür zu finden. Es ist also durchaus eine Rechtfertigung, anschließend keine Algorithmen mit polynomieller Laufzeit vorzustellen.

Im Burnt Pancake Problem haben Pfannkuchen eine verbrannte Seite, die nach dem Sortieren bei jedem Pfannkuchen unten liegen muss [3]. Das Konzept der verbrannten Seite wird in folgender Definition durch ein Vorzeichen vor jedem Element der Permutation formalisiert.

Definition 6 (Burnt Pancake-Problem)**.** Gegeben sei eine vorzeichenbehaftete Permutation

$$p = \sigma_0 p_0, \sigma_1 p_1, \ldots, \sigma_{n-1} p_{n-1}$$

wobei $0 \leq p_i \leq n-1$ und $\sigma_i \in \{-1, 1\}$ für $0 \leq i \leq n-1$ sowie $p_i \neq p_j$ für $i \neq j$. Was ist die minimale Anzahl an Präfixumkehrungen, wobei bei einer Präfixumkehrung auch alle Vorzeichen im umgekehrten Präfix invertiert werden, sodass p in I^n überführt wird und alle Vorzeichen positiv sind?

$\sigma_i = -1$ bedeutet, dass der i-te Pfannkuchen die verbrannte Seite oben hat, andernfalls liegt die unverbrannte Seite oben. (Hier zeigt sich ein kleiner Nachteil, die Elemente von p mit 0 beginnen zu lassen, denn wir müssen 0 und -0 unterscheiden.)

Satz 2. *Das Burnt Pancake-Problem kann in polynomieller Zeit auf γ_i-Pancake Sort reduziert werden.*

Beweis. Das Burnt Pancake-Problem wird in γ_i-Pancake Sort simuliert. Für eine vorzeichenbehaftete Permutation p der Länge n, die Eingabe für das Burnt Pancake-Problem, konstruieren wir eine nicht-vorzeichenbehaftete Permutation q der Länge $3n^2$ wie folgt:

$$q = a_0, a_1, \ldots, a_{n-1}$$

$$a_i = \begin{cases} p_i \cdot 3n, p_i \cdot 3n + 1, \ldots, (p_i + 1) \cdot 3n - 1 & \text{wenn } \sigma_i = 1 \\ (p_i + 1) \cdot 3n - 1, (p_i + 1) \cdot 3n - 2, \ldots, p_i \cdot 3n & \text{wenn } \sigma_i = -1 \end{cases}$$

Jedem Pfannkuchen in p wird je nach seiner Orientierung eine aufsteigende oder absteigende Folge von $3n$ aufeinanderfolgenden Zahlen zugeordnet. Dieses Vorgehen ist in Abbildung 2 veranschaulicht. Mit a_i wird die zu p_i zugehörige Teilfolge von q bezeichnet. Der Einfachheit halber bezeichnet a_i auch die zu p_i zugehörige Teilfolge in q nach einigen Sortierschritten, auch wenn diese dann andere Zahlen enthalten kann bzw. gekürzt worden sein kann. Die Folgen $a_0, a_1, \ldots, a_{n-1}$ in Zeile 1 werden zu einer Folge q zusammengefügt. Die Länge von q beträgt $n \cdot 3n = 3n^2$, ist also durch ein Polynom in n beschränkt, d. h. die Reduktion ist eine Polynomialzeitreduktion.

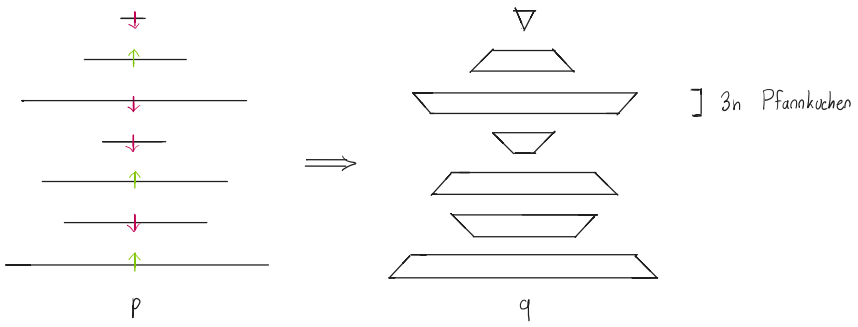


Abbildung 2: Reduktion des Burnt Pancake-Problems auf γ_i-Pancake Sort. Die Pfeile geben die Orientierung der verbrannten Pfannkuchen an, die Pfeilspitze zeigt zur nicht verbrannten Seite.

Ist $\sigma_i = 1$, d. h. der i-te Pfannkuchen ist richtig orientiert, wird er durch eine aufsteigende Folge repräsentiert. Das entspricht auch der richtigen Orientierung im Sortieren ohne Vorzeichen. Denn wäre p vollständig sortiert und $\sigma_i = 1$ für alle $0 \leq i \leq n - 1$, wäre auch q vollständig sortiert. Die Idee ist nun, zu zeigen, dass kein a_i in einer optimalen Folge an γ_i-Operationen in der Mitte getrennt oder vollständig aufgegessen wird. Denn dann werden durch γ_i-Operationen immer nur Pfannkuchen am Anfang und Ende einer der Folgen a_i entfernt, sodass sich diese schließlich wie ein verbrannter Pfannkuchen verhalten. Durch die $3n$ Pufferelemente bleiben von jedem a_i außerdem mindestens n Elemente übrig, sodass das Ergebnis des Sortierens mit γ_i als sortierter Stapel an verbrannten Pfannkuchen interpretiert werden kann.

Eine Folge a_i wird niemals durch eine optimale Folge an γ_i-Operationen vollständig entfernt, da eine vorzeichenbehaftete Permutation in maximal $2n$ Schritten sortiert werden kann. Diese obere Schranke stammt von Cohen und Blum [3]. Für $n \geq 10$ wurde dort sogar eine Schranke von $2n-2$ angegeben, der Einfachheit halber wird im Folgenden allerdings mit $2n$ gearbeitet. Daraus folgt, dass eine Folge an γ_i-Operationen, die länger als $2n$ ist, nicht optimal sein kann, denn man könnte q in maximal $2n$ Schritten sortieren, indem man eine optimale Folge an Präfixumkehrungen für p auf q anwendet, und dabei die i-te Teilfolge a_i wie einen verbrannten Pfannkuchen behandelt. Wenn in p das Präfix bis Index i umgekehrt wird, wendet man γ_j auf q an, wobei j die größtmöglich Zahl ist, sodass q_j zur Teilfolge von

p_i gehört (unter Beachtung der Tatsache, dass durch jede Anwendung von γ_i möglicherweise einige Elemente in q verringert werden). Da jedes Element von p nach der Sortierung ein positives Vorzeichen hat, sind nach der Definition von q auch alle Teilfolgen a_i für $0 \leq i \leq n-1$ aufsteigend sortiert. Da die Pfannkuchen in p aufsteigend sortiert sind, sind auch die ihnen zugeordneten Teilfolgen in q aufsteigend sortiert, daher lässt sich q tatsächlich in maximal $2n$ Schritten sortieren. Da also maximal $2n$ γ_i-Operationen durchgeführt werden und mit jeder γ_i-Operation genau ein Element aus q entfernt wird, muss jedes a_i nach dem Sortieren noch mindestens n Elemente enthalten.

Nun wird gezeigt, dass es unter den optimalen γ_i-Folgen zum Sortieren von q immer eine gibt, die im Sortierprozess keines der a_i in der Mitte trennt. Das ist notwendig, da ein solcher Schritt, übertragen auf das Burnt Pancake-Problem, unmöglich wäre. Man nehme an, a_j für $0 \leq j \leq n-1$ wird durch γ_k mit $l+1 \leq k \leq r-1$ in zwei Teile geteilt, wobei a_j vor der Anwendung von γ_k in q von q_l bis q_r reichte. Da nach vollständiger Sortierung von q noch Elemente von a_j vorhanden sein müssen, müssen die zwei Teile an einem bestimmten Punkt wieder in richtiger Orientierung zusammengefügt werden. Hätte man anstatt γ_k γ_r verwendet, kann man in maximal der gleichen Anzahl an Schritten zum gleichen Ergebnis gelangen, indem man $q_{k+1}, q_{k+2}, \ldots, q_r$ dauerhaft mit $q_l, q_{l+1}, \ldots, q_k$ mitführt. Das heißt, man führt alle Wendeoperationen hinter den gleichen a_i wie zuvor durch, behandelt $q_l, q_{l+1}, \ldots, q_r$ aber genau so wie zuvor $q_l, q_{l+1}, \ldots, q_{k-1}$. Der genaue Index der γ_i-Operationen mag sich unterscheiden, da in jedem Schritt aber die gleichen a_i umgekehrt werden, ist das Ergebnis ebenfalls ein sortierter Stapel. Die Argumentation ist in Abbildung 3 veranschaulicht. Durch die Veränderung wird die Gesamtzahl an γ_i-Operationen nicht vergrößert, daher gibt es immer eine optimale Folge an γ_i-Operationen ohne Trennung einer Folge a_i in der Mitte.

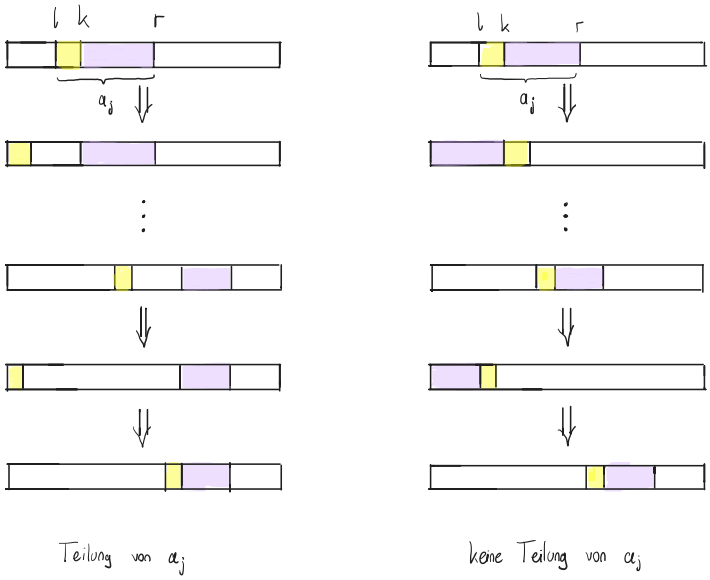


Abbildung 3: Eine optimale Folge an γ_i-Operationen kann immer ohne die Teilung von a_i in der Mitte auskommen.

Damit verhält sich eine Teilfolge, die einem verbrannten Pfannkuchen zugeordnet wurde, im Sortierprozess wie ein verbrannter Pfannkuchen. Denn sie kann weder vollständig verschwinden oder geteilt werden und muss am Ende, wie der verbrannte Pfannkuchen,

richtig orientiert an der richtigen Stelle platziert sein, damit q sortiert ist. Folglich ist die minimale Anzahl an γ_i-Operationen zum Sortieren von q gleich der minimalen Anzahl an Wendeoperationen zum Sortieren von p. Da jede Instanz des Burnt Pancake-Problems in polynomieller Zeit auf eine Instanz von γ_i-Pancake Sort reduziert werden kann, ist γ_i-Pancake Sort mindestens so schwierig wie das Burnt Pancake-Problem. □

Anmerkung. Von dem normalen (unburnt) Pancake-Problem ohne Aufessen ist die NP-Schwere bewiesen [2]. Eine Reduktion von diesem in ähnlicher Weise durchzuführen ist aber nicht möglich, da man den Teilfolgen keine Richtung geben dürfte. Man bräuchte also mehrere Pfannkuchen gleicher Größe - und würde damit keine gültige Eingabe für γ_i-Pancake Sort erhalten.

1.3 Finden der kürzesten Folge an γ_i-Operationen

In diesem Abschnitt werden drei Lösungsverfahren für Teilaufgabe a) entwickelt. Zunächst wird ein Brute-Force Ansatz vorgestellt, der bereits alle vorgegebenen Beispiele gut lösen kann. Anschließend wird eine untere Schranke für die nötige Anzahl an γ_i-Operationen zum Erreichen einer identischen Permutation hergeleitet, mit der sich der Brute-Force Algorithmus auf zwei Arten verbessern lässt.

Brute-Force Lösung. Es werden alle Möglichkeiten, eine γ_i-Operation durchzuführen, ausgetestet, also $\gamma_0, \gamma_1, \ldots, \gamma_{n-1}$. Von den entstehenden Permutationen wird das Verfahren wiederholt, bis eine identische Permutation erreicht ist. Alle Permutation der aktuellen Länge werden vollständig bearbeitet, bevor zu den Permutationen der nächstkleinerern Länge vorgedrungen wird. Das Verfahren entspricht also einer Breitensuche auf dem Pancake-Graphen, wobei nach dem kürzesten Pfad zu einer identischen Permutation gesucht wird. Da der Pancake-Graph ungewichtet ist, kann Breitensuche zum Finden kürzester Pfade verwendet werden.

Der Brute-Force Algorithmus ist im Pseudocode MinOperationsBFS zusammengefasst. Wir definieren eine Warteschlange Q, die alle noch nicht besuchten Permutationen enthält, die von einer besuchten Permutation durch eine γ_i-Operation erreicht werden können. Q enthält zu Beginn nur p. Um Speicher zu sparen, wird in der Warteschlange nicht die gesamte Permutation gespeichert, sondern lediglich ihre Länge und ihr μ-Wert, wodurch sie eindeutig bestimmt ist. Wie $\mu(p)$ berechnet und p aus $\mu(p)$ rekonstruiert werden kann, wird später diskutiert. Die Funktion, die eine Permutation s zurückgibt, sodass $\mu(s) = i$ und $|s| = n$, wird im Folgenden einfach mit IthPermutation(n, i) bezeichnet. Daneben wird ein Array an Hashmaps *pre* angelegt, in dem der Index der Vorgängerpermutation für jede besuchte Permutation gespeichert wird. Das ist zur Rekonstruktion der durchgeführten Operationen nötig. In *pre* befinden sich in der Hashmap bei Index i die Vorgänger aller Permutationen der Länge $i + 1$. In einer Iteration der while-Schleife werden für das erste Element der Warteschlange s alle $|s|$ möglichen γ_i-Operationen durchgeführt und die Permutationen $\gamma_i s$ für $0 \leq i \leq |s|$, die noch nicht besucht wurden, zu Q hinzugefügt. Beim Erreichen einer identischen Permutation wird ReconstructOperations aufgerufen und dessen Rückgabewert zurückgegeben.

Rekonstruktion der γ_i-Operationen. Um die durchgeführten γ_i-Operationen aus *pre* zu rekonstruieren, wird der aus der Breitensuche entstehende Baum von unten nach oben durchlaufen. Das Verfahren ist im Algorithms ReconstructOperations dargestellt. *operations* enthält die i aller γ_i-Operationen, die bereits rekonstruiert wurden. In der folgenden while-Schleife wird in einer Iteration jede mögliche γ_i-Operation auf dem Vorgänger s der aktuellen Permutation ausprobiert. Wenn $\mu(\gamma_i s) = \mathit{index}$, wurde die richtige Operation gefunden und wird an *operations* angefügt.

Berechnung von $\mu(p)$. Die Algorithmen zur Berechnung von $\mu(p)$ und Rekonstruktion von p aus $\mu(p)$ werden von Bonet [1] übernommen, eine formale Beschreibung ist dort zu finden. In dem Artikel wurden mehrere Verfahren zur Abbildung von Permutationen auf

Algorithmus 1 : MinOperationsBFS(p)

```
Q ← Queue containing (μ(p), |p|)
pre ← Array of Hashmaps of size |p|
while Q ≠ ∅ do
    (index, length) ← Dequeue(Q)
    s ← IthPermutation(length, index)
    if index = 0 then
        return ReconstructOperations(pre, length, index)
    for j ← 0 to length − 1 do
        if μ(γ_j s) ∉ pre[length − 2] then
            pre[length − 2][μ(γ_j s)] ← index
            Enqueue(Q, (μ(γ_j s), length − 1))
```

Algorithmus 2 : ReconstructOperations(*pre, length, index*)

```
operations ← empty Array
while length < n do
    s ← IthPermutation(length + 1, pre[length − 1][index])
    for i ← 0 to length + 1 do
        if μ(γ_i s) = index then
            append i to the front of operations
            break
    index ← pre[length − 1][index]
    length ← length + 1
return operations
```

Zahlen und umgekehrt vorgestellt. Einige laufen in linearer Zeit, sind aber auf große, vorher erstellte Datenstrukturen angewiesen oder funktionieren nur bis zu einer bestimmten Länge. Es werden die zwei Verfahren verwendet, die in $\Theta(n \log n)$ Zeit laufen, ohne vorher berechnete Datenstrukturen auskommen, und für jede Permutationslänge verwendet werden können.

Zur Berechnung von $\mu(p)$ wird wieder das fakultätsbasierte Zahlensystem zur Hilfe gezogen, indem zunächst die Ziffern von $\mu(p)$ berechnet werden. Dafür wird folgende Eigenschaft verwendet: $\mu(p)_i$ entspricht genau p_i minus der Anzahl links gelegener, kleinerer Elemente in p [1]. (Man beachte, dass die Elemente von p mit 0 beginnen.) Damit reduziert sich das Problem darauf, die Anzahl kleinerer, links gelegener Elemente von p_i zu zählen. Das kann mit einem Segmentbaum in $\Theta(\log n)$ Zeit pro Element gelöst werden, sodass sich insgesamt eine Zeitkomplexität von $\Theta(n \log n)$ ergibt.

Für IthPermutation wird ähnlich vorgegangen. Zunächst werden die Ziffern von $\mu(p)$ im fakultätsbasierten Zahlensystem durch wiederholtes Teilen mit Rest bestimmt. Wie bei der Berechnung der Ziffern einer Zahl im Binärsystem sind die Ziffern die Reste der Folge von Divisionen, nur dass der Divisor hier nicht konstant ist, sondern zuerst 1, dann 2, 3 und so weiter. Die Indizes von p werden anschließend aufsteigend bearbeitet, man nehme also im Folgenden an, dass p_i gerade bestimmt werden soll und p_j für $j < i$ bereits bekannt ist. Mit der Tatsache, dass $\mu(p)_i$ genau p_i minus der Anzahl kleinerer, links gelegener Elemente ist, reduziert sich das Problem darauf, zu bestimmen, welchen Wert p_i annehmen muss, sodass es genau $\mu(p)_i$ kleinere Elemente links davon gibt. Auch das lässt sich mit einem Segmentbaum in $\Theta(\log n)$ Zeit pro Element lösen. Für ihn gilt folgende Invariante: Ist ein Element y noch nicht aufgetreten, steht eine 1 bei Index y, andernfalls eine 0. Zur Bestimmung von p_i wird der Segmentbaum von oben nach unten heruntergelaufen und die Schritte so gewählt, dass die Summe strikt links gelegener Elemente genau $\mu(p)_i$ ist. Dadurch landet man genau bei dem p_i-ten Element des Segmentbaums, denn links liegen $\mu(p)_i$ Einsen plus die Anzahl

bereits aufgetretener, kleinerer Elemente, die im Segmentbaum bereits auf 0 gesetzt wurden. Nun wird auch p_i im Segmentbaum auf 0 gesetzt und so die Invariante erhalten. Auch hier ist die Zeitkomplexität $\Theta(n \log n)$.

Eine Unterschranke für $A(p)$. Mit einer Unterschranke für die Anzahl an benötigten γ_i-Operationen ist es möglich, während der Suche manche Permutationen außer Acht zu lassen, da sie nicht zu einem besseren Ergebnis führen können. Wir betrachten die Anzahl an monoton steigenden oder fallenden Teilstrings in einer Permutation. Ein monoton steigender Teilstring T von Länge k einer Permutation p ist eine Folge an Indizes $i, i+1, \ldots, i+k-1$, so, dass $p_j < p_{j+1}$ für $i \leq j \leq i+k-2$. Daneben gilt für einen monoton steigenden Teilstring $p_{i-1} > p_i$ oder $i = 0$, und $p_{i+k-1} > p_{i+k}$ oder $i+k = n$, d. h. er kann nicht links oder rechts erweitert werden. Ein monton fallender Teilstring ist ähnlich definiert, nur gilt $p_j > p_{j+1}$ für $i \leq j \leq i+k-2$, sowie $p_{i-1} < p_i$ oder $i = 0$, und $p_{i+k-1} < p_{i+k}$ oder $i+k = n$. Ein monotoner Teilstring ist ein monoton steigender oder monoton fallender Teilstring. Nach dieser Definition ist ein Element von p, dessen Nachbarn beide größer oder kleiner sind, Teil von genau 2 monotonen Teilstrings. Die identische Permutation besteht nur aus einem monoton steigenden Teilstring. Mithilfe von folgendem Lemma kann eine untere Schranke für $A(p)$ bestimmt werden.

Lemma 4. *Pro γ_i-Operation kann die Anzahl monotoner Teilstrings einer Permutation p maximal um 3 reduziert werden.*

Beweis. Die Anzahl monotoner Teilstrings in p wird im Folgenden x genannt. Die Veränderung von x durch eine γ_i-Operation wird Δx genannt. Die Zahl monotoner Teilstrings kann nur durch Hinzufügen bzw. Entfernen von Teilstrings, die p_0 oder p_i enthalten, verändert werden. Wir unterscheiden zwei Fälle: Entweder ist p_i Teil von zwei oder von einem monotonen Teilstring.

Im ersten Fall werden wieder einige Fälle unterschieden. Zunächst wird nur der Beitrag zu Δx der zwei Teilstrings, die p_i enthalten, betrachtet. Wenn p_i, p_{i+1} ein monotoner Teilstring von Länge 2 ist, wird er durch γ_i entfernt, folglich ist sein Beitrag zu Δx -1. Ist $p_i, p_{i+1}, p_{i+2}, \ldots$ ein längerer, monotoner Teilstring bleibt er erhalten. Das gleiche Argument gilt für p_{i-1}, p_i. Folglich ist der Beitrag zu Δx von p_i $0, -1$ oder -2. Für den Beitrag von p_0 werden folgende Fälle unterschieden.

1. $p_0 < p_{i+1}$
   - 1.1. $p_0 < p_1$
     - 1.1.1. $p_{i+1} < p_{i+2} \implies \Delta x = 0$
     - 1.1.2. $p_{i+1} > p_{i+2} \implies \Delta x = 1$
   - 1.2. $p_0 > p_1$
     - 1.2.1. $p_{i+1} < p_{i+2} \implies \Delta x = -1$
     - 1.2.2. $p_{i+1} > p_{i+2} \implies \Delta x = 0$
2. $p_0 > p_{i+1}$ Der Fall ist symmetrisch zu $p_0 < p_{i+1}$.

Man sieht, dass Δx insgesamt nicht kleiner als -3 sein kann. Für den Fall, dass p_i Teil von einem monotonen Teilstring ist, muss nur der Beitrag von p_i neu betrachtet werden. Da aber bereits festgestellt wurde, dass nur Teilstrings, die p_0 oder p_i enthalten, verschwinden können und p_i nur Teil von einem monotonen Teilstring ist, gilt hier $\Delta x \geq -2$, sodass in allen Fällen $\Delta x \geq -3$ ist.

Bisher wurde implizit angenommen, dass $1 \leq i \leq n-3$ gilt, da p_{i-1}, p_{i+1} und p_{i+2} in Betracht gezogen wurden. Wenn $i = 0$ oder $i = n-1$, gilt offensichtlich $\Delta x \in \{0, -1\}$. Wenn $i = n-2$, folgt aus der gleichen Fallunterscheidung wie oben $\Delta x \geq -2$. □

Satz 3. *Sei x die Anzahl monotoner Teilstrings einer Permutation p. Dann gilt*

$$A(p) \geq \left\lfloor \frac{x+1}{3} \right\rfloor$$

Beweis. Die identische Permutation besitzt genau einen monotonen Teilstring, d. h. durch die γ_i-Operationen müssen insgesamt $x-1$ monotone Teilstrings entfernt werden. Wenn $x-1$ durch 3 teilbar ist, sind dafür nach Lemma 3 mindestens

$$\frac{x-1}{3} = \frac{x-1}{3} + \left\lfloor \frac{2}{3} \right\rfloor = \left\lfloor \frac{x-1}{3} + \frac{2}{3} \right\rfloor = \left\lfloor \frac{x+1}{3} \right\rfloor$$

Operationen nötig. Wenn $x-1 \equiv 1 \bmod 3$, ist neben den mindestens $(x-2)/3$ Operationen mit $\Delta x = -3$ noch mindestens eine weitere Operation nötig. Insgesamt ergibt sich eine Unterschranke von

$$\frac{x-2}{3} + 1 = \frac{x-2}{3} + \frac{3}{3} = \frac{x+1}{3}$$

Für $x-1 \equiv 2 \bmod 3$ sind mindestens $(x-3)/3$ Operationen mit $\Delta x = -3$ und eine weitere erforderlich.

$$\frac{x-3}{3} + 1 = \frac{x}{3} = \left\lfloor \frac{x+1}{3} \right\rfloor \quad (\text{da } x \equiv 0 \bmod 3)$$

□

Die untere Schranke kann mit dem Algorithmus LowerBound berechnet werden. x ist die aktuelle Zahl monotoner Teilstrings, *incr* gibt an, ob aktuell ein steigender Teilstring vorliegt. Für jedes Element von p wird überprüft, ob sich das Steigungsverhalten bei ihm ändert, und x wird entsprechend erhöht. Die Laufzeit von LowerBound beträgt $\Theta(n)$.

Algorithmus 3 : LowerBound(p)

```
if n = 1 then
    return 0
x ← 1
incr ← p_1 > p_0
for i ← 2 to n − 1 do
    if (incr ∧ (p_{i−1} > p_i)) ∨ (¬incr ∧ (p_{i−1} < p_i)) then
        incr ← ¬incr
        x ← x + 1
return ⌊(x + 1)/3⌋
```

Ein A-basiertes Verfahren.* Mit der unteren Schranke wird das Brute-Force Verfahren nun in zwei Aspekten verbessert.

1. Anstatt der Warteschlange wird eine Prioritätswarteschlange verwendet, in der die Knoten aufsteigend nach unterer Schranke geordnet sind. Damit werden vielversprechende Pfade früher besucht.

2. Die kürzeste bisher gefundene Distanz zu einer identischen Permutation wird ständig gespeichert, um das Besuchen unnötiger Knoten zu vermeiden. Ist die untere Schranke eines Knotens größer oder gleich der aktuell kürzesten Distanz, kann dieser nicht zu einem besseren Ergebnis führen.

Das entsptricht einer Ausführung des A*-Algorithmus auf dem Pancake-Graphen. Im Algorithmus MinOperationsA* ist der Pseudocode dafür dargestellt. Die Knoten werden in Q als 3-Tupel mit Index, Länge und unterer Schranke gespeichert. *pre* speichert den Vorgänger jeder Permutation, wie in MinOperationsBFS. *ubound* ist die Länge des aktuell kürzesten Pfads zu einer identischen Permutation. In der while-Schleife wird die das Element von Q mit höchster Priorität (d.h. kleinstem dritten Element der 3-Tupel) bearbeitet. Zunächst wird überprüft, ob dessen untere Schranke schon zu groß ist, und ob es sich um eine identische Permutation handelt. Bei letzterem Ereignis wird *ubound* aktualisiert und zur nächsten Iteration der while-Schleife gesprungen. Andernfalls wird für die Permutation s jede mögliche γ_i-Operation ($0 \leq i \leq length - 1$) ausgeführt, und überprüft, ob die Unterschranke der entstehenden Permutation unter der Oberschranke liegt. Wenn ja, wird $\gamma_i s$ in Q eingefügt und sein Vorgänger auf $\mu(s)$ gesetzt.

Algorithmus 4 : MinOperationsA*(p)

```
Q ← Priority Queue containing (μ(p), n, LowerBound(p))
pre ← Array of Hashmaps of size n
ubound ← n
while Q ≠ ∅ do
    (index, length, lbound) ← Dequeue(Q)
    if lbound ≥ ubound then
        break
    if index = 0 then
        if n − length < ubound then
            ubound ← n − length
        continue
    s ← IthPermutation(length, index)
    for i ← 0 to length − 1 do
        if μ(γ_i s) ∉ pre[length − 2] ∧ n − length + LowerBound(γ_i s) + 1 < ubound
         then
            pre[length − 2][μ(γ_i s)] ← index
            Enqueue(Q, (μ(γ_i s), length − 1, n − length + LowerBound(γ_i s) + 1))
return ReconstructOperations(pre, n − ubound, 0)
```

Branch and Bound. Zum Vergleich mit dem A*-Verfahren wird noch ein Branch and Bound-Algorithmus vorgestellt. Er nutzt ebensfalls die untere Schranke für $A(p)$. Es wird keine Prioritätswarteschlange über alle Blätter der Suche verwaltet, sondern lediglich über die direkten Nachfolger eines Knotens. Dieses Verfahren arbeitet rekursiv und eher wie Tiefensuche anstatt Breitensuche im Gegensatz zu MinOperationsA*. Der Algorithmus ist in MinOperationsBnB zusammengefasst. Zunächst werden die Abbruchbedingungen der Rekursion überprüft: Das Erreichen einer identischen Permutation ($\mu(p) = 0$) und einer bereits besuchten Permutation. Die Rückgabe der Funktion besteht aus der kürzesten Folge gefundener Operationen und einem Wahrheitswert, ob eine kürzere Folge als *ubound* gefunden wurde. Da die Funktion rekursiv ist, ist kein explizites Verwalten des Vorgängers nötig. Es wird lediglich ein Array an Hashsets *vis* benötigt, das für jede Permutationslänge die Indizes der bereits besuchten Permutationen enthält. Die Prioritätswarteschlange Q enthält ein Paar für jeden Nachfolger, bestehend aus dessen unterer Schranke und dem i der γ_i-Operation, die zu ihm führt. *result* enthält die aktuell kürzeste Operationenfolge, *better* gibt an, ob eine kürzere Folge als *ubound* gefunden wurde. Die Nachfolger sind in Q aufsteigend nach unterer Schranke geordnet und werden in Reihenfolge abgearbeitet. *ubound* wird beim rekursiven Aufruf um 1 verringert, da eine γ_i-Operation durchgeführt werden muss, um zu der Permutation zu gelangen, die dem rekursiven Aufruf als erstes Argument gegeben wird. Wird eine kürzere Folge gefunden, wird *result* auf diese gesetzt und die aktuell durchgeführte Operation vorne angehängt. Vor der Rückgabe wird $\mu(p)$ zu den besuchten Indizes der

Permutationslänge n hinzugefügt.

Algorithmus 5 : MinOperationsBnB(p, *vis, ubound*)

```
if μ(p) = 0 then
    return (empty array, true)
if μ(p) ∈ vis[n − 1] then
    return (empty array, false)
Q ← Priority Queue containing {(LowerBound(γ_i p), i) : 0 ≤ i ≤ n − 1}
result ← empty array
better ← false
while Q ≠ ∅ do
    (lbound, i) ← Dequeue(Q)
    if lbound ≥ ubound then
        break
    (operations, found) ← MinOperationsBnB(γ_i p, vis, ubound − 1)
    if found ∧ |operations| + 1 < ubound then
        ubound ← |operations| + 1
        result ← operations with i appended at front
        better ← true
vis[n − 1] ← vis[n − 1] ∪ μ(p)
return (result, better)
```

1.4 Berechnung der PWUE-Zahl

Für diese Teilaufgabe wird Dynamische Programmierung verwendet. Man nehme an, ein Array y enthält bei Index i $A(p)$, für jedes $0 \leq i \leq (k-1)! - 1$, wobei $|p| = k - 1$ und $\mu(p) = i$. Den Wert von $A(q)$ einer Permutation q der Länge k zu berechnen, ist mithilfe von y einfach.

$$A(q) = \min_{0 \leq i \leq k-1} y[\mu(\gamma_i q)] + 1$$

Die einzige Ausnahme ist $q = I^k$, hier gilt natürlich $A(q) = 0$. Hat man alle $A(q)$ für Permutationen der Länge k berechnet, schreibt man diese wieder in y und kann mit $k + 1$ fortfahren. Dies wird solange fortgeführt, bis man n erreicht hat. Der Ansatz ist im Algorithmus Pwue zusammengefasst. Zu Beginn enthält y ausschließlich eine 0 bei Index 0, da für I^1 0 Operationen benötigt werden. Bevor y mit den Werten aller Permutationen von Länge k überschrieben werden kann, müssen alle davon berechnet sein, daher werden sie in einem weiteren Array z zwischengespeichert.

Hier kann die Symmetrie des Pancake-Graphen ausgenutzt werden: Nach obiger Formel muss für eine Permutation k Mal $\gamma_i p$ und anschließend $\mu(\gamma_i p)$ berechnet werden. Da $\mu(\gamma_i p^*) = (k-1)! - \mu(\gamma_i p) - 1$, können die Berechnungen von μ auf die Hälfte reduziert werden. Damit wird die gesamte Laufzeit auf ca. die Hälfte reduziert, da die Berechnungen von μ mit $\Theta(k \log k)$ der asymptotische Flaschenhals sind. Für jede Permutationslänge k wird also nur bis zum Permutationsindex $k!/2 - 1$ iteriert und die Permutationen mit Index i und $k! - i - 1$ gleichzeitig nach obigem Verfahren bearbeitet. Nachdem alle Permutationen der Länge k berabeitet wurden, werden y und z getauscht. Für Permutationen der Länge n gilt außerdem, dass $A(p)$ nicht mehr in z gespeichert werden muss. Daher geht k in der ersten for-Schleife nur bis $n - 1$, Permutationen der Länge n werden in der zweiten äußeren for-Schleife von Pwue behandelt. Für sie wird genauso verfahren, nur wird ein laufendes Maximum $a_{\max}$ aller $A(p)$ aktualisiert, sowie der Index einer Beispielpermutation *example* mit $A(p) = a_{\max}$. Innerhalb der for-Schleife bezeichnet a_1 den aktuell kleinsten gefundenen Wert für $A(p)$ und a_2 den für $A(p^*)$. Nach der Berechnung von $A(p)$ und $A(p^*)$ werden $a_{\max}$

und *example* gegebenenfalls aktualisiert. PWUE gibt $P(n)$ und den Index einer Beispielpermutation der Länge n zurück.

Algorithmus 6 : PWUE(n)

```
y ← {0}, z ← empty array
for k ← 2 to n − 1 do
    for i ← 0 to k!/2 − 1 do
        p ← IthPermutation(k, i)
        z[i] ← ∞
        z[k! − i − 1] ← ∞
        for j ← 0 to k − 1 do
            l ← μ(γ_j p)
            z[i] ← min(z[i], y[l] + 1)
            z[k! − i − 1] ← min(z[k! − i − 1], y[(k − 1)! − l − 1] + 1)
    z[0] ← 0
    swap(y, z)
a_max ← 0
example ← 0
for i ← 0 to n!/2 do
    a_1 ← ∞, a_2 ← ∞
    p ← IthPermutation(n, i)
    for j ← 0 to n − 1 do
        l ← μ(γ_j p)
        a_1 ← min(a_1, y[l])
        a_2 ← min(a_2, y[(n − 1)! − l − 1])
    if a_1 > a_max then
        a_max ← a_1
        example ← i
    if a_2 > a_max then
        a_max ← a_2
        example ← n! − i − 1
return (a_max, example)
```

2 Laufzeitanalyse

Für die Berechnung von Fakultäten wird im Folgenden stets eine konstante Laufzeit angenommen, da diese entweder im Voraus berechnet werden können, oder so zwischengespeichert werden können, dass das asymptotische Laufzeitverhalten unverändert bleibt.

2.1 Teilaufgabe a)

Zunächst wird die Laufzeit von ReconstructOperations bestimmt. Die while-Schleife wird maximal $n - m$ Mal durchgeführt, da m in jeder Iteration um 1 erhöht wird. Die for-Schleife durchläuft jeweils $m + 2$ Iterationen und in jeder wird einmal $\mu(s)$ berechnet, wobei $|s| = m + 1$ gilt. Im schlechtesten Fall, wenn $m = 1$, kann die Laufzeit wie folgt beschränkt

werden. (c ist eine positive Konstante.)

$$\begin{aligned}\sum_{m=1}^{n}\sum_{j=0}^{m+1}\Theta(m\log m) &= \sum_{m=1}^{n}\sum_{j=0}^{m+1} cm\log m\\ &= c\sum_{m=1}^{n}(m+2)m\log m\\ &\leq c\sum_{m=1}^{n} n^2\log n + 2n\log n\\ &= cn^3\log n + 2cn^2\log n\end{aligned}$$

Eine obere Schranke für die Laufzeit von ReconstructOperations ist folglich $O(n^3 \log n)$.

Um die Laufzeit der drei Algorithmen für Teilaufgabe a) begrenzen zu können, wird eine obere Schranke für die Suchtiefe benötigt. Die beste Schranke wäre natürlich $P(n)$, für diese Funktion konnte aber leider kein geschlossener Ausdruck gefunden werden. Daher wird folgende Oberschranke für $A(p)$ verwendet.

Lemma 5. $A(p) \leq \left\lceil \frac{2n}{3} \right\rceil$

Beweis. Wir betrachten folgenden Algorithmus zum Sortieren von p: Die Schritte 1 und 2 werden solange abwechselnd ausgeführt, bis p sortiert ist.

1. Sei i der Index des größten Elements von p, das nicht in einem sortierten Suffix liegt. Führe die Operation γ_{i+1} aus. Damit wird das größte, noch nicht einsortierte Element nach vorne gebracht.

2. Führe die Operation γ_{j-1} aus, wobei j der kleinste Index eines Elements im sortierten Suffix ist.

Da immer das größte Element, das nicht im sortierten Suffix liegt, gewählt wird, wird das sortierte Suffix mit jeder Ausführung dieser zwei Schritte um ein Element vergrößtert. Da gleichzeitig zwei Elemente aus p entfernt werden, besteht p spätestens nach $\lceil 2n/3 \rceil$ Schritten nur noch aus dem sortierten Suffix. □

MinOperationsBFS hält diese Schranke automatisch ein, da alle Knoten auf der aktuellen Ebene abgearbeitet werden, bevor zur nächsten vorgedrungen wird. Für MinOperationsA* und MinOperationsBnB kann festgelegt werden, dass keine Permutationen mit Länge kleiner als $n - \lceil 2n/3 \rceil$ besucht werden.

Die Gesamtlaufzeit von MinOperationsBFS wird durch die Anzahl besuchter Knoten auf jeder Ebene und die Zeit pro Knoten abgeschätzt. Von Länge n wird genau eine Permutation, p, besucht. Davon ausgehend gibt es n Möglichkeiten für eine γ_i-Operation, also werden maximal n Permutationen der Länge $n-1$ besucht. Von diesen werden durch γ_i-Operationen maximal $n(n-1)$ Permutationen der Länge $n-2$ erreicht. Diese Argumentation lässt sich rekursiv fortführen, es werden also maximal $n!/k!$ Knoten der Länge k besucht. Für kleine k ist $n!/k!$ jedoch größer als $k!$, da es aber nur $k!$ Permutationen der Länge k gibt und kein Knoten doppelt besucht wird, kann die Anzahl besuchter Knoten durch $\min(n!/k!, k!)$ beschränkt werden. Mit Lemma 4 kann k nach unten durch $\lfloor n/3 \rfloor$ begrenzt werden. Summiert man für alle k von $\lfloor n/3 \rfloor$ bis n auf, ergibt sich folgende Oberschranke für die Anzahl besuchter Knoten. Zur Auswertung der Fakultät wird die Stirlingformel verwendet. c steht

für eine positive Konstante.

$$\begin{aligned}
\sum_{k=\lfloor n/3 \rfloor}^{n} \min\left(\frac{n!}{k!}, k!\right) &\leq \sum_{k=\lfloor n/3 \rfloor}^{n} \frac{n!}{k!} \\
&\leq \frac{n!}{(n/3)!} + \sum_{k=\lfloor n/3 \rfloor+1}^{n} \frac{n!}{(n/3+1)!} \\
&= \frac{n!}{(n/3)!} + \sum_{k=\lfloor n/3 \rfloor+1}^{n} \frac{n!}{(n/3)! \cdot (n/3+1)} \\
&\leq \frac{n!}{(n/3)!} + n \cdot \frac{n!}{(n/3)! \cdot (n/3+1)} \\
&= \frac{n!}{(n/3)!} + \frac{n}{n/3+1} \cdot \frac{n!}{(n/3)!} \\
&= c \cdot \frac{n!}{(n/3)!} \\
&\approx c \cdot \frac{\sqrt{2\pi n} \cdot (n/e)^n}{\sqrt{2\pi n/3} \cdot (n/3e)^{n/3}} \\
&= c \cdot \frac{3^{n/3} \cdot (n/e)^n}{\sqrt{1/3} \cdot (n/e)^{n/3}} \\
&= c \cdot \sqrt{3} \cdot 3^{n/3} \cdot (n/e)^{2n/3}
\end{aligned}$$

Die Anzahl besuchter Knoten ist also $O(3^{n/3} \cdot (n/e)^{2n/3})$. Beim Einsetzen des kleinstmöglichen $k = \lfloor n/3 \rfloor$ in der zweiten Zeile wurde die Abrundungsfunktion der Einfachheit halber weggelassen, am asymptotischen Verhalten ändert das jedoch nichts. Von der min-Funktion wurde das erste Argument gewählt, da eine Wahl des zweiten Arguments nur zu der schwächeren Schranke von $O(n!)$ geführt hätte.

Laufzeit von MinOperationsBFS. Für einen Knoten der Länge k ergeben sich $\Theta(k^2 \log k)$ Schritte, denn in der for-Schleife wird für jeden der k Nachfolger einmal μ berechnet. γ_i kann einfach in $\Theta(k)$ Schritten berechnet werden und ist damit nicht relevant. Mit der Oberschranke für die Anzahl besuchter Knoten lässt sich die Worst Case-Laufzeit von MinOperationsBFS durch

$$O(3^{n/3} \cdot (n/e)^{2n/3} \cdot n^2 \log n + n^3 \log n)$$

beschränken. Der Term $O(n^3 \log n)$ stammt von ReconstructOperations.

Laufzeit von MinOperationsA.* Die Laufzeit bleibt grundsätzlich gleich, da im schlechtesten Fall dennoch alle Knoten besucht werden. Für einen Knoten kommt lediglich ein logarithmischer Faktor vom Einfügen in die Prioritätswarteschlange hinzu. Die Größe der Prioritätswarteschlange kann mit der Anzahl besuchter Knoten, $O(3^{n/3} \cdot (n/3)^{2n/3})$ begrenzt werden. Folglich ist die Worst-Case Zeitkomplexität von MinOperationsA*

$$\begin{aligned}
&O(3^{n/3} \cdot (n/e)^{2n/3} \cdot n^2 \log n) \cdot \log(O(3^{n/3} \cdot (n/e)^{2n/3})) \\
&\quad = O(3^{n/3} \cdot (n/e)^{2n/3} \cdot n^2 \log n) \cdot O(\log(3^{n/3} \cdot (n/e)^{2n/3})) \\
&\quad = O(3^{n/3} \cdot (n/e)^{2n/3} \cdot n^2 \log n) \cdot O(\log(3^{n/3}) + \log((n/e)^{2n/3})) \\
&\quad = O(3^{n/3} \cdot (n/e)^{2n/3} \cdot n^2 \log n) \cdot O(n/3 \cdot \log(3) + 2n/3 \cdot \log(n/e)) \\
&\quad = O(3^{n/3} \cdot (n/e)^{2n/3} \cdot n^2 \log n) \cdot O(n \log n) \\
&\quad = O(3^{n/3} \cdot (n/e)^{2n/3} \cdot n^3 \log^2 n)
\end{aligned}$$

ReconstructOperations kann hier außer Acht gelassen werden, da $n^3 \log n$ bereits als Faktor vorhanden ist.

Die Worst Case-Laufzeit von MinOperationsBnB pro besuchter Permutation der Länge k beträgt $O(k^2 \log k)$. Im Pseudocode ist das nicht direkt ersichtlich, denn pro Aufruf von MinOperationsBnB fällt ohne Beachtung der rekursiven Aufrufe nur eine Laufzeit von $O(k^2)$ an. Die Anzahl an besuchten Permutationen entspricht aber nicht der Anzahl an Aufrufen von MinOperationsBnB. Wenn eine Permutation bereits besucht wurde, geschieht dennoch ein rekursiver Aufruf, der dann nach einer Berechnung von $\mu(p)$ und einer Befragung von *vis* beendet wird. Man muss für einen besuchten Knoten also die $O(n)$ Berechnungen von $\mu(p)$ auf der folgenden Rekursionsebene mit jeweils $O(n \log n)$ Kosten einberechnen. Im Worst Case beträgt die Laufzeit von MinOperationsBnB also

$$O(3^{n/3} \cdot (n/e)^{2n/3} \cdot n^2 \log n + n^3 \log n)$$

Aussagen über die Average Case-Laufzeit würden eine Formel für das durchschnittliche $A(p)$ erfordern. Nimmt man an, dass es immer in einem konstanten Verhältnis zu n steht, das heißt $A(p)$ ist im Durchschnitt rn, für $0 < r \leq 2/3$, bleibt auch die Average Case Laufzeit exponentiell, denn

$$\sum_{k=\lfloor (1-r)n \rfloor}^{n} \min\left(\frac{n!}{k!}, k!\right) = O((1-r)^{-n(1-r)} \cdot (n/e)^{rn})$$

nach der gleichen Begründung wie oben.

2.2 Teilaufgabe b)

In Pwue werden für die Hälfte aller Permutation der Länge k $\Theta(k^2 \log k)$ Schritte durchgeführt, da für jedes $0 \leq i \leq k-1$ einmal $\mu(\gamma_i p)$ berechnet wird. Für jedes $2 \leq k \leq n$ geschieht das für $k!/2$ Permutationen. Bezüglich *swap* wird eine konstante Laufzeit angenommen, da einfach die Speicheradressen der Arrays getauscht werden können. Summiert man für alle k von 2 bis n auf, ergibt sich folgende Gesamtlaufzeit.

$$\begin{aligned}
\sum_{k=2}^{n} \frac{k!}{2} \Theta(k^2 \log k) &= \frac{1}{2} \sum_{k=2}^{n-1} k! \cdot \Theta(k^2 \log k) + \frac{n!}{2} \Theta(n^2 \log n) \\
&\leq \frac{1}{2} \sum_{k=2}^{n-1} (n-1)! \cdot \Theta(n^2 \log n) + \frac{n!}{2} \cdot \Theta(n^2 \log n) \\
&\leq \frac{n}{2} \cdot (n-1)! \cdot \Theta(n^2 \log n) + \frac{n!}{2} \cdot \Theta(n^2 \log n) \\
&= n! \cdot \Theta(n^2 \log n)
\end{aligned}$$

Die Laufzeit ist also durch $O(n! \cdot n^2 \log n)$ nach oben beschränkt. Das entspricht auch der asymptotischen Unterschranke, da

$$\sum_{k=2}^{n} \frac{k!}{2} \Theta(k^2 \log k) \geq \frac{n!}{2} \cdot \Theta(n^2 \log n)$$

ist. Folglich beträgt die Laufzeit von Pwue $\Theta(n! \cdot n^2 \log n)$.

3 Implementierung

Die zwei Programme für Teilaufgabe a) und b) werden in C++ implementiert. In den Dateien `aufgabe3_a.cpp` und `aufgabe3_b.cpp` steht der Quellcode speziell für die Teilaufgaben, in `util.cpp` Funktionen, die von beiden benötigt werden. Zum Kompilieren auf Linux kann `make a` bzw. `make b` im Ordner `aufgabe3` ausgeführt werden. Mit nur `make` werden beide Teilaufgaben kompiliert. Zur Kompilierung wird mindestens C++20 benötigt.

Dem Programm `aufgabe3_a` kann in der Kommandozeile als Argument `--bfs` oder `--bnb` mitgegeben werden, um MinOperationsBFS bzw. MinOperationsBnB als Algorithmus zu verwenden. Standardmäßig wird MinOperationsA* verwendet. Wenn z. B. MinOperationsBnB verwendet werden soll und `pancake0.txt` die Eingabe ist, sieht der Befehl (im Ordner `aufgabe3`) wie folgt aus:

```
./aufgabe3_a < beispiele/pancake0.txt --bnb
```

Ausgegeben wird eine Folge an Indizes, nach denen gewendet werden kann, um auf kürzestmöglichem Weg zu einem sortierten Stapel zu gelangen. Zusätzlich werden alle Stapel, die als Zwischenschritte im Sortierprozess entstehen, ausgegeben.

Das Programm für Teilaufgabe b) hat nur n als Eingabe. Ausgegeben wird $P(n)$ und eine Beispielpermutation p mit $A(p) = P(n)$. Um $P(10)$ zu berechnen, lautet der Befehl (im Ordner `aufgabe3`) also wie folgt:

```
./aufgabe3_b
```

Anschließend muss 10 und Enter eingegeben werden.

Im Quellcode wird eine Permutation als `vector<unsigned>` repräsentiert. Die Elemente beginnen bei 0, d. h. jedes Element wird beim Einlesen um 1 verringert. Die Namensgebung ist eng an die Namensgebung im Pseudocode geknüpft.

3.1 util.cpp

3.1.1 factorial

```
uint64_t factorial(uint64_t n)
```

Berechnet rekursiv $n!$.

3.1.2 ind

```
uint64_t ind(vector<unsigned> const &p)
```

Gibt $\mu(p)$ zurück. Wie bereits beschrieben, ist das mit einem Segmentbaum in $\Theta(n \log n)$ Zeit möglich. Für die Variablen *lgn* und m gilt $lgn = \lceil \log_2 n \rceil$ und $m = 2^{\lceil \log_2 n \rceil}$. Der Segmentbaum wird in dem Array *tree* in Heap-Anordnung gespeichert, d. h. die Wurzel liegt bei Index 1, der linke Nachfolger von i bei $2 \cdot i$ und der rechte Nachfolger von i bei $2 \cdot i + 1$. Da der Segmentbaum eine 1 bei Index i stehen hat, wenn Element i bereits auftrat, wird er zu Beginn mit 0 an allen Knoten initialisiert. In der for-Schleife ist `z` der Index des aktuellen Knotens im Segmentbaum, der zu Beginn das Blatt zugehörig zu p_j ist. Für k, die Rückgabe von `ind`, gilt bezüglich der äußeren for-Schleife folgende Invariante: Wenn $j = j'$, gilt nach am Ende der Iteration

$$k = \mu(p)_0 \frac{(n-1)!}{(n-1-j')!} + \mu(p)_1 \frac{(n-2)!}{(n-1-j')!} + \cdots + \mu(p)_{j'}$$

Bei $j = n-1$ gilt offensichtlich $k = \mu(p)$. Die Invariante für die Koeffizienten jedes $\mu(p)_i$ wird durch Multiplikation mit `p.size() - j` erhalten, so wird ein Faktor $n - j$ aus dem Nenner genommen. Da $\mu(p)_j$ gleich p_j minus der Anzahl links gelegener, kleinerer Elemente ist, muss nach der Addition von p_j nur noch die Zahl links gelgener, kleinerer Elemente abgezogen werden. Das geschieht mithilfe des Segmentbaums. Er wird vom Blatt von p_j beginnend nach oben gelaufen und die Summe des linken Geschwisterknoten, falls man aktuell bei einem rechten Nachfolger steht, abgezogen. Der Wert jedes Knotens auf dem Weg wird um 1 erhöht, da p_j nun selbst vorgekommen ist. Nachdem das für alle $0 \leq j \leq n-1$ geschehen ist, wird k zurückgegeben.

3.1.3 gamma

```
vector<unsigned> gamma(vector<unsigned> const &p, unsigned i)
```

Berechnet $\gamma_i p$. Es gilt $(\gamma_i p)_j = p_{i-j-1}$ für $j < i$ und $(\gamma_i p)_j = p_{j+1}$. Bei Elementen größer als p_i wird außerdem noch 1 abgezogen.

3.1.4 ind_gamma

```
uint64_t ind_gamma(vector<unsigned> const &p, unsigned i)
```

Berechnet $\mu(\gamma_i p)$. Da diese Vekettung häufig verwendet wird, wurde eine separate Funktion geschrieben, die das Erstellen eines neuen Vektors für $\gamma_i p$ vermeidet. Die Funktionsweise ist identisch zu der von `ind`, nur werden Elemente vor i in entsprechend umgekehrter Reihenfolge bearbeitet und gegebenenfalls 1 von ihrem Wert abgezogen. x bezeichnet in beiden for-Schleifen den Wert von $(\gamma_i p)_j$, ansonsten ist die Namensgebung gleich.

3.1.5 calc_factorial_digits

```
void calc_factorial_digits(uint64_t i, vector<unsigned> &digits)
```

Berechnet die Ziffern von i im fakultätsbasierten Zahlensystem und speichert sie in `digits`, wobei die 0!-Ziffer in an letzer Stelle von s steht. Die Berechnung geschieht analog zur Berechnung der Ziffern einer Zahl im Binärsystem: Zuerst wird durch 1 geteilt, dann durch 2, 3, usw. wobei die jeweiligen Reste den Ziffern von rechts nach links gelesen entsprechen.

3.1.6 ith_permutation

```
vector<unsigned> ith_permutation(unsigned n, uint64_t i)
```

Gibt eine Permutation p der Länge n zurück, sodass $\mu(p) = i$. Zunächst wird p als Vektor angelegt und darin die Ziffern von i im fakultätsbasierten Zahlensystem gespeichert. Wie in der Lösungsidee beschrieben enthält der Segmentbaum *tree* hier eine 1 bei Index j, wenn der Wert j noch nicht aufgetreten ist, weshalb der Baum mit Einsen initialisiert wird. In der folgenden for-Schlefe werden die Ziffern von links nach rechts bearbeitet. z ist der Index des aktuellen Knotens im Segmentbaum. Bei Bearbeitung j-ten Elements wird von der Wurzel ($z = 1$) gestartet, und p_j enthält immer die nötigen links gelegenen Einsen. Im Segmentbaum wird nach rechts gegangen, wenn im linken Nachfolger weniger Einsen als nötig vorhanden sind, oder genauso viele, andernfalls nach links. Der Wert des j-ten Elements der Permutation entspricht dann $z - m$, also dem Index des Blatts, bei dem man gelandet ist. Hier werden auf dem Weg nach unten die Werte aller Knoten um 1 reduziert, da p_j danach vorhanden ist.

3.1.7 reverse_and_eat

```
vector<unsigned> reverse_and_eat(vector<unsigned> const &p, unsigned i)
```

Gibt die Permutation zurück, wenn eine Wende-und-Ess-Operation bei Index i auf p angewandt wird, wie sie im Aufgabenblatt beschrieben ist. Die Funktion wird nur in Teilaufgabe a) zur Ausgabe der entstehenden Stapel während des Sortierens benötigt.

3.2 aufgabe3_a.cpp

3.2.1 min_operations_bfs

```
vector<unsigned> min_operations_bfs(vector<unsigned> const &p)
```

Die Funktion implementiert den in der Lösungsidee beschriebenen Algorithms MinOperationsBFS. Zum Abspeichern des Vorgängers jeder besuchten Permutation wird in einem

`vector` für jede Permutationslänge eine `unordered_map` angelegt. Die Elemente der Warteschlange sind keine 2-Tupels, wie im Pseudocode, sondern Objekte von `struct Node`, die Index, Länge und untere Schranke der Permutation enthalten. Für dieses Verfahren wird die untere Schranke nicht verwendet, um den Code kurz und einheitlich zu halten, werden dennoch `Node`-Objekte in der Warteschlange gespeichert.

3.2.2 is_increasing

```
bool is_increasing(bool incr, unsigned a, unsigned b, unsigned &x)
```

Gibt zurück, ob $a < b$ und erhöht x um 1, falls sich das Steigungsverhalten gegenüber vorher (gegeben durch `incr`) verändert hat. Diese Funktion ist dazu da, Codewiederholung in `get_lbound` und `get_lbound_gamma` zu vermeiden.

3.2.3 get_lbound

```
unsigned get_lbound(vector<unsigned> const &p)
```

Gibt die untere Schranke von $\lfloor (x+1)/3 \rfloor$ für die Anzahl benötigter γ_i-Operationen für p, zurück, wobei x die Anzahl monotoner Teilstrings ist. Die Überprüfung, ob bei Index i ein neuer monotoner Teilstring beginnt, wurde in die Funktion `is_increasing` ausgelagert. `x` wird von dieser automatisch erhöht.

3.2.4 get_lbound_gamma

```
unsigned get_lbound_gamma(vector<unsigned> const &p, unsigned i)
```

Berechnet LowerBound($\gamma_i p$). Eine separate Funktion dafür vermeidet das Anlegen eines Vektors für $\gamma_i p$. Die Funktionsweise ist aber identisch zu `get_lbound`.

3.2.5 min_operations_astar

```
vector<unsigned> min_operations_astar(vector<unsigned> const &p)
```

Mit dieser Funktion wird der Algorithmus MinOperationsA* implementiert. Für die Prioritätswarteschlange `q` wird eine `priority_queue` aus der C++ Standardbibliothek verwendet. Sie enthält Objekte von Typ `Node`, die aufsteigend nach unterer Schranke und Länge sortiert werden. Der Vergleich zweier `Node`-Objekte geschieht durch den in `Node` angegebenen Vergleichsoperator `bool operator<(Node const &x)`. Bei gleicher unterer Schranke eine kürzere Permutation zu bevorzugen ist sinnvoll, da man so tendenziell schneller die Suchtiefe vergrößert und durch das Finden einer identischen Permutation eine obere Schranke festlegen kann. Im Quellcode ist im Gegensatz zum Pseudocode eine weitere Bedingung, um eine Permutation zu `q` hinzuzufügen, dass `length - 1 >= n - (2 * n + 2) / 3`. Damit wird die in der Laufzeitanalyse hergeleitete Unterschranke von $A(p)$ von $n - \lceil 2n/3 \rceil$ beachtet.

3.2.6 min_operations_bnb

```
vector<unsigned> min_operations_bnb(vector<unsigned> const &p)
```

Das Branch and Bound-Verfahren benötigt in der Implementierung zwei Funktionen. In dieser Funktion wird das Array `vis` von `unordered_set`, das `min_operations_bnb_r` als Parameter mitgegeben wird, angelegt. Des Weiteren wird der erhaltene Vektor an γ_i-Operationen vor der Rückgabe umgekehrt, da die Operationen in der Implementierung im Gegensatz zum Pseudocode hinten an `res` angefügt werden. In Vektoren Elemente vorne anzufügen benötigt $\Theta(n)$ Zeit, während es hinten in amortisiert konstanter Zeit möglich ist.

3.2.7 min_operations_bnb_r

```
pair<vector<unsigned>, bool> min_operations_bnb_r(
    vector<unsigned> const &p, vector<unordered_set<uint64_t>> &vis,
    unsigned ubound = UINT_MAX)
```

Diese Funktion implementiert MinOperationsBnB. Die obere Schranke `ubound` muss beim ersten Aufruf nicht als Parameter mitgegeben werden, sondern wird standardmäßig auf einen großen Wert gesetzt. Wie im Pseudocode werden zunächst die Abbruchbedingungen der Rekursion überprüft. Hier kommt zu den Abbruchbedingungen aus dem Pseudocode noch `p.size() <= vis.size() - (2 * vis.size() + 2) / 3` hinzu. Ist das erfüllt, und `p` ist keine identische Permutationen, was zuvor geprüft wurde, ist die in der Laufzeitanalyse hergeleitete Unterschranke $n - \lceil 2n/3 \rceil$ von $A(p)$ verletzt. `p` kann also nicht auf dem kürzesten Pfad zu einer identischen Permutation liegen. `vis.size()` ist das n der anfänglichen Permutation. Anschließend werden alle durch eine γ_i-Operation erreichbaren Permutationen in eine Warteschlange eingefügt. Es werden `Node`-Objekte zum Speichern von Permutationen verwendet, die aufsteigend nach unterer Schranke und Länge sortiert werden. Ansonsten entspricht die Funktionsweise exakt dem Pseudocode.

3.2.8 reconstruct_operations

```
vector<unsigned> reconstruct_operations(
    vector<unordered_map<uint64_t, uint64_t>> const &pre, unsigned length,
    uint64_t index)
```

Gibt die Folge an γ_i-Operationen zurück, mit denen man zur Permutation p der Länge m mit $\mu(p) = i$ gelangt. `operations` enthält während der for-Schleife die Indizes der γ_i-Operationen in umgekehrter Reihenfolge, da sie in der for-Schleife hinten angefügt werden. Wie im Pseudocode werden alle möglichen γ_i-Operationen auf der Vorgängerpermutation ausprobiert, um die richtige zu finden. Anschließend werden `index` und `length` aktualisiert, sodass sie nun die Vorgängerpermutation repräsentieren.

3.3 aufgabe3_b.cpp

3.3.1 pwue

```
pair<unsigned, uint64_t> pwue(unsigned n)
```

Gibt $P(n)$ sowie den Index einer Permutation der Länge n mit $A(p) = P(n)$ zurück. Die Berechnung wird parallelisiert: Wenn gerade $A(p)$ für alle Permutationen der Länge k bestimmt werden soll, wird das Intervall $[0, k!/2 - 1]$ in `num_threads` gleich große Teile geteilt und jedem Thread das zu bearbeitende Intervall an Indizes mitgegeben. `num_threads` bezeichnet eine geeignete Anzahl an Threads für den Computer, auf dem das Programm ausgeführt wird. Sie kann mithilfe von `thread::hardware_concurrency()` aus der C++ Standardbibliothek festgelegt werden. Die Funktion `pwue` übernimmt in der Implementierung nur das Zuteilen der Threads und Sammeln der Ergebnisse, die eigentlichen Berechnungen finden in den zwei Threadfunktionen `update_z` und `get_max_a` statt. Wie im Pseudocode ist die Berechnung in zwei Teile gegliedert: Für k von 2 bis $n - 1$ wird $A(p)$ für jede Permutation in `z` geschrieben, daher wird in diesem Teil `update_z` als Threadfunktion verwendet. `y` und `z` werden als Arrays von 8-Bit-Ganzzahlen umgesetzt, um Speicher zu sparen. Es wurden keine Vektoren aus der C++-Standardbibliothek verwendet, sondern dynamische Arrays mit `malloc` erstellt, da diese möglicherweise etwas schneller sind. Mit `realloc` wird die Größe von `z` in jeder Iteration der for-Schlefe angepasst. 8-Bit-Zahlen sind geeignet, denn `y` und `z` müssen nur Zahlen bis n speichern können, und die Annahme $n \leq 2^8 - 1 = 255$ ist vertretbar. Im zweiten Teil hat jeder Thread eine Rückgabe, das maximale von ihm gefundene $A(p)$ und ein Beispiel dazu. Daher werden die Threads hier mit `async` gestartet, ihre Rückgabe kann dann über die zurückgegebenen `future`-Objekte erhalten werden.

3.3.2 update_z

```
void update_z(
    unsigned k, uint8_t const *const y, uint8_t *const z, uint64_t i1,
    uint64_t i2)
```

Berechnet $A(p)$ für alle Permutationen der Länge k mit $i_1 \leq \mu(p) \leq i_2 - 1$ und schreibt das Ergebnis in `z` an Index $\mu(p)$. In `y` muss $A(p)$ für jede Permutation der Länge $k-1$ stehen. Zu Beginn wird in `u` $k!$ und in `v` $(k-1)!$ gespeichert, da diese Fakultäten anschließend häufig benötigt werden. Da jeder Thread nur auf einen für ihn vorgegebenen Bereich von `z` zugreift, muss keine Synchronisation stattfinden, was die Parallelisierung sehr effizient macht. Anstatt wie im Pseudocode in jeder Iteration der äußeren for-Schleife `ith_permutation` aufzurufen, wird `next_permutation` aus der C++ Standardbibliothek verwendet, das die lexikographisch nächste Permutation erzeugt. Dessen Laufzeit beträgt nur $O(n)$ statt $\Theta(n \log n)$ von `ith_permutation`, allerdings ändert das nichts am asymptotischen Laufzeitverhalten.

3.3.3 get_max_a

```
pair<unsigned, uint64_t> get_max_a(
    unsigned n, uint8_t const *const y, uint64_t i1, uint64_t i2)
```

Gibt das maximale $A(p)$ für Permutationen der Länge n mit $i_1 \leq \mu(p) \leq i_2 - 1$ zurück, sowie den Index einer Beispielpermutation mit maximalem $A(p)$. Entspricht der zweiten Hälfte des Pseudocodes PWUE, nur dass nicht alle Permutationen der Länge n, sondern nur die im Intervall `[i1, i2)` bearbeitet werden.

4 Beispiele

4.1 Ausgaben

Das Programm für a) gibt neben den Indizes der nötigen γ_i-Operationen (0-indexiert) auch den aktuellen Stapel nach jedem Sortierschritt aus. Damit kann man nachvollziehen, dass tatsächlich gültige Wendeoperationen ausgeführt wurden. Die Ausgaben stammen von MINOPERATIONSA*, die anderen zwei Algorithmen geben im Allgemeinen andere, kürzeste Folgen aus. Die Beispiele *pancake8* bis *pancake13* wurden selbst hinzugefügt, wobei versucht wurde, möglichst schwierige Beispiele zu erstellen. „Schwierig" bedeutet nicht unbedingt ein großes n, sondern eine hohe Anzahl nötiger Operationen, weil so die Suchtiefe groß wird. Bestimmte Arten von Permutationen weisen dabei häufig eine großes $A(p)$ auf, z. B. haben *pancake10* und *pancake11* haben die Form

$$1, n, 2, n-1, 3, n-2, 4, n-3, \ldots, \lfloor n/2 \rfloor + 1$$

Mit einem „großen $A(p)$" ist gemeint, dass durch Ausprobieren einiger zufälliger Permutationen der Länge n keine mit höherem $A(p)$ gefunden werden konnte.

4.1.1 pancake0

```
2 Operationen notwendig. Dazu hinter folgenden Indizes wenden:
4, 3

Index |  p
4     |  3   2   4   5   1
3     |  5   4   2   3
      |  2   4   5
```

4.1.2 pancake1

```
3 Operationen notwendig. Dazu hinter folgenden Indizes wenden:
2, 3, 4

Index |  p
2     |  6   3   1   7   4   2   5
3     |  3   6   7   4   2   5
4     |  7   6   3   2   5
      |  2   3   6   7
```

4.1.3 pancake2

```
4 Operationen notwendig. Dazu hinter folgenden Indizes wenden:
1, 3, 3, 4

Index |  p
1     |  8   1   7   5   3   6   4   2
3     |  8   7   5   3   6   4   2
3     |  5   7   8   6   4   2
4     |  8   7   5   4   2
      |  4   5   7   8
```

4.1.4 pancake3

```
6 Operationen notwendig. Dazu hinter folgenden Indizes wenden:
10, 4, 5, 2, 2, 4

Index |  p
10    |  5   10  1   11  4   8   2   9   7   3   6
4     |  3   7   9   2   8   4   11  1   10  5
5     |  2   9   7   3   4   11  1   10  5
2     |  4   3   7   9   2   1   10  5
2     |  3   4   9   2   1   10  5
4     |  4   3   2   1   10  5
      |  1   2   3   4   5
```

4.1.5 pancake4

```
7 Operationen notwendig. Dazu hinter folgenden Indizes wenden:
3, 10, 9, 7, 3, 0, 6

Index |  p
3     |  7   4   11  5   10  6   1   13  12  9   3   8   2
10    |  11  4   7   10  6   1   13  12  9   3   8   2
9     |  3   9   12  13  1   6   10  7   4   11  2
7     |  4   7   10  6   1   13  12  9   3   2
3     |  12  13  1   6   10  7   4   3   2
0     |  1   13  12  10  7   4   3   2
6     |  13  12  10  7   4   3   2
      |  3   4   7   10  12  13
```

4.1.6 pancake5

```
6 Operationen notwendig. Dazu hinter folgenden Indizes wenden:
0, 12, 6, 8, 4, 5

Index |  p
0     |  4   13  10  8   2   3   7   9   14  1   12  6   5   11
12    |  13  10  8   2   3   7   9   14  1   12  6   5   11
6     |  5   6   12  1   14  9   7   3   2   8   10  13
8     |  9   14  1   12  6   5   3   2   8   10  13
4     |  2   3   5   6   12  1   14  9   10  13
5     |  6   5   3   2   1   14  9   10  13
      |  1   2   3   5   6   9   10  13
```

4.1.7 pancake6

```
8 Operationen notwendig. Dazu hinter folgenden Indizes wenden:
6, 9, 8, 3, 0, 4, 8, 0

Index |  p
6     |  14  8   4   12  13  2   1   15  7   11  3   9   5   10  6
9     |  2   13  12  4   8   14  15  7   11  3   9   5   10  6
8     |  11  7   15  14  8   4   12  13  2   9   5   10  6
3     |  13  12  4   8   14  15  7   11  9   5   10  6
0     |  4   12  13  14  15  7   11  9   5   10  6
4     |  12  13  14  15  7   11  9   5   10  6
```

```
8     |  15  14  13  12  11  9   5   10  6
0     |  10  5   9   11  12  13  14  15
      |  5   9   11  12  13  14  15
```

4.1.8 pancake7

```
8 Operationen notwendig. Dazu hinter folgenden Indizes wenden:
2, 2, 5, 9, 3, 10, 9, 5

Index |  p
2     |  8   5   10  15  3   7   13  6   2   4   12  9   1   14  16  11
2     |  5   8   15  3   7   13  6   2   4   12  9   1   14  16  11
5     |  8   5   3   7   13  6   2   4   12  9   1   14  16  11
9     |  13  7   3   5   8   2   4   12  9   1   14  16  11
3     |  9   12  4   2   8   5   3   7   13  14  16  11
10    |  4   12  9   8   5   3   7   13  14  16  11
9     |  16  14  13  7   3   5   8   9   12  4
5     |  12  9   8   5   3   7   13  14  16
      |  3   5   8   9   12  13  14  16
```

4.1.9 pancake8

Eingabe: 14 9 2 12 4 11 5 14 6 3 13 8 10 7 1

```
8 Operationen notwendig. Dazu hinter folgenden Indizes wenden:
12, 4, 8, 6, 4, 7, 6, 6

Index |  p
12    |  9   2   12  4   11  5   14  6   3   13  8   10  7   1
4     |  10  8   13  3   6   14  5   11  4   12  2   9   1
8     |  3   13  8   10  14  5   11  4   12  2   9   1
6     |  4   11  5   14  10  8   13  3   2   9   1
4     |  8   10  14  5   11  4   3   2   9   1
7     |  5   14  10  8   4   3   2   9   1
6     |  2   3   4   8   10  14  5   1
6     |  14  10  8   4   3   2   1
      |  2   3   4   8   10  14
```

4.1.10 pancake9

Eingabe: 16 15 2 6 10 4 8 12 1 13 16 9 14 7 5 11 3

```
9 Operationen notwendig. Dazu hinter folgenden Indizes wenden:
2, 10, 9, 5, 6, 4, 8, 6, 7

Index |  p
2     |  15  2   6   10  4   8   12  1   13  16  9   14  7   5   11  3
10    |  2   15  10  4   8   12  1   13  16  9   14  7   5   11  3
9     |  9   16  13  1   12  8   4   10  15  2   7   5   11  3
5     |  15  10  4   8   12  1   13  16  9   7   5   11  3
6     |  12  8   4   10  15  13  16  9   7   5   11  3
4     |  13  15  10  4   8   12  9   7   5   11  3
8     |  4   10  15  13  12  9   7   5   11  3
6     |  5   7   9   12  13  15  10  4   3
7     |  15  13  12  9   7   5   4   3
      |  4   5   7   9   12  13  15
```

4.1.11 pancake10

Eingabe: 18 1 18 2 17 3 16 4 15 5 14 6 13 7 12 8 11 9 10

```
10 Operationen notwendig. Dazu hinter folgenden Indizes wenden:
3, 10, 9, 5, 1, 5, 9, 8, 6, 5

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

4.1.12 pancake11

Eingabe: 20 1 20 2 19 3 18 4 17 5 16 6 15 7 14 8 13 9 12 10 11

```
11 Operationen notwendig. Dazu hinter folgenden Indizes wenden:
1, 8, 11, 9, 1, 7, 13, 3, 7, 6, 8
```

```
Index |  p
1     |  1   20  2   19  3   18  4   17  5   16  6   15  7   14  8   13  9   12  10  11
8     |  1   2   19  3   18  4   17  5   16  6   15  7   14  8   13  9   12  10  11
11    |  5   17  4   18  3   19  2   1   6   15  7   14  8   13  9   12  10  11
9     |  7   15  6   1   2   19  3   18  4   17  5   8   13  9   12  10  11
1     |  4   18  3   19  2   1   6   15  7   5   8   13  9   12  10  11
7     |  4   3   19  2   1   6   15  7   5   8   13  9   12  10  11
13    |  15  6   1   2   19  3   4   5   8   13  9   12  10  11
3     |  10  12  9   13  8   5   4   3   19  2   1   6   15
7     |  9   12  10  8   5   4   3   19  2   1   6   15
6     |  3   4   5   8   10  12  9   2   1   6   15
8     |  12  10  8   5   4   3   2   1   6   15
      |  1   2   3   4   5   8   10  12  15
```

4.1.13 pancake12

Eingabe: 19 9 14 1 17 7 15 2 5 16 10 4 3 12 6 19 8 11 18 13

```
10 Operationen notwendig. Dazu hinter folgenden Indizes wenden:
14, 8, 16, 5, 11, 2, 11, 4, 5, 9
```

```
Index |  p
14    |  9   14  1   17  7   15  2   5   16  10  4   3   12  6   19  8   11  18  13
8     |  6   12  3   4   10  16  5   2   15  7   17  1   14  9   8   11  18  13
16    |  2   5   16  10  4   3   12  6   7   17  1   14  9   8   11  18  13
5     |  18  11  8   9   14  1   17  7   6   12  3   4   10  16  5   2
11    |  14  9   8   11  18  17  7   6   12  3   4   10  16  5   2
2     |  4   3   12  6   7   17  18  11  8   9   14  16  5   2
11    |  3   4   6   7   17  18  11  8   9   14  16  5   2
4     |  16  14  9   8   11  18  17  7   6   4   3   2
5     |  8   9   14  16  18  17  7   6   4   3   2
9     |  18  16  14  9   8   7   6   4   3   2
      |  3   4   6   7   8   9   14  16  18
```

4.1.14 pancake13

Eingabe: 20 5 11 12 16 15 17 18 2 7 10 4 8 20 3 1 13 6 19 14 9

```
9 Operationen notwendig. Dazu hinter folgenden Indizes wenden:
12, 1, 17, 9, 7, 11, 13, 6, 2
```

```
Index |  p
12    |  5   11  12  16  15  17  18  2   7   10  4   8   20  3   1   13  6   19  14  9
1     |  8   4   10  7   2   18  17  15  16  12  11  5   3   1   13  6   19  14  9
17    |  8   10  7   2   18  17  15  16  12  11  5   3   1   13  6   19  14  9
9     |  14  19  6   13  1   3   5   11  12  16  15  17  18  2   7   10  8
7     |  12  11  5   3   1   13  6   19  14  15  17  18  2   7   10  8
11    |  6   13  1   3   5   11  12  14  15  17  18  2   7   10  8
13    |  18  17  15  14  12  11  5   3   1   13  6   7   10  8
6     |  10  7   6   13  1   3   5   11  12  14  15  17  18
2     |  3   1   13  6   7   10  11  12  14  15  17  18
      |  1   3   6   7   10  11  12  14  15  17  18
```

4.2 Vergleich der drei Algorithmen

Die drei Verfahren für Aufgabe a) wurden auf alle Beispiele angewandt und jeweils die Laufzeit (in s) sowie die Anzahl besuchter Permutationen gemessen. Diese Zahl wurde durch Zählen der Einträge in *pre* bzw. *vis* erhalten. Der verwendete PC hat einen AMD Ryzen 7 3700U (Mobile) Prozessor und ca. 5,7 GB verfügbaren Arbeitsspeicher.

	MinOperationsBFS		MinOperationsA*		MinOperationsBnB	
	Laufzeit	# Knoten	Laufzeit	# Knoten	Laufzeit	# Knoten
pancake0	0,003	11	0,003	11	0,003	8
pancake1	0,004	70	0,004	62	0,003	60
pancake2	0,004	148	0,003	118	0,004	147
pancake3	0,011	3063	0,010	2941	0,012	2941
pancake4	0,063	26544	0,050	24385	0,054	23880
pancake5	0,093	86718	0,059	42919	0,048	28865
pancake6	0,507	293320	0,423	258198	0,718	263616
pancake7	2,544	1056514	0,993	692598	1,973	693175
pancake8	0,121	83194	0,118	81564	0,209	80881
pancake9	2,490	1052794	2,782	1025096	4,524	1020675
pancake10	34,60	9071860	25,57	8088249	45,89	8597389
pancake11	–	–	360,2	96756054	664,2	105275870
pancake12	262,6	50747370	115,5	37119639	197,9	38180960
pancake13	–	–	23,80	15342000	72,98	21004993

Tabelle 1: Laufzeit (in s) und Anzahl besuchter Knoten der drei Algorithmen für Teilaufgabe a)

Für die kleinen Beispiele ist die Laufzeit wenig aussagekräftig, hier sieht man aber bereits, dass die zwei verbesserten Verfahren die Anzahl besuchter Permutationen verringern. Diese ist in allein Beispielen kleiner gleich der von MinOperationsBFS. Für größere Beispiele wirkt sich das auch positiv auf die Laufzeit aus. Zwischen MinOperationsA* und MinOperationsBnB lässt sich kein klarer Gewinner ausmachen. MinOperationsA* ist meistens schneller, obwohl manchmal mehr Knoten besucht wurden, was möglicherweise am zuätzlichen Aufwand der Rekursion von MinOperationsBnB liegt. *pancake11* und *pancake13* konnte von MinOperationsBFS aufgrund des zu hohen Speicherverbrauchs nicht gelöst werden. Der Speicherverbrauch der anderen zwei Verfahren lag bei *pancake11* bei ca. 5 GB und für *pancake13* bei ca. 1 GB.

4.3 Werte von $P(n)$

Mit dem Programm für Teilaufgabe b) konnte die PWUE-Zahl bis $n = 13$ berechnet werden. Die Ergebnisse sind in folgender Tabelle dargestellt.

n	$P(n)$	Beispiel mit $A(p) = P(n)$
8	5	6 3 8 5 2 7 4 1
9	5	9 7 3 8 5 2 6 4 1
10	6	10 3 5 9 4 6 8 2 7 1
11	6	11 10 4 6 9 3 8 2 7 5 1
12	7	12 9 4 11 5 8 3 6 10 2 7 1
13	7	13 12 9 4 11 8 3 7 2 6 10 5 1
14	8	9 2 12 4 11 5 14 6 3 13 8 10 7 1

Tabelle 2: PWUE-Zahlen bis $n = 14$ mit Beispielen

Dass $P(14) = 8$ ist, wurde nicht wie die anderen PWUE-Zahlen berechnet, kann aber wie folgt begründet werden. Es gilt $P(n) \leq P(n-1)+1$ für $n \geq 2$, da jede Permutation der Länge n durch eine Operation in eine Permutation der Länge $n-1$ umgewandelt werden kann. Da von dem in der Tabelle genannten Beispiel (entspricht *pancake8*) aber mithilfe des Programms für a) überprüft werden kann, dass $A(p) = 8$, muss $P(14) = 8$ sein.

Das tatsächliche Laufzeitverhalten des Programms. Im Verhältnis zur sehr schlechten asymptotischen Laufzeit von $\Theta(n! \cdot n^2 \log n)$ lief das Programm von Teilaufgabe b) relativ schnell. Zur Berechnung von $P(13)$ sollten nach der theoretischen Abschätzung in der Größenordnung von $13! \cdot 13^2 \cdot \log_2 13 \approx 3,89 \cdot 10^{12}$ Schritte nötig sein. Trotz dieser sehr großen Zahl konnte $P(13)$ mit 8 Threads in nur 22 min berechnet werden. Für $P(12)$ wurden 73 s benötigt, für $P(11)$ ca. 5,5 s. Auch der Speicherverbrauch war trotz des Speicherns von $A(p)$ für $12! = 479001600$ Permutationen wegen der Verwendung von 8-Bit-Zahlen niedrig, er lag bei ca. 500 MB.

5 Quellcode

5.1 util.cpp

```
#include <bits/stdc++.h>
#include "util.hpp"
using namespace std;

uint64_t factorial(uint64_t n) { return !n ? 1 : n * factorial(n - 1); }

// Berechnet den Index von p in einer lexikographisch sortierten Liste aller
// |p|! Permutationen der Laenge |p|.
uint64_t ind(vector<unsigned> const &p)
{
    uint64_t k = 0; // Der Index von p.
    size_t const lgn = countl_zero<size_t>(0) - countl_zero(p.size()),
                 m = 1U << lgn;
    unsigned tree[2 * m]; // Summensegmentbaum
    memset(tree, 0, sizeof tree);

    for (size_t j = 0; j < p.size(); j++)
    {
        size_t z = m + p[j]; // Aktueller Knoten.
        k *= (p.size() - j);
        k += p[j];
        // Angefangen beim p[j]-ten Blatt wird der Segmentbaum nach oben
        // durchlaufen.
        for (size_t l = 0; l < lgn; l++)
        {
            if (z & 1)            // Wenn bei einem rechten Nachfolger, ziehe die
                k -= tree[z - 1]; // Zahl links gelegener, kleinerer Elemente ab.
            tree[z]++;
            z >>= 1; // Gehe zum Elternknoten.
        }
        tree[z]++;
    }

    return k;
}

vector<unsigned> gamma(vector<unsigned> const &p, unsigned i)
{
    vector<unsigned> s(p.size() - 1);
```

```
    for (size_t j = 0; j < i; j++) // Verringere Elemente > p[i] um 1.
        s[j] = p[i - j - 1] - (p[i - j - 1] > p[i]);
    for (size_t j = i; j < p.size() - 1; j++)
        s[j] = p[j + 1] - (p[j + 1] > p[i]);

    return s;
}

// Berechnet mu(gamma_i p).
uint64_t ind_gamma(vector<unsigned> const &p, unsigned i)
{
    uint64_t k = 0;
    size_t const lgn = countl_zero<size_t>(0) - countl_zero(p.size() - 1),
                 m = 1U << lgn;
    unsigned tree[2 * m]; // Summensegmentbaum
    memset(tree, 0, sizeof tree);

    // Die Elemente vor i werden in umgekehrter Reihenfolge bearbeitet. Das j-te
    // Element gamma_i p ist das (i - j - 1)-te Element von p, für j < i.
    // Daneben wird jedes Element > p[i] von gamma_i um 1 verringert.
    for (size_t j = 0; j < i; j++)
    {
        unsigned const x = p[i - j - 1] - (p[i - j - 1] > p[i]);
        size_t z = m + x;
        k *= (p.size() - 1 - j);
        k += x;
        for (size_t l = 0; l < lgn; l++)
        {
            if (z & 1)
                k -= tree[z - 1];
            tree[z]++;
            z >>= 1;
        }
        tree[z]++;
    }

    for (size_t j = i; j < p.size() - 1; j++) // Elemente nach i.
    {
        unsigned const x = p[j + 1] - (p[j + 1] > p[i]);
        size_t z = m + x;
        k *= (p.size() - 1 - j);
        k += x;
        for (size_t l = 0; l < lgn; l++)
        {
            if (z & 1)
                k -= tree[z - 1];
            tree[z]++;
            z >>= 1;
        }
        tree[z]++;
    }

    return k;
```

```
}

// Schreibt die Ziffern von i im fakultätsbasierten Zahlensystem in digits.
void calc_factorial_digits(uint64_t i, vector<unsigned> &digits)
{
    for (size_t j = 1; j <= digits.size(); j++)
    {
        digits[digits.size() - j] = i % j;
        i /= j;
    }
}

vector<unsigned> ith_permutation(unsigned n, uint64_t i)
{
    vector<unsigned> p(n);        // Verwende p zunächst als Speicher für die
    calc_factorial_digits(i, p); // Ziffern im fankultätsbasierten Zahlensystem.

    size_t const lgn = countl_zero<size_t>(0) - countl_zero(p.size()),
                 m = 1 << lgn;
    unsigned tree[2 * m];
    for (size_t l = 0; l <= lgn; l++) // Initialisiere den Baum mit Einsen.
        for (size_t j = 0; j < (1U << l); j++)
            tree[(1 << l) + j] = 1 << (lgn - l);

    for (size_t j = 0; j < n; j++)
    {
        size_t z = 1; // Index des aktuellen Knotens im Segmentbaum
        for (size_t l = 0; l < lgn; l++)
        {
            tree[z]--; // p[j] ist nun vorhanden -> setzte seinen Wert auf 0.
            z <<= 1;
            // Wenn nach rechts gegangen wird, muss die Anahl benötigter,
            // kleinerer Elemente um die Zahl kleinerer Elemente im linken
            // Teilbaum verringert werden.
            if (p[j] >= tree[z])
                p[j] -= tree[z++];
        }
        tree[z] = 0;
        p[j] = z - m;
    }

    return p;
}

// Die eigentliche Wende-und-Ess-Operation.
vector<unsigned> reverse_and_eat(vector<unsigned> const &p, unsigned i)
{
    vector<unsigned> s(p.size() - 1);

    copy(p.begin(), p.begin() + i, s.begin());
    reverse(s.begin(), s.begin() + i);
    copy(p.begin() + i + 1, p.end(), s.begin() + i);
```

```
    return s;
}
```

5.2 aufgabe3_a.cpp

```
#include <bits/stdc++.h>
#include "aufgabe3_a.hpp"
#include "util.hpp"
using namespace std;

struct Node // Ein Knoten im Suchbaum. Enthält Index, Länge und untere Schranke.
{           // Bei Vergleich werden untere Schranke und Länge verglichen.
    uint64_t index;
    unsigned length, lbound;

    Node(uint64_t index_, unsigned length_, unsigned lbound_)
    {
        index = index_, length = length_, lbound = lbound_;
    }

    // Umgekehrter Vergleich wegen absteigender Sortierung in der priority_queue
    bool operator<(Node const &x) const
    {
        return lbound == x.lbound ? length > x.length : lbound > x.lbound;
    }
};

// Findet die kürzeste Folge an gamma-Operationen, um p in eine identische
// Permutation umzuformen durch Austesten aller möglichen Operationen.
vector<unsigned> min_operations_bfs(vector<unsigned> const &p)
{
    vector<unordered_map<uint64_t, uint64_t>> pre(p.size());

    queue<Node> q;
    q.emplace(ind(p), p.size(), 0); // Untere Schranke wird nicht verwendet.

    while (!q.empty())
    {
        auto const [index, length, _] = q.front();
        q.pop();

        if (!index) // Identische Permutation gefunden
            return reconstruct_operations(pre, length, index);

        vector<unsigned> const s = ith_permutation(length, index);

        for (unsigned i = 0; i < length; i++) // Wende alle möglichen gamma_i-
        {                                     // Operationen (0 <= i < length) an.
            Node const y(ind_gamma(s, i), length - 1, 0);
            // Prüfe, ob die Permutation gamma_i s schon besucht wurde.
            if (pre[y.length - 1].find(y.index) == pre[y.length - 1].end())
            {
                pre[y.length - 1][y.index] = index; // Vorgänger im Suchbaum.
                q.push(y);
```

```
            }
        }
    }

    exit(EXIT_FAILURE); // Sollte nie erreicht werden.
}

// Gibt zurück, ob a < b und erhäht x um 1, wenn sich das Steigungsverhalten
// ändert.
bool is_increasing(bool incr, unsigned a, unsigned b, unsigned &x)
{
    if (incr ^ (a < b))
    {
        x++;
        return !incr;
    }
    return incr;
}

// Bestimmt eine untere Schrankte für A(p).
unsigned get_lbound(vector<unsigned> const &p)
{
    if (p.size() == 1)
        return 0;

    unsigned x = 1; // Anzahl monotoner Teilstrings
    bool incr = p[0] < p[1];

    for (unsigned i = 2; i < p.size(); i++)
        incr = is_increasing(incr, p[i - 1], p[i], x);

    return (x + 1) / 3;
}

// Bestimmt eine untere Schranke für A(gamma_i p).
unsigned get_lbound_gamma(vector<unsigned> const &p, unsigned i)
{
    assert(i < p.size());

    if (p.size() <= 2)
        return 0;

    unsigned x = 1;
    bool incr =
        i >= 2 ? (p[i - 2] > p[i - 1])
               : (i == 1 ? (p[i + 1] > p[i - 1]) : (p[i + 2] > p[i + 1]));

    for (unsigned j = 2; j < i; j++) // umgekehrte Elemente vor i
        incr = is_increasing(incr, p[i - j], p[i - j - 1], x);

    if (i && i + 1 < p.size()) // neu benachbarte Elemente (p[0], p[i + 1])
        incr = is_increasing(incr, p[0], p[i + 1], x);
```

```
    for (unsigned j = i + 2; j < p.size(); j++) // Elemente nach i
        incr = is_increasing(incr, p[j - 1], p[j], x);

    return (x + 1) / 3;
}

// Gibt die kürzestmögliche Folge an gamma-Operationen zurück. Wie im A*-
// Algorithmus werden die Blätter der Suche in einer Prioritätswarteschlange
// gespeichert und aufsteigend nach unterer Schranke abgearbeitet.
vector<unsigned> min_operations_astar(vector<unsigned> const &p)
{
    unsigned const n = p.size();
    priority_queue<Node> q;
    q.emplace(ind(p), n, get_lbound(p));

    // Speichert für jede Permutationslänge die Indizes besuchter
    // Permutationen und deren Vorgänger.
    vector<unordered_map<uint64_t, uint64_t>> pre(n);

    unsigned ubound = n; // aktuelle Oberschranke

    while (!q.empty() && q.top().lbound < ubound)
    {
        auto const [index, length, lbound] = q.top();
        q.pop();

        if (!index)
        {
            // Eine identische Permutation wurde gefunden.
            if (n - length < ubound)
                ubound = n - length;
            continue;
        }

        vector<unsigned> const s = ith_permutation(length, index);

        // Füge jede durch eine gamma-Operation erreichbare Permutation zur
        // Warteschlange hinzu, die das Ergebnis noch verbessern kann.
        for (unsigned i = 0; i < length; i++)
        {
            Node const y(ind_gamma(s, i), length - 1,
                         n - length + get_lbound_gamma(s, i) + 1);

            if (pre[length - 2].find(y.index) == pre[length - 2].end() &&
                y.lbound < ubound && length - 1 >= n - (2 * n + 2) / 3)
            {
                pre[length - 2][y.index] = index;
                q.push(y);
            }
        }
    }

    return reconstruct_operations(pre, n - ubound, 0);
```

```
}

// Bestimmt rekursiv die kürzestmögliche Folge an gamma-Operationen zum
// Sortieren von p. Falls keine kürzere als ubound existiert, wird das zweite
// Element der Rückgabe auf 0 gesetzt.
pair<vector<unsigned>, bool> min_operations_bnb_r(
    vector<unsigned> const &p, vector<unordered_set<uint64_t>> &vis,
    unsigned ubound = UINT_MAX)
{
    if (!ind(p)) // Identische Permutation erreicht.
        return {{}, 1};

    // Prüfe, ob der Knoten schon besucht wurde oder die Oberschranke von A(p)
    // n - ceil(2n / 3) überschritten wurde.
    if (vis[p.size() - 1].find(ind(p)) != vis[p.size() - 1].end() ||
        p.size() <= vis.size() - (2 * vis.size() + 2) / 3)
        return {{}, 0};

    priority_queue<Node> q;
    for (size_t i = 0; i < p.size(); i++)
        q.emplace(i, p.size() - 1, get_lbound_gamma(p, i));

    vector<unsigned> res;
    bool found_better = 0;

    // Gehe die Nachfolgerknoten aufsteigend nach unterer Schranke durch und
    // bestimme rekursiv die kürzeste Operationenfolge.
    while (!q.empty() && q.top().lbound < ubound)
    {
        auto const [i, lbound, length] = q.top();
        q.pop();

        auto const [op, found] = // Rekursionsschritt
            min_operations_bnb_r(gamma(p, i), vis, ubound - 1);

        if (found && op.size() + 1 < ubound)
        {
            // Neue kürzeste Folge gefunden.
            ubound = op.size() + 1;
            res = op;
            res.push_back(i);
            found_better = 1;
        }
    }

    vis[p.size() - 1].insert(ind(p));
    return {res, found_better};
}

// Stellt vis für die rekursive Funktion min_operations_bnb_r bereit und bringt
// die Operationen in die richtige Reihenfolge.
vector<unsigned> min_operations_bnb(vector<unsigned> const &p)
{
```

```
    vector<unordered_set<uint64_t>> vis(p.size());
    vector<unsigned> res = min_operations_bnb_r(p, vis).first;
    reverse(res.begin(), res.end());
    return res;
}

// Läuft den durch pre gegebenen Suchbaum hoch und sammelt die gamma-
// Operationen auf dem Pfad ein. m ist die Länge der anfänglichen Permutation,
// i ihr Index.
vector<unsigned> reconstruct_operations(
    vector<unordered_map<uint64_t, uint64_t>> const &pre, unsigned length,
    uint64_t index)
{
    vector<unsigned> operations;

    while (length < pre.size())
    {
        vector<unsigned> const s =
            ith_permutation(length + 1, pre[length - 1].at(index));

        // Suche nach der gamma-Operation, die den Vorgänger (p) in den
        // Nachfolger (i-te Permutation der Länge n) umwandelt.
        for (unsigned j = 0; j < length + 1; j++)
            if (ind_gamma(s, j) == index)
            {
                operations.push_back(j);
                break;
            }

        // Gehe zur vorherigen Permutation.
        index = pre[length - 1].at(index);
        length++;
    }

    // Die Operationen wurden umgekehrt eingefügt.
    reverse(operations.begin(), operations.end());
    return operations;
}

void print_operations(vector<unsigned> p, vector<unsigned> const &op)
{
    if (!op.empty())
    {
        cout << op.size() << " Operationen notwendig. Dazu hinter folgenden"
                             " Indizes wenden:\n";
        for (auto it = op.cbegin(); it != --op.cend(); it++)
            cout << *it << ", ";
        cout << *(--op.cend()) << "\n\n"
             << "Index |  p\n";

        for (auto it = op.cbegin(); it != op.cend(); it++)
        {
            cout << left << setw(6) << *it << "|  ";
```

```
            for (unsigned const &x : p)
                cout << left << setw(4) << x + 1;
            p = reverse_and_eat(p, *it);
            cout << '\n';
        }

        cout << "      |  ";
        for (unsigned const &x : p)
            cout << left << setw(4) << x + 1;
    }
    else
        cout << "Der Stapel ist bereits sortiert.";
    cout << '\n';
}

int main(int argc, char *argv[])
{
    unsigned n;
    cin >> n;

    vector<unsigned> p(n);
    for (unsigned &x : p)
    {
        cin >> x;
        x--;
    }

    vector<unsigned> op;

    if (argc == 2 && !strcmp(argv[1], "--bnb"))
        op = min_operations_bnb(p);
    else if (argc == 2 && !strcmp(argv[1], "--bfs"))
        op = min_operations_bfs(p);
    else
        op = min_operations_astar(p);

    print_operations(p, op);
}
```

5.3 aufgabe3_b.cpp

```
#include <bits/stdc++.h>
#include "util.hpp"
using namespace std;

// Schreibt A(q) für jede Permutation der Länge k mit Index im Intervall
// [i1, i2) in z. In y muss A(p) für jede Permutation p der Laenge k - 1 stehen.
void update_z(
    unsigned k, uint8_t const *const y, uint8_t *const z, uint64_t i1,
    uint64_t i2)
{
    vector<unsigned> p = ith_permutation(k, i1);
    uint64_t const u = factorial(k), v = factorial(k - 1);
```

```
    for (uint64_t i = i1; i < i2; i++)
    {
        z[i] = k;          // Durch die Symmetrie des Pancake-Graphen kann ein
        z[u - i - 1] = k; // symmetrisches Paar gleichzeitig behandelt werden.

        for (unsigned j = 0; j < k; j++)
        {
            uint64_t const l = ind_gamma(p, j);
            z[i] = min<uint8_t>(z[i], y[l] + 1);
            z[u - i - 1] = min<uint8_t>(z[u - i - 1], y[v - l - 1] + 1);
        }

        next_permutation(p.begin(), p.end());
    }
}

// Findet ein maximales A(p) unter den Permutationen der Länge n mit Index
// zwischen i1 und i2 - 1. Zurückgegeben wird A(p) und der Index von p.
pair<unsigned, uint64_t> get_max_a(
    unsigned n, uint8_t const *const y, uint64_t i1, uint64_t i2)
{
    pair<unsigned, uint64_t> res = {0, -1};
    vector<unsigned> p = ith_permutation(n, i1);
    uint64_t const u = factorial(n), v = factorial(n - 1);

    for (uint64_t i = i1; i < i2; i++)
    {
        unsigned a1 = n, a2 = n;

        for (unsigned j = 0; j < n; j++)
        {
            uint64_t const l = ind_gamma(p, j);
            a1 = min<unsigned>(a1, y[l] + 1);          // Bearbeite p und p*
            a2 = min<unsigned>(a2, y[v - l - 1] + 1); // gleichzeitig.
        }

        res = max(res, max(make_pair(a1, i), make_pair(a2, u - i - 1)));
        next_permutation(p.begin(), p.end());
    }

    return res;
}

pair<unsigned, uint64_t> pwue(unsigned n)
{
    // Arrays zum Speichern von A(p) aller Permutationen einer bestimmten Länge.
    // In der folgenden for-Schleife gilt y[i] = A(p), wenn ind(p) = i, für jede
    // Permutation p der Länge k - 1. In z wird dann A(p) für alle Permutationen
    // von Länge k geschrieben.
    uint8_t *y = (uint8_t *)malloc(sizeof *y),
            *z = (uint8_t *)malloc(sizeof *z);
    y[0] = 0;
    unsigned const num_threads = thread::hardware_concurrency();
```

```
    for (unsigned k = 2; k < n; k++)
    {
        uint64_t const k_factorial = factorial(k);
        z = (uint8_t *)realloc(z, k_factorial * sizeof *z);
        vector<thread> threads;

        // Die k! Permutationen werden ausgeglichen unter den num_threads
        // Threads aufgeteilt.
        for (unsigned i = 0; i < num_threads; i++)
            threads.emplace_back(update_z, k, y, z,
                                 (k_factorial / 2) * i / num_threads,
                                 (k_factorial / 2) * (i + 1) / num_threads);

        for (thread &t : threads) // Warte, bis alle Threads fertig sind.
            t.join();

        z[0] = 0;
        swap(y, z);
    }

    free(z);
    vector<future<pair<unsigned, uint64_t>>> fut;

    // Für Länge n ist es nicht mehr nötig, A(p) zu speichern, daher wird
    // get_max_a als Threadfunktion verwendet. Die future-Objekte erlauben es
    // auf die Rückgabe der Threads zuzugreifen.
    uint64_t const n_factorial = factorial(n);
    for (unsigned i = 0; i < num_threads; i++)
        fut.emplace_back(async(get_max_a, n, y,
                               (n_factorial / 2) * i / num_threads,
                               (n_factorial / 2) * (i + 1) / num_threads));

    // P(n), Index einer Beispielpermutation
    pair<unsigned, uint64_t> res = {0, -1};
    for (auto &f : fut) // Finde das maximale A(p) aller Threads.
        res = max(res, f.get());
    free(y);

    return res;
}

int main()
{
    unsigned n;
    cin >> n;

    pair<unsigned, uint64_t> const res = pwue(n);

    cout << "P(" << n << ") = " << res.first << '\n';
    cout << "Beispiel mit A(p) = P(" << n << "): ";
    for (unsigned const x : ith_permutation(n, res.second))
        cout << x + 1 << ' ';
```

```
    cout << '\n';
}
```

Literatur

[1] Bonet, B. (2008). Efficient Algorithms to Rank and Unrank Permutations in Lexicographic Order.
https://bonetblai.github.io/reports/AAAI08-ws10-ranking.pdf

[2] Bulteau, L., Fertin, G., Rusu, I. (2011). Pancake Flipping is Hard.
https://arxiv.org/abs/1111.0434v1

[3] Cohen, D. S., Blum, M. (1993). On the problem of sorting burnt pancakes.
https://www.sciencedirect.com/science/article/pii/0166218X94000093

[4] Gates, W. H., Papadimitriou, C. H. (1979). Bounds for sorting by prefix reversal.
https://www.sciencedirect.com/science/article/pii/0012365X79900682

[5] Johnsonbaugh, R. (2017). Discrete Mathematics (8. Auflage). Pearson Verlag.

[6] Wikipedia (2023). Factorial number system.
https://en.wikipedia.org/wiki/Factorial_number_system

[7] Wikipedia (2023). Lexicographic order.
https://en.wikipedia.org/wiki/Lexicographic_order