Aufgabe 1: LaTeX-Dokument

Teilnahme-ID: ?????

Bearbeiter/-in dieser Aufgabe:
Vor- und Nachname

14. April 2023

Inhaltsverzeichnis

1 Lösungsidee

1.1 Notation

Mit SStapelïst im Folgenden immer der Begriff der Aufgabenstellung gemeint, nicht die Datenstruktur. Die Menge der möglichen Stapel der Höhe n wird mit P_n bezeichnet. Die Möglichen Pfannkuchen-Wende-und-Ess-Operationen für einem Stapel mit n Pfannkuchen wird mit W_n bezeichnet. Die Menge der möglichen Umkehroperationen für solch einen Stapel wird mit W_n^{-1} bezeichnet. Wenn der Stapel $S \in \mathsf{P}_n$ durch die Operation $w \in \mathsf{W}_n$ verändert wird, so wird der neue Stapel $S' = wS$ bezeichnet. Operationen assoziieren nach rechts, d.h. $w_1 w_2 S = w_1(w_2 S)$. Die Funktionen A und P werden aus der Aufgabenstellung übernommen. Wird ein Stapel im Text dargestellt, dann steht der erste Pfannkuchen für den obersten, der zweite für den zweitobersten und so weiter. Pfannkuchenstapel werden entweder in Klammern durch Kommas getrennt (z.B. $(2, 7, 1, 8)$) oder als nicht getrennte Ziffern dargestellt (z.B. 2718). Bei der zweiten Notation können Pfannkuchen dann maximal die Breite 9 haben, um die Eindeutigkeit der Darstellung zu gewährleisten.

1.2 Sortieren

Die möglichen Stapel können in einem gerichteten Graphen dargestellt werden (Siehe Abbildung 1). Die Knoten des Graphen sind die Stapel, die Kanten sind die Operationen. Die Kanten sind gerichtet, denn eine PWUE-Operation wandelt einen Stapel in einen anderen um. Die Identität eines Stapels wird durch die Reihenfolge der Pfannkuchen bestimmt, nicht deren genauen Größen. Das heißt, dass zum Beispiel die Stapel $(10, 12, 3)$ und $(2, 3, 1)$ gleich sind, denn die Elemente haben die gleiche Reihenfolge.

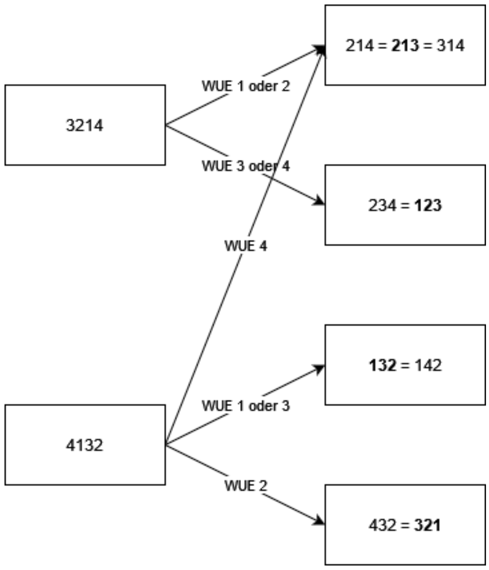

Abbildung 1: Ausschnitt aus dem Graph der Pfannkuchenstapel. Die Kanten sind beschriftet mit der zugehörigen Operation.

Das hängt damit zusammen, dass der Zielstapel nur durch seine aufsteigende Reihenfolge definiert ist. Äquivalente Stapel können in eine kanonische Form gebracht werden, in dem die kleinste Größe durch 1, die zweitkleinste durch 2 und so weiter ersetzt werden. Dadurch ist $(2,3,1)$ in kanonischer Form. Die kanonische Form kann folgendermaßen bestimmt werden:

procedure Kanonische Form(Stapel S)
  Initialisiere Integer-Array V der Länge $\max(S) - \min(S) + 1$
  Setze jeden Wert von V auf -1
  for $s \in S$ **do**
    $V_{s-\min(S)} \leftarrow 1$
  end for
  Initalisiere Zähler $c = 1$
  for $s \in (0, 1, \ldots, |V| - 1)$ **do**
    if $V_s \neq -1$ **then**
      $V_s \leftarrow c$
      $c \leftarrow c + 1$
    end if
  end for
  Initalisiere Integer-Array E der Länge $|S|$
  for $s \in (0, 1, \ldots, |S| - 1)$ **do**
    $E_s \leftarrow V_{s-\min(S)}$
  end for
  return E
end procedure

Dieser Algorithmus konstruiert im Array V eine Abbildung der Zahlenwerte der ursprünglichen Pfannkuchen in jene der kanonischen Form, wobei der Array-Index den Parameter der Abbildung darstellt. Dafür wird zunächst jeder Index, der zu einer Breite im ursprünglichen Stapel gehört, markiert. Die markierten Indices werden dann aufsteigend nummeriert, der kleinste bekommt also den Wert 1, der zweitkleinste 2 und so weiter. So ist die Konstruktion der Abbildung vollständig. Im letzten Schritt wird die Abbildung auf die Elemente von S angewandt. Da die Abbildung nur für bestimmte Werte zwischen einschließlich dem kleinsten und größten von S definiert ist, kann die Länge des Arrays auf $\max(S) - \min(S)$ begrenzt werden. Für die Abbildung muss deshalb immer noch $\min(S)$ vom Parameter abgezogen werden.

Um die Operationen zu bestimmen, die einen Stapel optimal sortieren, muss ein kürzester Pfad im Graphen vom Stapel zu einem sortierten Stapel gefunden werden. Dafür lässt sich Dijkstra's Algorithmus [Dijkstra, 1959] verwenden.

procedure Dijkstra's Algorithmus(Stapel S)

```
    Intialisiere Prioritätswarteschlange Q
    Initialisiere Map V
    Initialisiere Map C
    Füge S mit Priotität 0 in Q ein
    Füge S mit Vorgänger () in V ein
    Füge S mit Kosten 0 in C ein
    while Q nicht leer do
        Entferne Knoten S mit der niedrigsten Priorität aus Q
        if S ist sortiert then
            Rekonstruiere Pfad von S zu () mit V
        end if
        for alle w ∈ W_n do
            S' ← wS
            k ← C(S) + 1
            if S' nicht in C oder K < C(S') then
                Füge S' mit Priorität k in Q ein
                Füge S' mit Vorgänger S in V ein
                Füge S' mit Kosten k in C ein
            end if
        end for
    end while
end procedure
```

Dieser Algorithmus hat außer der Länge der Permutationskette keine Informationen über die Stapel. Da Dijkstras Algorithmus alle kürzeren erkundeten Pfade erweitert bevor ein längerer Pfad erweitert wird, ist er hier sehr langsam. Schnellere Ergebnisse lassen sich mit Hilfe vom A*-Algorithmus [Hart et al., 1968] erreichen. Dieser Algorithmus ähnelt Dijkstras Algorithmus, verwendet aber eine Heuristik, welche die Distanz zum Ziel schätzt. Die Heuristik darf die tätsächliche Entferung zum Ziel niemals überschätzen. Mit $H(S)$ als Heuristik für den Stapel S lautet der Algorithmus:

```
procedure A*-Algorithmus(Stapel S)
    Intialisiere Prioritätswarteschlange Q
    Initialisiere Map V
    Initialisiere Map C
    Füge S mit Priotität 0 in Q ein
    Füge S mit Vorgänger () in V ein
    Füge S mit Kosten 0 in C ein
    while Q nicht leer do
        Entferne Knoten S mit der niedrigsten Priorität aus Q
        if S ist sortiert then
            Rekonstruiere Pfad von S zu () mit V
        end if
        for alle w ∈ W_n do
            S' ← wS
            k ← C(S) + 1 − H(S) + H(S')
            if S' nicht in C oder K < C(S') then
                Füge S' mit Priorität k in Q ein
                Füge S' mit Vorgänger S in V ein
                Füge S' mit Kosten k in C ein
            end if
        end for
    end while
end procedure
```

Eine Heuristik für das Pfannkuchen sortieren ist die Anzahl der Adjazenzen [Gates and Papadimitriou, 1979]. Als Adjazenz bezeichne ich zwei Pfannkuchen die direkt nebeneinander im Stapel liegen und für die es keinen Pfannkuchen gibt, dessen Größe zwischen den beiden liegt. Mit Hilfe der Adjazenzen lässt sich eine untere Schranke für die Anzahl der Sortierschritte eines Stapels berechnen.

Lemma 1. *Ein Stapel der Höhe h mit a_0 Adjazenzen kann in nicht weniger als $\lceil \frac{h-a_0}{3} \rceil$ Schritten sortiert werden.*

Beweis. In einer Operation können sich höchstens zwei neue Adjazenzen bilden. Weil in einer Operation sich nur die Nachbarn von zwei Pfannkuchen ändern, (nämlich des obersten, der nach unten gewendet wird, und dessen, der direkt unter dem Pfannenwender liegen bleibt) kann sich nur zwischen diesen beiden eine Adjazenz bilden. Eine weitere Adjazenz lässt sich dadurch bilden, dass der aufgegessene Pfannkuchen die Breite zwischen zwei nebeneinanderliegenden hatte, welche nach der Operation keine Pfannkuchen mit Größe zwischen ihnen haben. Als Ajdazenz wird auch gezählt, wenn der größte Pfannkuchen ganz unten liegt. Ein sortierter Stapel der Höhe n hat n Adjazenzen. Seien a_0 die Anzahl der Adjazenzen im untersuchten Stapel, h die Höhe des Stapels und n die Anzahl der Sortieroperationen. Weiterhin seien a_f und h_f die Anzahl der Adjazenzen und die Höhe des sortierten Stapels. Dann gilt:

I.	$a_f = h_f$	Adjazenzen und Höhe des Stapels müssen gleich sein
II.	$a_f \leq a_0 + 2n$	Pro Operation können höchstens zwei neue Adjazenzen entstehen
III.	$h_f = h - n$	Der Stapel wird in jedem Schritt um einen Pfannkuchen kleiner

II. und III. in I. einsetzen:

$$\begin{aligned} & a_0 + 2n \geq h - n \\ \leftrightarrow \quad & n \geq \frac{h - a_0}{3} \end{aligned}$$

Weil $n \in \mathbb{N}^+$ kann aufgerundet werden:

$$n \geq \lceil \frac{h - a_0}{3} \rceil$$

□

Als untere Schranke kann diese Erkenntnis als für den A*-Algorithmus geeignete Heuristik verwendet werden, mit $a(S)$ als Anzahl der Adjazenzen im Stapel S und $h(S)$ als Höhe des Stapels S:

$$H(S) = \lceil \frac{h(S) - a(S)}{3} \rceil$$

1.3 PWUE-Zahl

Die PWUE-Zahl kann rekursiv mit Hilfe von dynamischer Programmierung berechnet werden. Dafür definieren wir die Funktion $K(n,a) = \{s \in \mathsf{P}_n \mid A(s) = a\}$, die die Menge aller Stapel der Höhe n enthält, die in mindestens a Schritten sortiert werden können. Die Funktion lässt sich rekursiv berechnen:

$$\begin{aligned} K(n,a) &= \{wS, w \in \mathsf{W}_{n-1}^{-1}, S \in K(n-1, a-1)\} | \forall v \in \mathsf{W}_n : A(vwS) \geq a - 1\} \\ &= \{wS, w \in \mathsf{W}_{n-1}^{-1}, S \in K(n-1, a-1)\} | \forall v \in \mathsf{W}_n : \exists b \geq a - 1 : A(vwS) = b\} \\ &= \{wS, w \in \mathsf{W}_{n-1}^{-1}, S \in K(n-1, a-1)\} | \forall v \in \mathsf{W}_n : \exists b \geq a - 1 : vwS \in K(n-1, b)\} \end{aligned}$$

$K(n,a)$ enthält also alle Stapel, die durch eine Umkehroperation aus Stapeln der Höhe $n-1$ mit mindestens $a-1$ Sortieroperationen entstehen können und für die keine andere Sortieroperation eine Stapel bildet, der in weniger als $a-1$ Schritten sortiert werden kann. Nach dieser Definition würde $K(n,1)$ allerdings auch die komplett sortierten Stapel enthalten, weshalb noch die Bedingung $(a > 1) \vee (s \notin K(n,0))$ ergänzt werden muss. Da die komplett sortierten Stapel nicht weiter sortiert werden müssen, setzen wir $K(n,0) = \{(1,\ldots,n)\}$, es handelt sich dabei um das Ende der Rekursion. Die Funktion $K(n,a)$ ist also definiert als

$K(n,0) = \{(1,\ldots,n)\}$
$K(n,a) = \{wS, w \in \mathsf{W}_{n-1}^{-1}, S \in K(n-1, a-1)\} | \forall v \in \mathsf{W}_n : \exists b \geq a-1 : vwS \in K(n-1,b) \wedge ((a > 1) \vee (S \notin K(n,0)))\}$

Dass diese Definition richtig ist, lässt sich überprüfen durch die Substitution $A(S) = k, S \in \mathsf{P}_n \iff (\forall w \in \mathsf{W}_n : A(ws) \geq k - 1) \wedge (\exists w \in \mathsf{W}_n : A(ws) = k - 1)$. Ein Problem bei der Berechnung dieser Funktion ist, dass $\exists b \geq a - 1 : vwS \in K(n-1, b) \wedge ((a > 1) \vee (S \notin K(n, 0)))$ nicht begrenzt ist, sondern alle natürlichen Zahlen durchprobieren müsste. Das ist natürlich Unfug, denn wir können nicht alle natürlichen Zahlen in endlicher Zeit durchprobieren. Dass wir das nicht brauchen, zeigt folgendes Lemma:

Lemma 2. *Jeder Stapel der Höhe $3n + b$ mit $n, b \in \mathbb{N}$ und $b < 3$ kann in weniger als oder gleich $2n + 1$ Schritten sortiert werden.*

Beweis. Wir beweisen durch starke Induktion nach n. Für $n = 0$ kann der Stapel die Höhe 1 oder 2 haben. Im ersten Fall ist er Stapel schon sortiert und die Aussage ist erfüllt. Im zweiten Fall ist der Stapel sortiert oder noch falsch herum. In beiden Fällen ist nicht mehr als $2n + 1 = 1$ Sortierschritt notwendig.
Für einen Stapel mit $n > 0$ liegen die m größten Pfannkuchen in richtiger Reihenfolge ganz unten. Wenn $m \geq 3$ muss somit nur der darüberliegende Stapel mit $3m + c; m < n$ Pfannkuchen sortiert werden. Das ist nach Induktion in $2m + 1$ Schritten möglich, was kleiner als $2n + 1$ ist, wodurch die Aussage erfüllt ist. Wenn $m < 3$, können wir den $m + 1$t-größten Pfannkuchen in einer Operation nach oben bringen und in einer zweiten an die richtige Stelle am Ende. Danach hat der unsortierte Teil des Stapels, also alles vor den letzten $m + 1$ Pfannkuchen, die Höhe $3n + b - 2 - (m + 1) = 3n + b - 3 - m = 3(n - 1) + b - m$, weil zwei Pfannkuchen verspeist wurden und der sortierte Teil um einen größer geworden ist. Nach der Induktion kann dieser Teil der Höhe $3(n-1)+b-m$ in $2(n-1)+1$ Schritten sortiert werden. Zusammen mit den zwei Wendeoperationen wurde der Stapel in $2(n - 1) + 1 + 2 = 2n + 1$ Schritten sortert. □

Es folgt sofort, dass für einen Stapel der Höhe h gilt $n = \lfloor \frac{h}{3} \rfloor$ und dieser deshalb in $2\lfloor \frac{h}{3} \rfloor + 1$ Schritten sortiert werden kann. Allerdings lässt sich dadurch nicht direkt die PWUE-Zahl bestimmen, es handelt sich lediglich um eine obere Schranke für diese. Da jeder Stapel $S \in \mathsf{P}_n$ in $2\lfloor \frac{h}{3} \rfloor + 1$ Schritten sortiert werden kann, reicht es aus, die Funktion $K(n, a)$ für alle $\lceil \frac{n}{1.5} \rceil$ zu berechnen. Damit lässt sich auch $\exists b \geq a - 1 : vws \in K(n-1, b)$ durch $\exists \lceil \frac{n}{1.5} \rceil \geq b \geq a - 1 : vwS \in K(n-1, b)$ ersetzen, wodurch nicht unendlich viele Werte für b ausprobiert werden müssen. Um jetzt die PWUE-Zahl zu berechnen, muss nur noch die Funktion $K(n, a)$ für alle $a \leq 2\lfloor \frac{h}{3} \rfloor + 1$ berechnet werden und überprüft werden, ob sie Elemente enthält.

2 Laufzeit und Speicherbedarf

2.1 Sortieren

Der A*-Algorithmus hat für einen Graph mit Kanten V und Knoten E eine Laufzeit in[1] $\mathcal{O}(|E| \log |V|)$ und einen Speicherbedarf in $\mathcal{O}(V)$ [Sedgewick and Wayne, 2011, 654]. Wenn wir einen Ausgangsstapel der Höhe h haben, sind die Knoten des zu untersuchenden Graphs $V = \dot\bigcup_{n=1}^{h-1} \mathsf{P}_n$. Da es $n!$ Permutationen von n Elementen gibt, ist $|V| = \sum_{n=1}^{h-1} n!$. Diese Summe ist in $\mathcal{O}(h!)$:

$$\begin{aligned}
|V| &= \sum_{n=1}^{h-1} n! \\
&= (h-1)! \cdot (1 + \frac{1}{h-1} + \frac{1}{(h-1)(h-2)} + \ldots + \frac{1}{(h-1)!}) \\
&= (h-1)! \cdot (1 + \mathcal{O}(1)) \\
&= \mathcal{O}((h-1)!) && |\log \\
\log |V| &= \mathcal{O}((h-1)\log(h-1))
\end{aligned}$$

[1] Da die Landau-Symbole Mengen von Funktionen darstellen, ist es mathematisch korrekt zu sagen, die Zeit (-funktion) ist in $\mathcal{O}(f(x))$. In den Gleichungen weiter unten knüpfe ich an die verbreitere Notation an, in der $\mathcal{O}(f(x))$ für einen anonymen Funktionsterm dieser Menge steht.

Von einem Knoten, der zu einem Stapel der Höhe n gehört, gibt es maximal n verschiedene PWUE-Operationen und somit auch maximal n ausgehende Kanten. Das heißt

$$\begin{aligned}
|E| &= \sum_{n=1}^{h-1} n \cdot |\mathsf{P}_n| \\
&= \sum_{n=1}^{h-1} n \cdot n! \\
&= (h-1) \cdot (h-1)! \cdot (1 + \frac{h-2}{(h-1)^2} + \frac{h-3}{(h-1)^2(h-2)} + \dots \frac{1}{(h-1) \cdot (h-1)!}) \\
&= (h-1) \cdot (h-1)! \cdot (1 + \mathcal{O}(1)) \\
&= \mathcal{O}((h-1) \cdot (h-1)!) \\
&= \mathcal{O}(h!)
\end{aligned}$$

Demnach wäre die Laufzeit $\mathcal{O}(h! \cdot (h-1) \cdot \log(h-1))$ und der Speicherbedarf $\mathcal{O}(h!)$. Leider ist das etwas zu kurz gedacht, denn in jedem Expansionsschritt des Algorithmus muss für jeden neuen Knoten noch die kanonische Form gebildet werden. Außerdem kann der zu erweiternde Knoten nicht direkt mit einem Zielknoten verglichen werden, sondern es muss geprüft werden, ob seine Pfannkuchen in aufsteigender Reihenfolge sind. Diese Prüfung nimmt pro Stapel eine Zeit in $\mathcal{O}(h)$ in Anspruch, da jeder Pfannkuchen mit seinem Vorgänger verglichen wird und jeder Stapel nicht mehr als h Pfannkuchen enthält, was zu h Vergleichsoperationen führt. Im schlechtesten Fall wird maximal jeder der Knoten überprüft, das heißt diese Operation nimmt eine Zeit in $\mathcal{O}(h \cdot |V|) = \mathcal{O}(h!)$ in Anspruch. Dieser Term wächst langsamer als $h! \cdot (h-1) \cdot \log(h-1)$, die Zeit bleibt also in $\mathcal{O}(h! \cdot (h-1) \cdot \log(h-1) + h!) = \mathcal{O}(h! \cdot (h-1) \cdot \log(h-1))$. Die Prüfung auf Sortiertheit nimmt keinen zusätzlichen Speicher in Anspruch.
Betrachten wir nun die Zeit, in der die Stapel in kanonische Form gebracht werden. Die benötigte Zeit liegt für einen Stapel liegt in $\mathcal{O}(h)$: Um $\min(S)$ und $\max(S)$ zu finden, muss der Stapel, der nicht mehr als h Pfannkuchen enthält, durchgegangen werden. Dann wird der gleiche Stapel noch einmal durchgegangen, um die Werte im Array V zu ändern, wo wieder nicht mehr als h Iterationen benötigt werden. Dann werden die Elemente des Array V durchgegangen. Dieses hat die Länge $\max(S) - \min(S) + 1$. Diese Länge ist auch niemals größer als h, weil der größte Pfannkuchen die Breite h und der kleinste die Breite 1 hat. Zuletzt wird noch einmal S durchgegangen, um die Abbildung anzuwenden. Diese Operation wird für jeden neu erkundeten Stapel durchgeführt, also für jede Kante. Insgesamt wird dadurch die Zeit $\mathcal{O}(h \cdot |V|) = \mathcal{O}(h \cdot (h-1)!) = \mathcal{O}(h!)$ benötigt, was auch langsamer als der schon ermittelte Term für die Zeit wächst und somit vernachlässigt werden kann. Der zusätzliche Speicherbedarf liegt in $\mathcal{O}(h)$ für das Array V, was auch vernachlässigbar ist. Die Kanonisierung eines Stapels der Höhe h muss mindestens in $\mathcal{O}(n)$ liegen, da das Ergebnis der Länge n ja ausgegeben muss. Da die Höhe der Stapel zu h proportional ist, ist die Zeit $\mathcal{O}(h)$ meines Algorithmus somit optimal.

2.2 PWUE-Zahl

Um die PWUE-Zahl der Höhe h zu berechnen, wird $K(n,a)$ höchstens für alle $1 \leq n \leq h$ und $0 \leq a \leq 2\lfloor \frac{n}{3} \rfloor + 1$ berechnet. Die Anzahl der notwendigen Berechnungen B von $K(n,a)$ ist also:

$$\begin{aligned}
B &= \sum_{n=1}^{h} 2\lfloor \frac{n}{3} \rfloor + 1 \\
&= \sum_{n=1}^{h} \mathcal{O}(n) && = \sum_{n=1}^{h} \mathcal{O}(h) = h \cdot \mathcal{O}(h) && = \mathcal{O}(h^2)
\end{aligned}$$

Die Größe von $K(n,a)$ ist:

$$|K(n,a)| \leq |$$

3 Umsetzung

Zur Lösung der ersten Aufgabe habe ich Python verwendet. Für die zweite Aufgabe habe ich Java verwendet. Die Lösung der ersten Aufgabe verlangt einen Dateipfad zum Pfannkuchenstapel. Diese Datei ist in dem Format, welches die Beispieleingaben auf der BwInf-Webseite haben. Das Programm sucht die Lösung und gibt dann die PWUE-Operationen sowie die Zwischenstände des Stapels aus. Die Notation weicht etwas von der hier verwendeten ab, denn die einzelnen Pfannkuchen werden nicht durch Kommata sondern durch Leerzeichen getrennt und der Stapel ist nicht umklammert.

4 Beispiele

4.1 Sortieren

Die Beispiele `pancake0.txt` bis `pancake7.txt` sind direkt von den BwInf-Webseiten übernommen. Die Beispiele `pancake2h.txt` bis `pancake11h.txt` sind die in der zweiten Teilaufgabe gefundenen aufwändigsten Pfannkuchenstapel mit Höhe 2-11. Die Beispiele sind im Ordner `beispiele` zu finden.
Eingabe: `pancake0.txt`
Ausgabe:

```
-- schritte --
3 2 4 5 1
Wende erste 5
5 4 2 3
Wende erste 4
2 4 5
```

Eingabe: `pancake1.txt`
Ausgabe:

```
-- schritte --
6 3 1 7 4 2 5
Wende erste 7
2 4 7 1 3 6
Wende erste 3
4 2 1 3 6
Wende erste 4
1 2 4 6
```

Eingabe: `pancake2.txt`
Ausgabe:

```
-- schritte --
8 1 7 5 3 6 4 2
Wende erste 6
3 5 7 1 8 4 2
Wende erste 6
8 1 7 5 3 2
Wende erste 2
8 7 5 3 2
Wende erste 5
3 5 7 8
```

Eingabe: `pancake3.txt`
Ausgabe:

```
-- schritte --
5 10 1 11 4 8 2 9 7 3 6
Wende erste 5
11 1 10 5 8 2 9 7 3 6
Wende erste 2
11 10 5 8 2 9 7 3 6
Wende erste 9
3 7 9 2 8 5 10 11
Wende erste 5
2 9 7 3 5 10 11
Wende erste 3
9 2 3 5 10 11
Wende erste 1
2 3 5 10 11
```

Eingabe: `pancake4.txt`
Ausgabe:

```
-- schritte --
7 4 11 5 10 6 1 13 12 9 3 8 2
Wende erste 4
11 4 7 10 6 1 13 12 9 3 8 2
Wende erste 7
1 6 10 7 4 11 12 9 3 8 2
Wende erste 8
12 11 4 7 10 6 1 3 8 2
Wende erste 10
8 3 1 6 10 7 4 11 12
Wende erste 4
1 3 8 10 7 4 11 12
Wende erste 5
10 8 3 1 4 11 12
Wende erste 5
1 3 8 10 11 12
```

Eingabe: `pancake5.txt`
Ausgabe:

```
-- schritte --
4 13 10 8 2 3 7 9 14 1 12 6 5 11
Wende erste 1
13 10 8 2 3 7 9 14 1 12 6 5 11
Wende erste 13
5 6 12 1 14 9 7 3 2 8 10 13
Wende erste 7
9 14 1 12 6 5 3 2 8 10 13
Wende erste 9
2 3 5 6 12 1 14 9 10 13
Wende erste 5
6 5 3 2 1 14 9 10 13
Wende erste 6
1 2 3 5 6 9 10 13
```

Eingabe: `pancake6.txt`
Ausgabe:

```
-- schritte --
14 8 4 12 13 2 1 15 7 11 3 9 5 10 6
Wende erste 15
10 5 9 3 11 7 15 1 2 13 12 4 8 14
Wende erste 2
10 9 3 11 7 15 1 2 13 12 4 8 14
Wende erste 9
2 1 15 7 11 3 9 10 12 4 8 14
Wende erste 5
7 15 1 2 3 9 10 12 4 8 14
Wende erste 2
7 1 2 3 9 10 12 4 8 14
Wende erste 8
12 10 9 3 2 1 7 8 14
Wende erste 8
7 1 2 3 9 10 12 14
Wende erste 1
1 2 3 9 10 12 14
```

Eingabe: `pancake7.txt`
Ausgabe:

```
-- schritte --
8 5 10 15 3 7 13 6 2 4 12 9 1 14 16 11
Wende erste 16
16 14 1 9 12 4 2 6 13 7 3 15 10 5 8
Wende erste 15
5 10 15 3 7 13 6 2 4 12 9 1 14 16
```

```
Wende erste 3
10 5 3 7 13 6 2 4 12 9 1 14 16
Wende erste 8
2 6 13 7 3 5 10 12 9 1 14 16
Wende erste 4
13 6 2 3 5 10 12 9 1 14 16
Wende erste 9
9 12 10 5 3 2 6 13 14 16
Wende erste 1
12 10 5 3 2 6 13 14 16
Wende erste 6
2 3 5 10 12 13 14 16
```

Eingabe: `pancake2h.txt`
Ausgabe:

```
-- schritte --
2 1
Wende erste 1
1
```

Eingabe: `pancake3h.txt`
Ausgabe:

```
-- schritte --
2 3 1
Wende erste 1
3 1
Wende erste 1
1
```

Eingabe: `pancake4h.txt`
Ausgabe:

```
-- schritte --
1 4 3 2
Wende erste 3
4 1 2
Wende erste 1
1 2
```

Eingabe: `pancake5h.txt`
Ausgabe:

```
-- schritte --
1 4 2 5 3
Wende erste 2
1 2 5 3
Wende erste 1
2 5 3
Wende erste 2
2 3
```

Eingabe: `pancake6h.txt`
Ausgabe:

```
-- schritte --
1 5 6 3 2 4
Wende erste 1
5 6 3 2 4
Wende erste 3
6 5 2 4
Wende erste 4
2 5 6
```

Eingabe: `pancake7h.txt`
Ausgabe:

```
-- schritte --
1 6 2 7 3 5 4
Wende erste 1
6 2 7 3 5 4
Wende erste 4
7 2 6 5 4
Wende erste 2
7 6 5 4
Wende erste 4
5 6 7
```

Eingabe: `pancake8h.txt`
Ausgabe:

```
-- schritte --
4 8 1 6 2 7 5 3
Wende erste 1
8 1 6 2 7 5 3
Wende erste 4
6 1 8 7 5 3
Wende erste 3
1 6 7 5 3
Wende erste 4
7 6 1 3
Wende erste 4
1 6 7
```

Eingabe: `pancake9h.txt`
Ausgabe:

```
-- schritte --
1 8 2 6 9 3 7 5 4
Wende erste 4
2 8 1 9 3 7 5 4
Wende erste 1
8 1 9 3 7 5 4
Wende erste 4
9 1 8 7 5 4
Wende erste 2
9 8 7 5 4
Wende erste 5
5 7 8 9
```

Eingabe: `pancake10h.txt`
Ausgabe:

```
-- schritte --
1 10 2 8 5 7 3 9 6 4
Wende erste 9
9 3 7 5 8 2 10 1 4
Wende erste 6
8 5 7 3 9 10 1 4
Wende erste 2
8 7 3 9 10 1 4
Wende erste 7
1 10 9 3 7 8
Wende erste 2
1 9 3 7 8
Wende erste 2
1 3 7 8
```

Eingabe: `pancake11h.txt`
Ausgabe:

```
-- schritte --
1 10 4 7 5 9 6 11 3 8 2
Wende erste 2
1 4 7 5 9 6 11 3 8 2
Wende erste 6
9 5 7 4 1 11 3 8 2
Wende erste 7
11 1 4 7 5 9 8 2
Wende erste 8
8 9 5 7 4 1 11
Wende erste 3
9 8 7 4 1 11
Wende erste 5
4 7 8 9 11
```

4.2 PWUE-Zahl

```
P(2)=1
Beispiel:
2 1

P(3)=2
Beispiel:
2 3 1

P(4)=2
Beispiel:
1 4 3 2

P(5)=3
Beispiel:
1 4 2 5 3

P(6)=3
Beispiel:
1 5 6 3 2 4

p(7)=4
Beispiel:
1 6 2 7 3 5 4

P(8)=5
Beispiel:
4 8 1 6 2 7 5 3

P(9)=5
Beispiel:
1 8 2 6 9 3 7 5 4

P(10)=6
Beispiel:
1 10 2 8 5 7 3 9 6 4

P(11)=6
Beispiel:
1 10 4 7 5 9 6 11 3 8 2
```

5 Quellcode

```
from queue import PriorityQueue

# finds the shortest path from start node to a node that fullfills target_pred. returns the path
def a_star(start_node, target_pred, adj_func, cost_func, heur_func, count_steps=False):
    if count_steps:
        steps = 0
```

```python
    i = 0
    queue = PriorityQueue()
    queue.put((0, heur_func(start_node), i, start_node))
    prev = {start_node: None}
    cost = {start_node: 0 + heur_func(start_node)}
    while not queue.empty():
        if count_steps:
            steps += 1
        _, _, _, node = queue.get()
        if target_pred(node):
            if count_steps:
                return reconstruct_path(node, prev), steps
            return reconstruct_path(node, prev)
        for adj_node in adj_func(node):
            new_cost = cost[node] - heur_func(node) + cost_func(node, adj_node) + heur_func(adj_node)
            if adj_node not in cost or new_cost < cost[adj_node]:
                i -= 1
                cost[adj_node] = new_cost
                queue.put((new_cost, heur_func(node), i, adj_node))
                prev[adj_node] = node

def reconstruct_path(node, prev):
    path = [node]
    while prev[node] is not None:
        node = prev[node]
        path.append(node)
    return list(reversed(path))
```

a_star.py

```python
import math
from queue import PriorityQueue

# finds the shortest path from start node to a node that fullfills target_pred. returns the path
def a_star(start_node, target_pred, adj_func, cost_func, heur_func, count_steps=False):
    if count_steps:
        steps = 0
    i = 0
    queue = PriorityQueue()
    queue.put((0, heur_func(start_node), i, start_node))
    prev = {start_node: None}
    cost = {start_node: 0 + heur_func(start_node)}
    while not queue.empty():
        if count_steps:
            steps += 1
        _, _, _, node = queue.get()
        if target_pred(node):
            if count_steps:
                return reconstruct_path(node, prev), steps
            return reconstruct_path(node, prev)
        for adj_node in adj_func(node):
            new_cost = cost[node] - heur_func(node) + cost_func(node, adj_node) + heur_func(adj_node)
            if adj_node not in cost or new_cost < cost[adj_node]:
                i -= 1
                cost[adj_node] = new_cost
                queue.put((new_cost, heur_func(node), i, adj_node))
                prev[adj_node] = node

def reconstruct_path(node, prev):
    path = [node]
    while prev[node] is not None:
        node = prev[node]
        path.append(node)
    return list(reversed(path))


# Pfannkuchenstapel umdrehen und Pfannkuchen essen.
def flip(arr, k):
    return arr[: k - 1][::-1] + arr[k:]


# Gibt alle moeglichen naechsten Reihenfolgen zurueck.
def next_arrs(arr):
    for i in range(1, len(arr) + 1):
```

```
        yield normalize(flip(arr, i))


# Zaehlt, wie viele aufeinanderfolgende Pfannkuchen nebeneinander liegen.
def count_adj(arr):
    adj = 0
    for i in range(1, len(arr)):
        if arr[i] - arr[i - 1] in (1, -1):
            adj += 1
    if arr[-1] == max(arr):
        adj += 1
    return adj


# Veraendert die Zahlen in der Liste so, dass sie in [0, ..., n-1] liegen,
# wobei die Reihenfolge erhalten bleibt.
# Algorithmus mit O(n), was auch die kleinstmoegliche Zeitkomplexitaet ist,
# da ja schon die ausgabe des ergebnisses Zeit O(n) braucht
def normalize(arr):
    a_min = min(arr)
    a_max = max(arr)
    values = [-1 for _ in range(a_max - a_min + 1)]
    for item in arr:
        values[item - a_min] = item
    counter = 0
    for i in range(len(values)):
        if values[i] != -1:
            values[i] = counter
            counter += 1
    return tuple(values[x - a_min] for x in arr)


# Naehert die minimale Anzahl von flips()s mit count_adj() an.
def heuristic(arr):
    return math.ceil((len(arr) - count_adj(normalize(arr))) / 3)


# prueft, ob die Liste in der richtigen Reihenfolge ist.
def is_sorted(arr):
    return all(arr[i] <= arr[i + 1] for i in range(len(arr) - 1))


# Gibt die Optimale Reihenfolge von flip()s zurueck, um die Liste zu sortieren.
def least_flips(arr, count_steps=False):
    return a_star(normalize(arr), is_sorted, next_arrs, lambda a, b: 1, heuristic, count_steps)


# Findet die PWUE-Operation von pre zu post.
def find_flip(pre, post):
    for i in range(1, len(pre) + 1):
        if normalize(flip(pre, i)) == normalize(post):
            return i


def main():
    path = input("Pfad: ")
    with open(path) as f:
        n_pancakes = int(f.readline())
        pancakes = tuple(int(x) for x in f.readlines())
    pancakes = normalize(pancakes)
    steps = least_flips(pancakes)

    print("-- schritte --")
    pre = None
    not_normalized = list(map(lambda x: x+1, steps[0]))
    print(" ".join(map(str, not_normalized)))
    for step in steps:
        if pre is not None:
            ix = find_flip(pre, step)
            print("Wende erste", ix)
            not_normalized = flip(not_normalized, ix)
            print(" ".join(map(str, not_normalized)))
        pre = step
```

```python

if __name__ == "__main__":
    main()
```

least_flips.py

```java
import java.util.Arrays;
import java.util.HashMap;
import java.util.HashSet;
import java.util.List;
import java.util.Map;
import java.util.Optional;
import java.util.Scanner;
import java.util.Set;

public class Pwue {
    static class IntPair implements Comparable<IntPair> {
        private int num1;
        private int num2;

        public IntPair(int key, int value) {
            this.num1 = key;
            this.num2 = value;
        }

        public int first() {
            return num1;
        }

        public int second() {
            return num2;
        }

        @Override
        public boolean equals(Object o) {
            if (this == o)
                return true;
            if (o == null || getClass() != o.getClass())
                return false;
            IntPair pair = (IntPair) o;
            return (num1 == pair.num1 && num2 == pair.num2);
        }

        @Override
        public int hashCode() {
            // Polynom-hash mit horner schema
            int hash = 17;
            hash = hash * 31 + num1;
            hash = hash * 31 + num2;
            return hash;
        }

        @Override
        public String toString() {
            return "(" + num1 + ",␣" + num2 + ")";
        }

        @Override
        public int compareTo(Pwue.IntPair o) {
            if (num1 < o.num1) {
                return -1;
            }
            if (num1 > o.num1) {
                return 1;
            }
            if (num2 < o.num2) {
                return -1;
            }
            if (num2 > o.num2) {
                return 1;
            }
            return 0;
        }
```

```
    }

    static Integer[] flipOp(Integer[] a, int i) {
        Integer[] b = new Integer[a.length - 1];
        for (int j = 0; j < i - 1; j++) {
            b[j] = a[i - j - 2];
        }
        for (int j = i; j < a.length; j++) {
            b[j - 1] = a[j];
        }
        return canonical(b);
    }

    static Integer[] revFlipOp(Integer[] a, int i, int n) {
        i--;
        Integer[] b = new Integer[a.length + 1];
        for (int j = 0; j < i; j++) {
            if (a[i - j - 1] >= n) {
                b[j] = a[i - j - 1] + 1;
            } else {
                b[j] = a[i - j - 1];
            }
        }
        b[i] = n;
        for (int j = i; j < a.length; j++) {
            if (a[j] >= n) {
                b[j + 1] = a[j] + 1;
            } else {
                b[j + 1] = a[j];
            }
        }
        return canonical(b);
    }

    // Bringt die Permutation in die kanonische Form, die allerdings bei 0 statt 1 anfaengt.
    // Folgt der Konvention, dass Indices bei 0 anfangen, nicht bei 1, der kleinste Pfannkuchen
    // ist also der nullte hat also den Index 0.
    static Integer[] canonical(Integer[] a) {
        Integer min = Integer.MAX_VALUE;
        Integer max = Integer.MIN_VALUE;
        for (int i = 0; i < a.length; i++) {
            if (a[i] < min)
                min = a[i];
            if (a[i] > max)
                max = a[i];
        }
        Integer[] values = new Integer[max - min + 1];

        for (int i = 0; i < values.length; i++)
            values[i] = -1;

        for (int i = 0; i < a.length; i++)
            values[a[i] - min] = a[i];

        Integer counter = 0;

        for (int i = 0; i < values.length; i++)
            if (values[i] != -1)
                values[i] = counter++;

        Integer[] result = new Integer[a.length];
        for (int i = 0; i < a.length; i++)
            result[i] = values[a[i] - min];

        return result;

    }

    static Integer[] allFlipOps(int n) {
        Integer[] a = new Integer[n];
        for (int i = 0; i < n; i++) {
            a[i] = i + 1;
        }
```

```java
        return a;
    }

    static IntPair[] allRevFlipOps(int n) {
        IntPair[] a = new IntPair[n * (n + 1)];
        for (int i = 0; i < n; i++) {
            for (int j = 0; j <= n; j++) {
                a[i * (n + 1) + j] = new IntPair(i + 1, j);
            }
        }
        return a;
    }

    static Integer[] range(int n) {
        assert n >= 0;
        Integer[] a = new Integer[n];
        for (int i = 0; i < n; i++) {
            a[i] = i;
        }
        return a;
    }

    // Hier werden die Zwischenergebnisse der dynamischen Programmierung gespeichert
    static Map<IntPair, Set<List<Integer>>> memo = new HashMap<>();

    static Set<List<Integer>> k(int n, int a) {
        // Dynamische Programmierung: ggf. schon vorhandenes Ergebnis zurueckgeben
        IntPair key = new IntPair(n, a);
        if (memo.containsKey(key)) {
            return memo.get(key);
        }
        if (a == 0) {
            Set<List<Integer>> result = new HashSet<>();
            result.add(Arrays.asList(range(n)));
            memo.put(key, result);
            return result;
        }
        if (n == 1 && a != 0) {
            Set<List<Integer>> result = new HashSet<>();
            memo.put(key, result);
            return result;
        }
        HashSet<List<Integer>> result = new HashSet<>();
        for (IntPair rFlip : allRevFlipOps(n)) {

            for (List<Integer> seqL : k(n - 1, a - 1)) {
                Integer[] seq = seqL.toArray(new Integer[0]);
                Integer[] rFlipped = revFlipOp(seq, rFlip.first(), rFlip.second());
                if (!(a > 1 || !Arrays.equals(rFlipped, range(n)))) {
                    continue;
                }
                boolean r1 = true;
                boolean r2 = false;
                for (Integer flip : allFlipOps(n)) {
                    for (int b = a - 1; b < 2 * Math.floor(n / 3) + 2; b++) {
                        r2 = false;
                        if (k(n - 1, b).contains(Arrays.asList(flipOp(rFlipped, flip)))) {
                            r2 = true;
                            break;
                        }
                    }
                    r1 = r1 && r2;
                    if (!r1) {
                        break;
                    }
                }
                if (r1) {
                    result.add(Arrays.asList(rFlipped));
                }
            }
        }
        memo.put(key, result);
        return result;
```

```
    }

    // Gibt nur ein Element aus k(n, a) zurueck, falls es eins gibt
    static Optional<Integer[]> kHasSolution(int n, int a) {
        if (a == 0) {
            return Optional.of(range(n));
        }
        if (n == 1 && a != 0) {
            return Optional.empty();
        }
        for (IntPair rFlip : allRevFlipOps(n)) {
            for (List<Integer> seqL : k(n - 1, a - 1)) {
                Integer[] seq = seqL.toArray(new Integer[0]);
                Integer[] rFlipped = revFlipOp(seq, rFlip.first(), rFlip.second());
                if (!(a > 1 || !Arrays.equals(rFlipped, range(n))))
                    continue;

                // forall steht fuer den Existenzquantor ueber P_n
                // exists steht fuer den Allquantor ueber N zwischen a-1 und 2*floor(n/3)+1
                boolean forall = true;
                boolean exists = false;
                for (Integer flip : allFlipOps(n)) {
                    exists = false;
                    for (int b = a - 1; b < 2 * Math.floor(n / 3) + 2; b++) {
                        if (k(n - 1, b).contains(Arrays.asList(flipOp(rFlipped, flip)))) {
                            exists = true;
                            break;
                        }
                    }
                    forall = forall && exists;
                    if (!forall)
                        break;
                }
                if (forall)
                    return Optional.of(rFlipped);
            }
        }
        return Optional.empty();
    }

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.print("n:␣");
        int n = scanner.nextInt();
        scanner.close();
        long startTime = System.currentTimeMillis();
        for (int a = (int) Math.floor(n / 3) * 2 + 1; a > 0; a--) {
            Optional<Integer[]> result = kHasSolution(n, a);
            if (result.isPresent()) {
                System.out.println("max␣a:␣" + a);
                for (int i = 0; i < result.get().length; i++) {
                    System.out.print((result.get()[i] + 1) + "␣");
                }
                System.out.println();
                System.out.println("time:␣" + (System.currentTimeMillis() - startTime) + "ms");
                break;
            }
        }

    }
}
```

Pwue.java

Literatur

[Dijkstra, 1959] Dijkstra, E. W. (1959). A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1):269–271.

[Gates and Papadimitriou, 1979] Gates, W. H. and Papadimitriou, C. H. (1979). Bounds for sorting by prefix reversal. *Discrete mathematics*, 27(1):47–57.

[Hart et al., 1968] Hart, P. E., Nilsson, N. J., and Raphael, B. (1968). A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107.

[Sedgewick and Wayne, 2011] Sedgewick, R. and Wayne, K. D. (2011). *Algorithms*. Addison-Wesley.