

Aufgabe 1: L^AT_EX-Dokument

Teilnahme-ID: ?????

Bearbeiter/-in dieser Aufgabe:
Vor- und Nachname

15. April 2023

Inhaltsverzeichnis

1	Lösungsidee	1
1.1	Notation	1
1.2	Sortieren	1
1.3	PWUE-Zahl	4
2	Laufzeit und Speicherbedarf	6
2.1	Sortieren	6
2.2	PWUE-Zahl	7
3	Erweiterung: Beliebige große Eingabezahlen	9
4	Umsetzung	9
5	Beispiele	9
5.1	Sortieren	9
5.2	PWUE-Zahl	13
6	Quellcode	14

1 Lösungsidee

1.1 Notation

Mit SS Stapel 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. Pfannkuchentapel 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

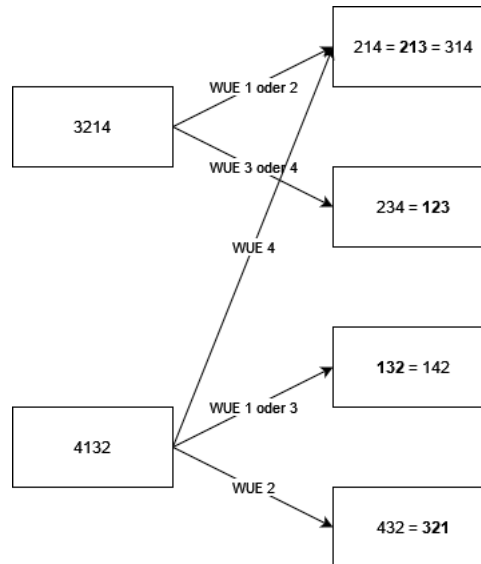


Abbildung 1: Ausschnitt aus dem Graph der Pfannkuchenstapel. Die Kanten sind beschriftet mit der zugehörigen Operation.

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:

```

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
  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(Stapel  $S$ )
  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 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
       $S' \leftarrow wS$ 
       $k \leftarrow C(S) + 1 - H(S) + H(S')$ 
      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 $A(S) = k, S \in P_n \iff (\forall w \in W_n : A(ws) \geq k - 1) \wedge (\exists w \in W_n : A(ws) = k - 1)$. 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 $S \in P_n$ 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:

```

procedure PWUE-ZAHL(Höhe  $h$ )
  Initialisiere Map  $\tilde{K}$ 
  Initialisiere Map  $\tilde{S}$ 
  for  $i \in (2\lfloor \frac{h}{3} \rfloor + 1, 2\lfloor \frac{h}{3} \rfloor, \dots, 0)$  do
    if  $K(h, i) \neq \emptyset$  then return  $i, K(h, i)$ 
    end if
  end for
end procedure
procedure K(Höhe  $n$ , Schritte  $a$ )
  if  $\tilde{K}$  enthält Schlüssel  $(n, a)$  then return  $\tilde{K}[(n, a)]$ 
  end if
  if  $a = 0$  then
     $\tilde{K}[(n, a)] \leftarrow (1, 2, \dots, n)$ 
     $\tilde{S}[(1, 2, \dots, n)] \leftarrow 0$  return  $\{(1, 2, \dots, n)\}$ 
  end if
  if  $a > 0 \wedge n = 1$  then
     $\tilde{K}[(n, a)] \leftarrow (1, 2, \dots, n)$  return  $\{(1, 2, \dots, n)\}$ 
  end if
  Initialisiere leere Menge  $R$ 
  for  $w \in W_{n-1}^{-1}, S \in K(n - 1, a - 1)$  do
    if  $\neg(a > 1 \vee ws = (1, 2, \dots, n))$  then
      continue
    end if
    Initialisiere boolesche Variable  $A$  mit wahr
    for  $v \in W_n$  do
      if  $\tilde{S}$  enthält Schlüssel  $vwS$  then
        if  $\tilde{S}[vwS] < a - 1$  then

```

```

        A ← falsch
        break
    else
        continue
    end if
end if
Initialisiere boolesche Variable E mit falsch
for b ∈ (a − 1, a, ..., 2⌊ $\frac{h}{3}$ ⌋ + 1) do
    if vwS ∈ K(n − 1, b) then
        E ← wahr
        break
    end if
end for
A ← A ∧ E
if ¬A then
    break
end if
end for
if A then
    R ← R ∪ {wS}
     $\tilde{S}[wS] \leftarrow a$ 
end if
end for
 $\tilde{K}[(n, a)] \leftarrow R$  return R
end procedure

```

Dieser Algorithmus kann noch schneller gemacht werden, in dem in der Prozedur "PWUE-Zahl" für $K(n, a)$ nicht alle Elemente gesucht werden, sondern das erste zurückgegeben wird, denn dort sind nicht alle Elemente von $K(n, a)$ notwendig, sondern irgendeines, falls es existiert. Die Aufgabenstellung lautet ja nur einen Stapel zu finden, der in der größten Zahl von Schritten sortiert werden kann.

2 Laufzeit und Speicherbedarf

2.1 Sortieren

Der A*-Algorithmus hat für einen Graph mit Kanten V und Knoten E eine Laufzeit in¹ $\mathcal{O}(|E| \log |V|)$ 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 $V = \dot{\bigcup}_{n=1}^{h-1} P_n$. Da es $n!$ Permutationen von n Elementen gibt, ist $|V| = \sum_{n=1}^{h-1} n!$. Diese Summe ist in $\mathcal{O}(h!)$:

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

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.

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

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

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 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.

Zusammenfassend ist die Zeit in $\mathcal{O}(h! \cdot (h-1) \cdot \log(h-1))$ und der Speicherbedarf in $\mathcal{O}(h!)$.

Abbildung 2 zeigt, dass die tatsächliche Zeit zwar exponentiell wächst (lineares Wachstum auf einer logarithmischen Skala), wie man aber in Abbildung 2a sieht, ist die Steigung kleiner als die theoretische Analyse ergeben hat. Trotzdem scheint die Zeit in $\mathcal{O}(h! \cdot (h-1) \cdot \log(h-1))$ zu liegen, denn ab einem bestimmten Punkt ist die Zeit kleiner als eine Funktion dieser Menge.

2.2 PWUE-Zahl

Um $K(n, a)$ zu ermitteln, werden alle $S \in K(n-1, a-1)$, $w \in W_{n-1}^{-1}$, $v \in W_n$ durchgegangen. $|W_{n-1}^{-1}| = (n+1)^2$, weil wir $(n+1)$ verschiedene Pfannkuchen auf den Stapel legen können (den kleinsten, zweitkleinsten, ..., größten) und dann $(n+1)$ verschiedene Wendeoperationen durchführen können. $|W_n| = n$, weil wir nur den ersten Pfannkuchen wenden, die ersten zwei Pfannkuchen wenden, usw. können, bis wir alle n Pfannkuchen wenden.

Da die Summe alle möglichen Permutationen umfassen muss, gilt:

$$\sum_{a=0}^{2\lfloor \frac{n}{3} \rfloor + 1} |K(n, a)| = n!$$

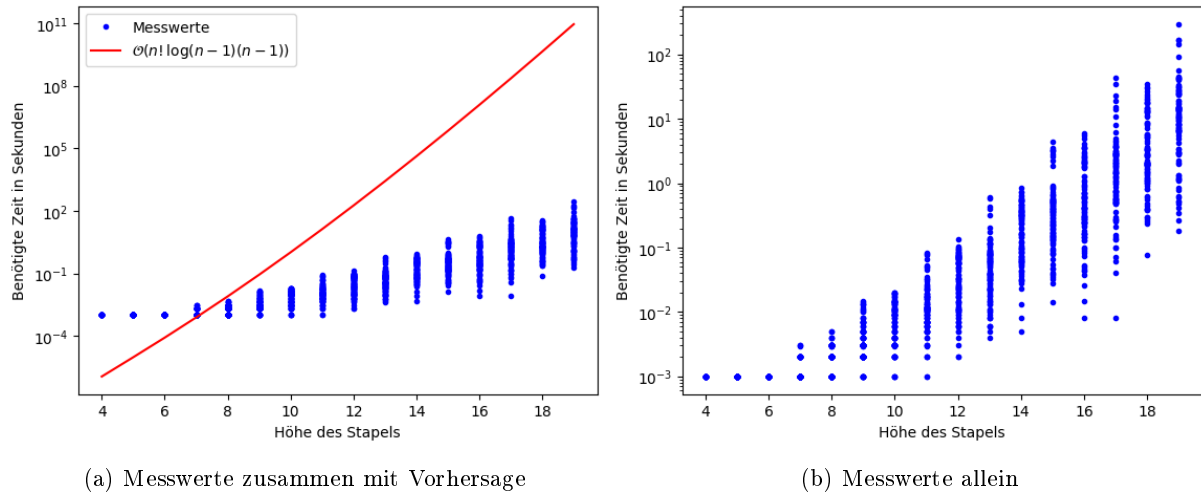


Abbildung 2: Die Diagramme zeigen die benötigte Zeit für jeweils 100 zufällige Stapel unterschiedlicher Höhen. Man bemerke die logarithmische Skala der Zeit. Durch die logarithmische Skala wird die Steigung unabhängig vom Koeffizienten der $\mathcal{O}(n!(n-1)\log(n-1))$ -Funktion, dieser verschiebt den Graphen nur.

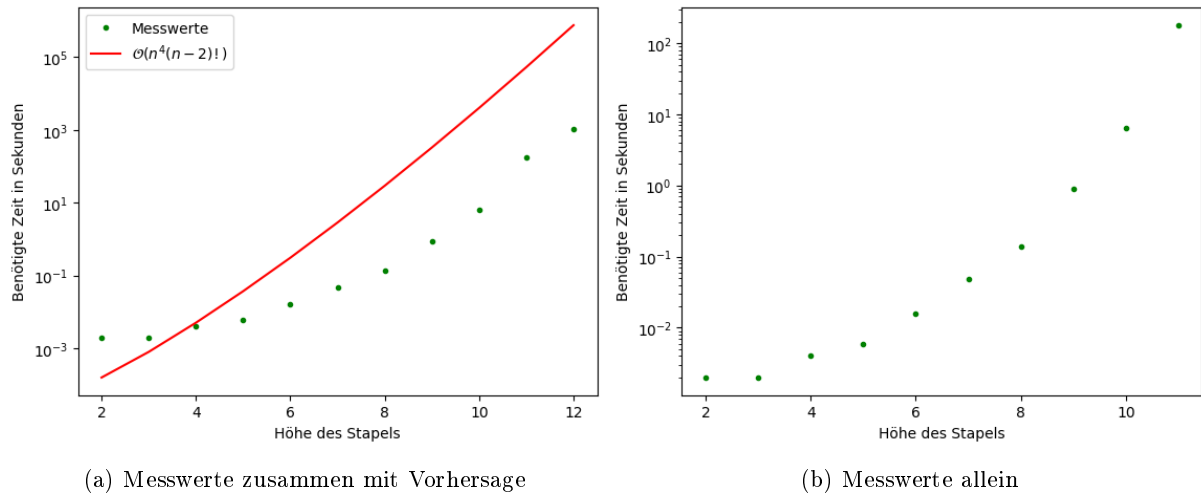


Abbildung 3: Zeiten, um verschiedene PWUE-Zahlen zu berechnen

Wenn wir nun für ein n alle $K(n, a)$ berechnen, benötigen wir

$$\sum_{a=0}^{2\lfloor \frac{n}{3} \rfloor + 1} |K(n-1, a-1)| \cdot |W_{n-1}^{-1}| \cdot |W_n| = |W_{n-1}^{-1}| \cdot |W_n| \cdot \sum_{a=0}^{2\lfloor \frac{n}{3} \rfloor + 1} |K(n-1, a-1)| = (n+1)(n+1)n \cdot \sum_{a=0}^{2\lfloor \frac{n}{3} \rfloor + 1} |K(n-1, a-1)|$$

Kombinationen. Da wir n für alle $1 \leq n \leq h-1$ berechnen müssen, um die PWUE-Zahl zu ermitteln, werden insgesamt

$$\sum_{n=1}^{h-1} \mathcal{O}(n^3(n-1)!) = \mathcal{O}((h-1)^3(h-2)! \cdot (h-1)) = \mathcal{O}(h^3(h-2)!)$$

Kombinationen durchprobiert. Für eine Kombination wird eine Zeit in $\mathcal{O}(h)$ verwendet, da die Kanonisierung wie oben erläutert diese Zeit benötigt, und die Vergleichs- und Wendeoperationen so wie die Hashfunktionen der Hashmap-Datenstrukturen auch lineare Zeit benötigen. Die Zeit liegt also in $\mathcal{O}(h^4(h-2)!)$. Da auch alle Kombinationen gespeichert werden und eine Kombination Länge in $\mathcal{O}(h)$ hat, ist die Speicherkomplexität auch in $\mathcal{O}(h^4(h-2)!)$.

Auch bei diesem Aufgabenteil steigt die theoretische Zeit steiler an als die tatsächliche. Die tatsächliche Zeit hat aber eine stärkere Krümmung, was darauf schließen lässt, dass die Zeit für größere Werte schneller steigt.

3 Erweiterung: Beliebige große Eingabezahlen

Ich habe meine Lösung so erweitert, dass auch Stapel sortiert werden können, deren Zahlen nicht bloß eine Permutation von aufeinanderfolgenden Zahlen sind, sondern beliebig große Lücken dazwischen liegen können. Dafür habe ich einen zweiten Algorithmus zur Kanonisierung entwickelt: Eine Kopie des Stapels wird sortiert und dann wird für jeden Pfannkuchen geschaut, an welcher Stelle er sich befindet. Diese Stelle ist dann die neue Breite. So hat der kleinste Pfannkuchen die erste Stelle und somit die Größe 1, der zweitgrößte Größe 2 und so weiter. Das Sortieren mit geläufigen Algorithmen braucht Zeit in $\mathcal{O}(n \log(n))$. Um für jeden der Pfannkuchen seine Stelle zu finden, kann binäre Suche verwendet werden, um für jeden Pfannkuchen Zeit in $\mathcal{O}(\log(n))$ zu benötigen, insgesamt also nochmal $\mathcal{O}(n \log(n))$. Es kann auch auf triviale Weise in $\mathcal{O}(n^2)$ gesucht werden, wenn wir für jeden Pfannkuchen den sortierten Stapel durchgehen. Da der Rest des Algorithmus sehr viel mehr Zeit braucht, macht die Wahl des Suchalgorithmus hier keinen großen Unterschied.

4 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. Die zweite Lösung fragt nach der Zahl n , für die die PWUE-Zahl berechnet werden soll. Nach Eingabe dieser wird die PWUE-Zahl berechnet und zusammen mit einem Beispiel für einen solchen Stapel ausgegeben.

5 Beispiele

5.1 Sortieren

Die Beispiele `pancake0.txt` bis `pancake7.txt` sind direkt von den BwInf-Webseiten übernommen. Die Beispiele `pancake2h.txt` bis `pancake12h.txt` sind die in der zweiten Teilaufgabe gefundenen aufwändigsten Pfannkuchenstapel mit Höhe 2-12. Die Beispiele sind im Ordner `beispiele` zu finden.

Eingabe: `pancake0.txt`

Ausgabe:

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

Eingabe: `pancake1.txt`

Ausgabe:

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

Eingabe: `pancake2.txt`

Ausgabe:

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

```
Wende erste 6
6 8 1 7 5 3 2
Wende erste 2
8 8 7 5 3 2
Wende erste 5
10 3 5 7 8
```

Eingabe: pancake3.txt

Ausgabe:

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

Eingabe: pancake4.txt

Ausgabe:

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

Eingabe: pancake5.txt

Ausgabe:

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

Eingabe: pancake6.txt

Ausgabe:

```
-- schritte --
2 14 8 4 12 13 2 1 15 7 11 3 9 5 10 6
  Wende erste 15
4 10 5 9 3 11 7 15 1 2 13 12 4 8 14
  Wende erste 2
6 10 9 3 11 7 15 1 2 13 12 4 8 14
  Wende erste 9
8 2 1 15 7 11 3 9 10 12 4 8 14
  Wende erste 5
10 7 15 1 2 3 9 10 12 4 8 14
  Wende erste 2
12 7 1 2 3 9 10 12 4 8 14
  Wende erste 8
14 12 10 9 3 2 1 7 8 14
  Wende erste 8
16 7 1 2 3 9 10 12 14
  Wende erste 1
18 1 2 3 9 10 12 14
```

Eingabe: pancake7.txt

Ausgabe:

```
-- schritte --
2 8 5 10 15 3 7 13 6 2 4 12 9 1 14 16 11
  Wende erste 16
4 16 14 1 9 12 4 2 6 13 7 3 15 10 5 8
  Wende erste 15
6 5 10 15 3 7 13 6 2 4 12 9 1 14 16
  Wende erste 3
8 10 5 3 7 13 6 2 4 12 9 1 14 16
  Wende erste 8
10 2 6 13 7 3 5 10 12 9 1 14 16
  Wende erste 4
12 13 6 2 3 5 10 12 9 1 14 16
  Wende erste 9
14 9 12 10 5 3 2 6 13 14 16
  Wende erste 1
16 12 10 5 3 2 6 13 14 16
  Wende erste 6
18 2 3 5 10 12 13 14 16
```

Eingabe: pancake2h.txt

Ausgabe:

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

Eingabe: pancake3h.txt

Ausgabe:

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

Eingabe: pancake4h.txt

Ausgabe:

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

6 1 2

Eingabe: pancake5h.txt

Ausgabe:

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

Eingabe: pancake6h.txt

Ausgabe:

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

Eingabe: pancake7h.txt

Ausgabe:

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

Eingabe: pancake8h.txt

Ausgabe:

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

Eingabe: pancake9h.txt

Ausgabe:

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

```
8 9 1 8 7 5 4
   Wende erste 2
10 9 8 7 5 4
   Wende erste 5
12 5 7 8 9
```

Eingabe: pancake10h.txt

Ausgabe:

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

Eingabe: pancake11h.txt

Ausgabe:

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

Eingabe: pancake12h.txt

Ausgabe:

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

5.2 PWUE-Zahl

```

P(2)=1
2 Beispiel:
2 1

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

8
P(4)=2
10 Beispiel:
1 4 3 2

12
P(5)=3
14 Beispiel:
1 4 2 5 3

16
P(6)=3
18 Beispiel:
1 5 6 3 2 4

20
p(7)=4
22 Beispiel:
1 6 2 7 3 5 4

24
P(8)=5
26 Beispiel:
4 8 1 6 2 7 5 3

28
P(9)=5
30 Beispiel:
1 8 2 6 9 3 7 5 4

32
P(10)=6
34 Beispiel:
1 10 2 8 5 7 3 9 6 4

36
P(11)=6
38 Beispiel:
1 10 4 7 5 9 6 11 3 8 2

40
P(12)=7
42 Beispiel:
1 9 2 10 8 4 7 12 6 11 5 3

```

6 Quellcode

```

1 from queue import PriorityQueue

3 # 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):
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

```

```

        cost[adj_node] = new_cost
25         queue.put((new_cost, heur_func(node), i, adj_node))
        prev[adj_node] = node
27
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 queue import PriorityQueue

4 # findet den kuerzesten pfad von start_node zu einem knoten, der target_pred erfuellt.
def a_star(start_node, target_pred, adj_func, cost_func, heur_func, count_steps=False):
6     if count_steps:
        steps = 0
8     i = 0
    queue = PriorityQueue()
10    queue.put((0, heur_func(start_node), i, start_node))
    prev = {start_node: None}
12    cost = {start_node: 0 + heur_func(start_node)}
    while not queue.empty():
14        if count_steps:
            steps += 1
16        _, _, _, node = queue.get()
        if target_pred(node):
18            if count_steps:
                return reconstruct_path(node, prev), steps
            return reconstruct_path(node, prev)
        for adj_node in adj_func(node):
22            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]:
24                i -= 1
                cost[adj_node] = new_cost
26                queue.put((new_cost, heur_func(node), i, adj_node))
                prev[adj_node] = node
28
30 # gibt den pfad von node zum anfang zurueck.
def reconstruct_path(node, prev):
32     path = [node]
    while prev[node] is not None:
34         node = prev[node]
        path.append(node)
36     return list(reversed(path))
38
# Pfannkuchenstapel umdrehen und Pfannkuchen essen.
40 def flip(arr, k):
    return arr[: k - 1][::-1] + arr[k:]
42
44 # Gibt alle moeglichen naechsten Stapel zurueck.
def next_arrs(arr):
46     for i in range(1, len(arr) + 1):
        yield canonize(flip(arr, i))
48
50 # Zaehlt, wie viele aufeinanderfolgende Pfannkuchen nebeneinander liegen.
def count_adj(arr):
52     adj = 0
    for i in range(1, len(arr)):
54         if arr[i] - arr[i - 1] in (1, -1):
            adj += 1
56     if arr[-1] == max(arr):
        adj += 1
58     return adj
60
# Veraendert die Zahlen in der Liste so, dass sie in [0, ..., n-1] liegen,

```

```

62 # wobei die Reihenfolge erhalten bleibt.
63 # Algorithmus mit O(n), was auch die kleinstmögliche Zeitkomplexität ist,
64 # da ja schon die Ausgabe des Ergebnisses Zeit O(n) braucht
65 def canonize(arr):
66     a_min = min(arr)
67     a_max = max(arr)
68     values = [-1 for _ in range(a_max - a_min + 1)]
69     for item in arr:
70         values[item - a_min] = item
71     counter = 0
72     for i in range(len(values)):
73         if values[i] != -1:
74             values[i] = counter
75             counter += 1
76     return tuple(values[x - a_min] for x in arr)

78 # Zweite Implementierung von normalize, die langsamer ist, aber besser mit
79 # grösseren Zahlen funktioniert.
80 def canonize2(arr):
81     return tuple(sorted(arr).index(x) for x in arr)

82 # Nähert die minimale Anzahl von flips()s mit count_adj() an.
83 def heuristic(arr):
84     return math.ceil((len(arr) - count_adj(canonize(arr))) / 3)
85
86
88 # prüft, ob die Liste in der richtigen Reihenfolge ist.
89 def is_sorted(arr):
90     return all(arr[i] <= arr[i + 1] for i in range(len(arr) - 1))
91
92 # Gibt die Optimale Reihenfolge von flip()s zurück, um die Liste zu sortieren.
93 def least_flips(arr, count_steps=False):
94     return a_star(canonize(arr), is_sorted, next_arrs, lambda a, b: 1, heuristic, count_steps)
95
96
98 # Findet die PWUE-Operation von pre zu post.
99 def find_flip(pre, post):
100     for i in range(1, len(pre) + 1):
101         if canonize(flip(pre, i)) == canonize(post):
102             return i
103
104
105 def main():
106     # stapel einlesen
107     path = input("Pfad: ")
108     with open(path) as f:
109         n_pancakes = int(f.readline())
110         pancakes = tuple(int(x) for x in f.readlines())
111
112     # Da nach Erweiterung beliebig grosse Zahlen verwendet werden koennen,
113     # muss die langsamere Version verwendet werden.
114     pancakes = canonize2(pancakes)
115     steps = least_flips(pancakes)
116
117     print("-- Schritte --")
118     pre = None
119     not_normalized = list(map(lambda x: x+1, steps[0]))
120     print(" ".join(map(str, not_normalized)))
121     for step in steps:
122         if pre is not None:
123             ix = find_flip(pre, step)
124             print("Wende ", ix)
125             not_normalized = flip(not_normalized, ix)
126             print(" ".join(map(str, not_normalized)))
127         pre = step
128
129
130 if __name__ == "__main__":
131     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
public class Pwue {
11     static class IntPair implements Comparable<IntPair> {
        private int num1;
13         private int num2;

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

19         public int first() {
21             return num1;
        }

23         public int second() {
25             return num2;
        }

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

37         @Override
39         public int hashCode() {
            // Polynom-hash mit horner schema
41             int hash = 17;
            hash = hash * 31 + num1;
43             hash = hash * 31 + num2;
            return hash;
45         }

47         @Override
49         public String toString() {
            return "(" + num1 + ", " + num2 + ")";
51         }

53         @Override
55         public int compareTo(Pwue.IntPair o) {
            if (num1 < o.num1) {
57                 return -1;
            }
            if (num1 > o.num1) {
59                 return 1;
            }
            if (num2 < o.num2) {
61                 return -1;
            }
            if (num2 > o.num2) {
63                 return 1;
            }
65             return 0;
67         }
    }

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

```

```

75         for (int j = i; j < a.length; j++) {
76             b[j - 1] = a[j];
77         }
78         return canonical(b);
79     }

81     static Integer[] revFlipOp(Integer[] a, int pos, int newSize) {
82         Integer[] b = new Integer[a.length + 1];
83         b[0] = newSize;
84         for (int i = 0; i < a.length; i++) {
85             if (a[i] >= newSize) {
86                 b[i + 1] = a[i] + 1;
87             } else {
88                 b[i + 1] = a[i];
89             }
90         }
91         for (int i = 0; i < pos / 2; i++) {
92             int tmp = b[i];
93             b[i] = b[pos - i - 1];
94             b[pos - i - 1] = tmp;
95         }
96         //System.out.println("revFlipOp(" + Arrays.toString(a) + ", " + pos + ", " + newSize + ") = " +
97         return canonical(b);
98     }

99     // Bringt die Permutation in die kanonische Form, die allerdings bei 0 statt 1 anfaengt.
100    // Folgt der Konvention, dass Indices bei 0 anfangen, nicht bei 1, der kleinste Pfannkuchen
101    // ist also der nullte hat also den Index 0.
102    static Integer[] canonical(Integer[] a) {
103        Integer min = Integer.MAX_VALUE;
104        Integer max = Integer.MIN_VALUE;
105        for (int i = 0; i < a.length; i++) {
106            if (a[i] < min)
107                min = a[i];
108            if (a[i] > max)
109                max = a[i];
110        }
111        Integer[] values = new Integer[max - min + 1];
112
113        for (int i = 0; i < values.length; i++)
114            values[i] = -1;
115
116        for (int i = 0; i < a.length; i++)
117            values[a[i] - min] = a[i];
118
119        Integer counter = 0;
120
121        for (int i = 0; i < values.length; i++)
122            if (values[i] != -1)
123                values[i] = counter++;
124
125        Integer[] result = new Integer[a.length];
126        for (int i = 0; i < a.length; i++)
127            result[i] = values[a[i] - min];
128
129        return result;
130    }

131 }

132
133 static Integer[] allFlipOps(int n) {
134     Integer[] a = new Integer[n];
135     for (int i = 0; i < n; i++) {
136         a[i] = i + 1;
137     }
138     return a;
139 }

140
141 static IntPair[] allRevFlipOps(int n) {
142     IntPair[] a = new IntPair[(n + 1) * (n + 1)];
143     for (int pos = 0; pos <= n; pos++) {
144         for (int newSize = 0; newSize <= n; newSize++) {
145             a[pos * (n + 1) + newSize] = new IntPair(pos + 1, newSize);
146         }
147     }

```

```

    }
149     return a;
    }

151
    static Integer[] range(int n) {
153         assert n >= 0;
        Integer[] a = new Integer[n];
155         for (int i = 0; i < n; i++) {
            a[i] = i;
157         }
        return a;
159     }

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

165     static void memoBackref(Set<List<Integer>> s, IntPair p) {
        for (List<Integer> l : s) {
167             backref.put(l, p);
        }
169     }

171     static Set<List<Integer>> k(int n, int a) {
        // Dynamische Programmierung: ggf. schon vorhandenes Ergebnis zurueckgeben
173         IntPair key = new IntPair(n, a);
        if (memo.containsKey(key)) {
175             return memo.get(key);
        }
177         if (a == 0) {
            Set<List<Integer>> result = new HashSet<>();
179             result.add(Arrays.asList(range(n)));
            memo.put(key, result);
            memoBackref(result, key);
181             return result;
        }
183         if (n == 1 && a != 0) {
            Set<List<Integer>> result = new HashSet<>();
185             memo.put(key, result);
            memoBackref(result, key);
187             return result;
        }
189         HashSet<List<Integer>> result = new HashSet<>();
        for (IntPair rFlip : allRevFlipOps(n-1)) {
191
            for (List<Integer> seqL : k(n - 1, a - 1)) {
                Integer[] seq = seqL.toArray(new Integer[0]);
195                 Integer[] rFlipped = revFlipOp(seq, rFlip.first(), rFlip.second());
                if (!(a > 1 || !Arrays.equals(rFlipped, range(n)))) {
197                     continue;
                }
199                 boolean forall = true;
                for (Integer flip : allFlipOps(n)) {
201                     if (backref.containsKey(Arrays.asList(flipOp(rFlipped, flip)))) {
                        IntPair p = backref.get(Arrays.asList(flipOp(rFlipped, flip)));
203                         if (p.second() < a - 1) {
                            forall = false;
205                             break;
                        } else {
207                             continue;
                        }
                    }
209                 }
                boolean exists = false;
211                 for (int b = a - 1; b < 2 * Math.floor(n / 3) + 2; b++) {
                    if (k(n - 1, b).contains(Arrays.asList(flipOp(rFlipped, flip)))) {
213                         exists = true;
                        break;
215                     }
                }
                forall = forall && exists;
217                 if (!forall) {
                    break;
219                 }
            }
        }
    }

```

```

221         }
222         if (forall) {
223             result.add(Arrays.asList(rFlipped));
224         }
225     }
226 }
227 memo.put(key, result);
228 memoBackref(result, key);
229 return result;
230 }
231
232 // Gibt nur ein Element aus k(n, a) zurueck, falls es eins gibt
233 static Optional<Integer[]> kHasSolution(int n, int a) {
234     if (a == 0) {
235         return Optional.of(range(n));
236     }
237     if (n == 1 && a != 0) {
238         return Optional.empty();
239     }
240     for (IntPair rFlip : allRevFlipOps(n-1)) {
241         for (List<Integer> seqL : k(n - 1, a - 1)) {
242             Integer[] seq = seqL.toArray(new Integer[0]);
243             Integer[] rFlipped = revFlipOp(seq, rFlip.first(), rFlip.second());
244             if (!(a > 1 || !Arrays.equals(rFlipped, range(n))))
245                 continue;
246
247             // forall steht fuer den Existenzquantor ueber P_n
248             // exists steht fuer den Allquantor ueber N zwischen a-1 und 2*floor(n/3)+1
249             boolean forall = true;
250             boolean exists = false;
251             for (Integer flip : allFlipOps(n)) {
252                 if (backref.containsKey(Arrays.asList(flipOp(rFlipped, flip)))) {
253                     IntPair p = backref.get(Arrays.asList(flipOp(rFlipped, flip)));
254                     if (p.second() < a - 1) {
255                         forall = false;
256                         break;
257                     } else {
258                         continue;
259                     }
260                 }
261             }
262             exists = false;
263             for (int b = a - 1; b < 2 * Math.floor(n / 3) + 2; b++) {
264                 if (k(n - 1, b).contains(Arrays.asList(flipOp(rFlipped, flip)))) {
265                     exists = true;
266                     break;
267                 }
268             }
269             forall = forall && exists;
270             if (!forall)
271                 break;
272         }
273         if (forall)
274             return Optional.of(rFlipped);
275     }
276     return Optional.empty();
277 }
278
279 public static void main(String[] args) {
280     Scanner scanner = new Scanner(System.in);
281     System.out.print("n: ");
282     int n = scanner.nextInt();
283     scanner.close();
284     long startTime = System.currentTimeMillis();
285     for (int a = (int) Math.floor(n / 3) * 2 + 1; a > 0; a--) {
286         Optional<Integer[]> result = kHasSolution(n, a);
287         if (result.isPresent()) {
288             System.out.println("max: " + a);
289             for (int i = 0; i < result.get().length; i++) {
290                 System.out.print((result.get()[i] + 1) + " ");
291             }
292             System.out.println();
293             System.out.println("time: " + (System.currentTimeMillis() - startTime) + " ms");

```

```
295         }  
296         }  
297     }  
298 }  
299 }
```

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.