# Aufgabe 1: LaTeX-Dokument

Teilnahme-ID: ?????

Bearbeiter/-in dieser Aufgabe:
Vor- und Nachname

11. April 2023

## Inhaltsverzeichnis


**1 Lösungsidee** **1**
1.1 Notation . . . 1
1.2 Sortieren . . . 1
1.3 PWUE-Zahl . . . 4

**2 Laufzeit und Speicherbedarf** **5**
2.1 Sortieren . . . 5

**3 Umsetzung** **6**

**4 Beispiele** **6**

**5 Quellcode** **6**


# 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 $\mathsf{P}_n$ bezeichnet. Die Möglichen Pfannkuchen-Wende-und-Ess-Operationen für einem Stapel mit $n$ Pfannkuchen wird mit $\mathsf{W}_n$ bezeichnet. Die Menge der möglichen Umkehroperationen für solch einen Stapel wird mit $\mathsf{W}_n^{-1}$ 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. $w_1w_2S = w_1(w_2S)$. 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. 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:

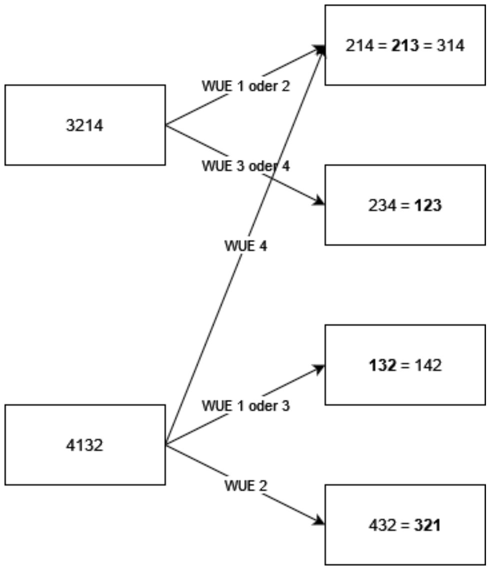


Abbildung 1: Ausschnitt aus dem Graph der Pfannkuchenstapel. Die Kanten sind beschriftet mit der zugehörigen Operation.

**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**
      $V_s \leftarrow c$
      $s \leftarrow s + 1$
    **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 $w \in \mathsf{W}_n$ **do**
$S' \leftarrow wS$
$k \leftarrow C(S) + 1$
**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 $w \in \mathsf{W}_n$ **do**
$S' \leftarrow wS$
$k \leftarrow C(S) + 1 - H(S) + H(S')$
**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:

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

$\square$

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

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

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 $n, b \in mathbbN$ 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 $n > 0$ liegt der größte Pfannkuchen des Stapels entweder ganz unten oder über anderen Pfannkuchen. Im zweiten Fall können wir den Stapel zuerst so wenden, dass der größte Pfannkuchen nach dem Essen ganz oben liegt, und dann den ganzen Stapel wenden, wobei zwei Pfannkuchen verspeist werden. Der Stapel ohne den größten Pfannkuchen ganz unten hat die Höhe $3n+b-3 = 3(n-1)+b$. Weil dieser durch die Induktion in $2(n - 1) + 1$ Schritten sortiert werden kann, brauchen wir mit den beiden PWUE-Operationen $2(n - 1) + 1 + 2 = 2n + 1$ Sortierschritte. Liegt der größte Pfannkuchen ganz unten, können wir die gleiche Fallunterscheidung für den zweitgrößten Pfannkuchen machen. Liegt dieser nicht an zweiter Stelle von hinten, können wir ihn genauso in zwei Schritten an diese Stelle bringen, es bleibt ein Stapel der Höhe $3n+b-4 = 4(n-1)+b-1$ zu sortieren. Das ist auch in $2(n-1)+1$ Schritten möglich, wir sind wieder bei insgesamt $2(n - 1) + 1 + 2 = 2n + 1$ Schritten. Liegt der zweitgrößte Pfannkuchen an zweitletzer Stelle, machen wir das Gleiche mit dem drittgrößten. Liegt dieser an drittletzer Stelle, müssen nur die darüber liegenden $3n + b - 3 = 3(n - 1) + b$ Pfannkuchen sortiert werden, was nach der Induktion in $2(n - 1) + 1 \leq 2n + 1$ Schritten möglich ist. ☐

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. 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 $a$ 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 $|V| = \Sigma_{n=1}^{h-1} n!$. Diese Summe ist in $\mathcal{O}(h!)$:

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

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

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

[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.

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.

## 3 Umsetzung

Zur Lösung der ersten Aufgabe habe ich Python verwendet. Für die zweite Aufgabe habe ich Java verwendet.

## 4 Beispiele

Genügend Beispiele einbinden! Die Beispiele von der BwInf-Webseite sollten hier diskutiert werden, aber auch eigene Beispiele sind sehr gut – besonders wenn sie Spezialfälle abdecken. Aber bitte nicht 30 Seiten Programmausgabe hier einfügen!

## 5 Quellcode

```python
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

```

```python
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 a_star import a_star

# 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)


def find_flip(pre, post):
    for i in range(1, len(pre) + 1):
        if normalize(flip(pre, i)) == post:
            return i


def main():
```

```python
    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(steps)
    print(len(steps) - 1, "Schritte")
    print(len(pancakes), "Pfannkuchen")
    print(heuristic(pancakes), "heuristik")
    print(math.ceil(len(pancakes) / 1.5), "upper bound")
    print(len(pancakes) / 2, "lower bound")

    print("-- schritte --")
    pre = None
    not_normalized = steps[0]
    print(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(not_normalized)
        pre = step


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() {
            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;
        }

        public void setFirst(int num1) {
            this.num1 = num1;
        }

        public void setSecond(int num2) {
            this.num2 = num2;
        }

        public void swap() {
            int temp = num1;
            num1 = num2;
            num2 = temp;
        }
    }

    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 normalize(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 normalize(b);
    }

    // O(n)
```

```java
    static Integer[] normalize(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;
        }
        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) {
        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, int depth) {
        // 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;
        }
```

```java
        System.out.println("|␣".repeat(depth) + "|␣k(" + n + ",␣" + a + ")␣"
                + ((int) Math.ceil(n / 1.5) - a + 1) * n * (n + 1)
                        * k(n - 1, a - 1, depth + 1).size()
                + "␣Its");
        HashSet<List<Integer>> result = new HashSet<>();
        for (IntPair rFlip : allRevFlipOps(n)) {

            for (List<Integer> seqL : k(n - 1, a - 1, depth + 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 < Math.ceil(n / 1.5); b++) {
                        r2 = false;
                        if (k(n - 1, b, depth + 1)
                                .contains(Arrays.asList(flipOp(rFlipped, flip)))) {
                            r2 = true;
                            break;
                        }
                    }
                    r1 = r1 && r2;
                    if (!r1) {
                        break;
                    }
                }
                if (r1) {
                    result.add(Arrays.asList(rFlipped));
                }
            }
        }
        System.out.println("|␣".repeat(depth) + "|␣k(" + n + ",␣" + a + ")␣Fertig␣␣␣␣␣␣␣␣␣␣");
        memo.put(key, result);
        return result;
    }

    static Optional<Integer[]> kHasSolution(int n, int a) {
        for (IntPair rFlip : allRevFlipOps(n)) {
            for (List<Integer> seqL : k(n - 1, a - 1, 0)) {
                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, 0).contains(Arrays.asList(flipOp(rFlipped, flip)))) {
                            r2 = true;
                            break;
                        }
                    }
                    r1 = r1 && r2;
                    if (!r1) {
                        break;
                    }
                }
                if (r1) {
                    return Optional.of(rFlipped);
                }
            }
        }
        return Optional.empty();
    }

    public static void main(String[] args) {
        System.out.println(Arrays.deepToString(allRevFlipOps(5)));
```

```
        Scanner scanner = new Scanner(System.in);
        System.out.print("n: ");
        int n = scanner.nextInt();
        scanner.close();
        for (int a = (int) Math.ceil(n / 1.5); a > 0; a--) {
            Optional<Integer[]> result = kHasSolution(n, a);
            if (result.isPresent()) {
                System.out.println("max a: " + a);
                System.out.println(Arrays.deepToString(result.get()));
                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.