

Aufgabe 1: L^AT_EX-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 SStapel ist 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 einen 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 $S \in P_n$ durch die Operation $w \in W_n$ 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. 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:

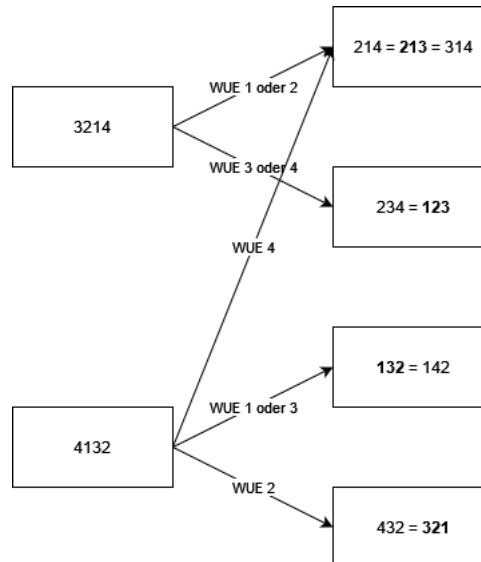


Abbildung 1: Ausschnitt aus dem Graph der Pfannkuchentapel. Die Kanten sind beschriftet mit der zugehörigen Operation.

```

procedure KANONISCHE FORM(Stackel  $S$ )
  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
  Initialisiere Zähler  $c = 1$ 
  for  $s \in (0, 1, \dots, |V| - 1)$  do
    if  $V_s \neq -1$  then
       $V_s \leftarrow c$ 
       $c \leftarrow c + 1$ 
    end if
  end for
  Initialisiere Integer-Array  $E$  der Länge  $|S|$ 
  for  $s \in (0, 1, \dots, |S| - 1)$  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(Stackel  $S$ )
  Initialisiere Prioritätswarteschlange  $Q$ 
  Initialisiere Map  $V$ 
  Initialisiere Map  $C$ 
  Füge  $S$  mit Priorität 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  $w \in W_n$  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 tatsächliche Entfernung zum Ziel niemals überschätzen. Mit $H(S)$ als Heuristik für den Stapel S lautet der Algorithmus:

```

procedure A*-ALGORITHMUS(Stapel  $S$ )
  Initialisiere 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  $w \in W_n$  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 aufgegebene Pfannkuchen die Breite zwischen zwei nebeneinanderliegenden hatte, welche nach der Operation keine Pfannkuchen mit Größe zwischen ihnen haben. Als Adjazenz 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:

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

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 $K(n, a) = \{s \in \mathcal{P}_n \mid A(s) = a\}$, die die Menge aller Stapel der Höhe n enthält, die in mindestens a Schritten sortiert werden können. Die Funktion lässt sich rekursiv berechnen:

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

$K(n, a)$ 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 einen 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 $K(n, 0) = \{(1, \dots, n)\}$, es handelt sich dabei um das Ende der Rekursion. Die Funktion $K(n, a)$ ist also definiert als

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

Dass diese Definition richtig ist, lässt sich überprüfen durch die Substitution $A(S) = k, S \in \mathcal{P}_n \iff (\forall w \in \mathcal{W}_n : A(ws) \geq k-1) \wedge (\exists w \in \mathcal{W}_n : A(ws) = k-1)$. 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 $m + 1$ -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 sortiert. \square

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 $S \in \mathcal{P}_n$ 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 $\mathcal{O}(|E| \log |V|)$ 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 $V = \bigcup_{n=1}^{h-1} \mathcal{P}_n$. Da es $n!$ Permutationen von n Elementen gibt, ist $|V| = \sum_{n=1}^{h-1} n!$. Diese Summe ist in $\mathcal{O}(h!)$:

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

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

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

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

¹Da 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 verbreitete Notation an, in der $\mathcal{O}(f(x))$ für einen anonymen Funktionsterm dieser Menge steht.

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

```

1 from queue import PriorityQueue

3 # finds the shortest path from start node to a node that fullfills target_pred. returns the path
4 def a_star(start_node, target_pred, adj_func, cost_func, heur_func, count_steps=False):
5     if count_steps:
6         steps = 0
7     i = 0
8     queue = PriorityQueue()
9     queue.put((0, heur_func(start_node), i, start_node))
10    prev = {start_node: None}
11    cost = {start_node: 0 + heur_func(start_node)}
12    while not queue.empty():
13        if count_steps:
14            steps += 1
15        _, _, _, node = queue.get()
16        if target_pred(node):
17            if count_steps:
18                return reconstruct_path(node, prev), steps
19            return reconstruct_path(node, prev)
20        for adj_node in adj_func(node):
21            new_cost = cost[node] - heur_func(node) + cost_func(node, adj_node) + heur_func(adj_node)
22            if adj_node not in cost or new_cost < cost[adj_node]:
23                i += 1
24                cost[adj_node] = new_cost
25                queue.put((new_cost, heur_func(node), i, adj_node))
26                prev[adj_node] = node
27
28 def reconstruct_path(node, prev):
29     path = [node]

```

```

    while prev[node] is not None:
31         node = prev[node]
            path.append(node)
33     return list(reversed(path))

a_star.py

import math
2 from a_star import a_star

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

8
9 # Gibt alle moeglichen naechsten Reihenfolgen zurueck.
10 def next_arrs(arr):
    for i in range(1, len(arr) + 1):
12         yield normalize(flip(arr, i))

14
15 # Zaehlt, wie viele aufeinanderfolgende Pfannkuchen nebeneinander liegen.
16 def count_adj(arr):
    adj = 0
18     for i in range(1, len(arr)):
        if arr[i] - arr[i - 1] in (1, -1):
20             adj += 1
        if arr[-1] == max(arr):
22             adj += 1
    return adj

24
25 # Veraendert die Zahlen in der Liste so, dass sie in [0, ..., n-1] liegen,
# wobei die Reihenfolge erhalten bleibt.
28 # Algorithmus mit O(n), was auch die kleinstmoegliche Zeitkomplexitaet ist,
# da ja schon die ausgabe des ergebnisses Zeit O(n) braucht
30 def normalize(arr):
    a_min = min(arr)
32     a_max = max(arr)
    values = [-1 for _ in range(a_max - a_min + 1)]
34     for item in arr:
        values[item - a_min] = item
36     counter = 0
    for i in range(len(values)):
38         if values[i] != -1:
            values[i] = counter
            counter += 1
40     return tuple(values[x - a_min] for x in arr)

42
43 # Naehert die minimale Anzahl von flips()s mit count_adj() an.
def heuristic(arr):
46     return math.ceil((len(arr) - count_adj(normalize(arr))) / 3)

48
49 # prueft, ob die Liste in der richtigen Reihenfolge ist.
50 def is_sorted(arr):
    return all(arr[i] <= arr[i + 1] for i in range(len(arr) - 1))

52
53 # Gibt die Optimale Reihenfolge von flip()s zurueck, um die Liste zu sortieren.
def least_flips(arr, count_steps=False):
56     return a_star(normalize(arr), is_sorted, next_arrs, lambda a, b: 1, heuristic, count_steps)

58
59 def find_flip(pre, post):
60     for i in range(1, len(pre) + 1):
        if normalize(flip(pre, i)) == post:
62             return i

64
65 def main():
66     path = input("Pfad: ")
    with open(path) as f:

```

```

68     n_pancakes = int(f.readline())
        pancakes = tuple(int(x) for x in f.readlines())
70     pancakes = normalize(pancakes)
        steps = least_flips(pancakes)
72     print(steps)
        print(len(steps) - 1, "Schritte")
74     print(len(pancakes), "Pfannkuchen")
        print(heuristic(pancakes), "heuristik")
76     print(math.ceil(len(pancakes) / 1.5), "upper_bound")
        print(len(pancakes) / 2, "lower_bound")
78
        print("-- schritte --")
80     pre = None
        not_normalized = steps[0]
82     print(not_normalized)
        for step in steps:
84         if pre is not None:
            ix = find_flip(pre, step)
86             print("Wende_erste", ix)
            not_normalized = flip(not_normalized, ix)
88             print(not_normalized)
            pre = step
90
92 if __name__ == "__main__":
    main()

```

least_flips.py

```

1 import java.util.Arrays;
import java.util.HashMap;
3 import java.util.HashSet;
import java.util.List;
5 import java.util.Map;
import java.util.Optional;
7 import java.util.Scanner;
import java.util.Set;
9
10 public class Pwue {
11     static class IntPair implements Comparable<IntPair> {
12         private int num1;
13         private int num2;
14
15
16         public IntPair(int key, int value) {
17             this.num1 = key;
18             this.num2 = value;
19         }
20
21         public int first() {
22             return num1;
23         }
24
25         public int second() {
26             return num2;
27         }
28
29         @Override
30         public boolean equals(Object o) {
31             if (this == o)
32                 return true;
33             if (o == null || getClass() != o.getClass())
34                 return false;
35             IntPair pair = (IntPair) o;
36             return (num1 == pair.num1 && num2 == pair.num2);
37         }
38
39         @Override
40         public int hashCode() {
41             int hash = 17;
42             hash = hash * 31 + num1;
43             hash = hash * 31 + num2;
44             return hash;
45         }
46     }
47 }

```



```

47     @Override
48     public String toString() {
49         return "(" + num1 + ", " + num2 + ")";
50     }
51
52     @Override
53     public int compareTo(Pwue.IntPair o) {
54         if (num1 < o.num1) {
55             return -1;
56         }
57         if (num1 > o.num1) {
58             return 1;
59         }
60         if (num2 < o.num2) {
61             return -1;
62         }
63         if (num2 > o.num2) {
64             return 1;
65         }
66         return 0;
67     }
68
69     public void setFirst(int num1) {
70         this.num1 = num1;
71     }
72
73     public void setSecond(int num2) {
74         this.num2 = num2;
75     }
76
77     public void swap() {
78         int temp = num1;
79         num1 = num2;
80         num2 = temp;
81     }
82 }
83
84 static Integer[] flipOp(Integer[] a, int i) {
85     Integer[] b = new Integer[a.length - 1];
86     for (int j = 0; j < i - 1; j++) {
87         b[j] = a[i - j - 2];
88     }
89     for (int j = i; j < a.length; j++) {
90         b[j - 1] = a[j];
91     }
92     return normalize(b);
93 }
94
95 static Integer[] revFlipOp(Integer[] a, int i, int n) {
96     i--;
97     Integer[] b = new Integer[a.length + 1];
98     for (int j = 0; j < i; j++) {
99         if (a[i - j - 1] >= n) {
100             b[j] = a[i - j - 1] + 1;
101         } else {
102             b[j] = a[i - j - 1];
103         }
104     }
105     b[i] = n;
106     for (int j = i; j < a.length; j++) {
107         if (a[j] >= n) {
108             b[j + 1] = a[j] + 1;
109         } else {
110             b[j + 1] = a[j];
111         }
112     }
113     return normalize(b);
114 }
115
116 // O(n)
117 static Integer[] normalize(Integer[] a) {
118     Integer min = Integer.MAX_VALUE;

```

```

119     Integer max = Integer.MIN_VALUE;
120     for (int i = 0; i < a.length; i++) {
121         if (a[i] < min)
122             min = a[i];
123         if (a[i] > max)
124             max = a[i];
125     }
126     Integer[] values = new Integer[max - min + 1];
127
128     for (int i = 0; i < values.length; i++)
129         values[i] = -1;
130
131     for (int i = 0; i < a.length; i++)
132         values[a[i] - min] = a[i];
133
134     Integer counter = 0;
135
136     for (int i = 0; i < values.length; i++)
137         if (values[i] != -1)
138             values[i] = counter++;
139
140     Integer[] result = new Integer[a.length];
141     for (int i = 0; i < a.length; i++)
142         result[i] = values[a[i] - min];
143
144     return result;
145 }
146
147 static Integer[] allFlipOps(int n) {
148     Integer[] a = new Integer[n];
149     for (int i = 0; i < n; i++) {
150         a[i] = i + 1;
151     }
152     return a;
153 }
154
155 static IntPair[] allRevFlipOps(int n) {
156     IntPair[] a = new IntPair[n * (n + 1)];
157     for (int i = 0; i < n; i++) {
158         for (int j = 0; j <= n; j++) {
159             a[i * (n + 1) + j] = new IntPair(i + 1, j);
160         }
161     }
162     return a;
163 }
164
165 static Integer[] range(int n) {
166     assert n >= 0;
167     Integer[] a = new Integer[n];
168     for (int i = 0; i < n; i++) {
169         a[i] = i;
170     }
171     return a;
172 }
173
174 // Hier werden die Zwischenergebnisse der dynamischen Programmierung gespeichert
175 static Map<IntPair, Set<List<Integer>>> memo = new HashMap<>();
176
177
178 static Set<List<Integer>> k(int n, int a) {
179     // Dynamische Programmierung: ggf. schon vorhandenes Ergebnis zurueckgeben
180     IntPair key = new IntPair(n, a);
181     if (memo.containsKey(key)) {
182         return memo.get(key);
183     }
184     if (a == 0) {
185         Set<List<Integer>> result = new HashSet<>();
186         result.add(Arrays.asList(range(n)));
187         memo.put(key, result);
188         return result;
189     }
190     HashSet<List<Integer>> result = new HashSet<>();

```

```

193     for (IntPair rFlip : allRevFlipOps(n)) {
194         for (List<Integer> seqL : k(n - 1, a - 1)) {
195             Integer[] seq = seqL.toArray(new Integer[0]);
196             Integer[] rFlipped = revFlipOp(seq, rFlip.first(), rFlip.second());
197             if (!(a > 1 || !Arrays.equals(rFlipped, range(n)))) {
198                 continue;
199             }
200             boolean r1 = true;
201             boolean r2 = false;
202             for (Integer flip : allFlipOps(n)) {
203                 for (int b = a - 1; b < 2 * Math.floor(n / 3) + 2; b++) {
204                     r2 = false;
205                     if (k(n - 1, b).contains(Arrays.asList(flipOp(rFlipped, flip)))) {
206                         r2 = true;
207                         break;
208                     }
209                 }
210                 r1 = r1 && r2;
211                 if (!r1) {
212                     break;
213                 }
214             }
215             if (r1) {
216                 result.add(Arrays.asList(rFlipped));
217             }
218         }
219     }
220     memo.put(key, result);
221     return result;
222 }
223
224 static Optional<Integer[]> kHasSolution(int n, int a) {
225     for (IntPair rFlip : allRevFlipOps(n)) {
226         for (List<Integer> seqL : k(n - 1, a - 1)) {
227             Integer[] seq = seqL.toArray(new Integer[0]);
228             Integer[] rFlipped = revFlipOp(seq, rFlip.first(), rFlip.second());
229             if (!(a > 1 || !Arrays.equals(rFlipped, range(n)))) {
230                 continue;
231             }
232
233             boolean r1 = true;
234             boolean r2 = false;
235             for (Integer flip : allFlipOps(n)) {
236                 for (int b = a - 1; b < 2 * Math.floor(n / 3) + 2; b++) {
237                     r2 = false;
238                     if (k(n - 1, b).contains(Arrays.asList(flipOp(rFlipped, flip)))) {
239                         r2 = true;
240                         break;
241                     }
242                 }
243                 r1 = r1 && r2;
244                 if (!r1) {
245                     break;
246                 }
247             }
248             if (r1) {
249                 return Optional.of(rFlipped);
250             }
251         }
252     }
253     return Optional.empty();
254 }
255
256 public static void main(String[] args) {
257     System.out.println(Arrays.deepToString(allRevFlipOps(5)));
258     Scanner scanner = new Scanner(System.in);
259     System.out.print("n: ");
260     int n = scanner.nextInt();
261     scanner.close();
262     for (int a = (int) Math.ceil(n / 1.5); a > 0; a--) {
263         Optional<Integer[]> result = kHasSolution(n, a);
264         if (result.isPresent()) {

```

```
265         System.out.println("max_a: " + a);
266         System.out.println(Arrays.deepToString(result.get()));
267         break;
268     }
269 }
270
271 }
}
```

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.