Aufgabe 1: LaTeX-Dokument

Teilnahme-ID: ?????

Bearbeiter/-in dieser Aufgabe:
Vor- und Nachname

4. Februar 2023

Inhaltsverzeichnis

Anleitung: Trage oben in den Zeilen 8 bis 10 die Aufgabennummer, die Teilnahme-ID und die/den Bearbeiterin/Bearbeiter dieser Aufgabe mit Vor- und Nachnamen ein. Vergiss nicht, auch den Aufgabennamen anzupassen (statt „LaTeX-Dokument“)!

Dann kannst du dieses Dokument mit deiner LaTeX-Umgebung übersetzen.

Die Texte, die hier bereits stehen, geben ein paar Hinweise zur Einsendung. Du solltest sie aber in deiner Einsendung wieder entfernen!

1 Lösungsidee

1.1 Notation

Die Menge der möglichen Stapel der Höhe n wird mit P_n bezeichnet. Die Möglichen Pfannkuchen-Wende-und-Ess-Operationen für einem Stapel mit n Pfannkuchen wird mit W_n bezeichnet. Die Menge der möglichen Umkehroperationen für solch einen Stapel wird mit W_n^{-1} bezeichnet. Wenn der Stapel $S \in \mathsf{P}_n$ durch die Operation $w \in \mathsf{W}_n$ verändert wird, so wird der neue Stapel $S' = wS$ bezeichnet. Operationen assoziieren nach rechts, d.h. $w_1 w_2 S = w_1(w_2 S)$. Die Funktionen A und P werden aus der Aufgabenstellung übernommen.

1.2 Sortieren

Die möglichen Stapel können in einem gerichteten Graphen dargestellt werden. Die Knoten des Graphen sind die Stapel, die Kanten sind die Operationen. Die Operationen sind gerichtet, denn eine PWUE-Operation kann nicht umgekehrt 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 verwenden. //TODO: Referenz zu Algorithms 4th Edition

procedure Dijkstra's Algorithmus(Stapel S)
  Intialisiere Prioritätswarteschlange Q
  Initialisiere Map V
  Initialisiere Map C

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

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

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

Eine Heuristik für das Pfannkuchensortieren ist die Anzahl der Adjazenzen. //TODO referenz zu gates und papadimitrou 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. In einer Operation können sich höchstens zwei neue Adjazenzen bilden. Weil in einer Operation sich nur die Nachbarn von zwei Pfannkuchen ändern, (nämlich des obersten, der nach unten gewendet wird, und dessen, der direkt unter dem Pfannenwender liegen bleibt) kann sich nur zwischen diesen beiden eine Adjazenz bilden. Eine weitere Adjazenz lässt sich dadurch bilden, dass der aufgegessene Pfannkuchen

die Breite zwischen zwei nebeneinanderliegenden hatte, welche nach der Operation keine Pfannkuchen mit Größe zwischen ihnen haben. Als Ajdazenz wird auch gezählt, wenn der größte Pfannkuchen ganz unten liegt. Ein sortierter Stapel der Höhe n hat n Adjazenzen. Seien a_0 die Anzahl der Adjazenzen im untersuchten Stapel, h die Höhe des Stapels und n die Anzahl der Sortieroperationen. Weiterhin seien a_f und h_f die Anzahl der Adjazenzen und die Höhe des sortierten Stapels. Dann gilt:

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

II. und III. in I. einsetzen:

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

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

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

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

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

1.3 PWUE-Zahl

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

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

$K(n, a)$ enthält also alle Stapel, die durch eine Umkehroperation aus Stapeln der Höhe $n-1$ mit mindestens $a-1$ Sortieroperationen entstehen können und für die keine andere Sortieroperation eine Stapel bildet, der in weniger als $a-1$ Schritten sortiert werden kann. Nach dieser Definition würde $K(n, 1)$ allerdings auch die komplett sortierten Stapel enthalten, weshalb noch die Bedingung $(a > 1) \vee (s \notin K(n, 0))$ ergänzt werden muss. Die Funktion $K(n, a)$ ist also definiert als

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

Dass diese Definition richtig ist, lässt sich überprüfen durch die Substitution $A(S) = k, S \in \mathsf{P}_n \iff (\forall w \in \mathsf{W}_n : A(ws) \geq k-1) \wedge (\exists w \in \mathsf{W}_n : A(ws) = k-1)$. 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. Da jeder Stapel $S \in \mathsf{P}_n$ in $\lceil \frac{n}{1.5} \rceil$ 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. Zuletzt muss noch ein Ende der Rekursion eingeführt werden, wir setzen $K(n, 0) = \{(1, \ldots, n)\}$, denn ein sortierter Stapel kann in 0 Schritten sortiert werden.

2 Umsetzung

Hier wird kurz erläutert, wie die Lösungsidee im Programm tatsächlich umgesetzt wurde. Hier können auch Implementierungsdetails erwähnt werden.

3 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!

4 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):
    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():
        _, _, _, node = queue.get()
        if target_pred(node):
            return reconstruct_path(node, prev)
        for adj_node in adj_func(node):
            new_cost = cost[node] - heur_func(node) + cost_func(node, adj_node) + heur_func(adj_node)
            if adj_node not in cost or new_cost < cost[adj_node]:
                i -= 1
                cost[adj_node] = new_cost
                queue.put((new_cost, heur_func(node), i, adj_node))
                prev[adj_node] = node

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

a_star.py

```python
import math
from 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.
def normalize(arr):
    order = sorted(arr)
    return tuple(order.index(x) for x in arr)


```

```python
# 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):
    return a_star(normalize(arr), is_sorted, next_arrs, lambda a, b: 1, heuristic)


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


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

```
22              return num1;
        }
24
        public int second() {
26              return num2;
        }
28
        @Override
30      public boolean equals(Object o) {
            if (this == o)
32              return true;
            if (o == null || getClass() != o.getClass())
34              return false;
            IntPair pair = (IntPair) o;
36          return (num1 == pair.num1 && num2 == pair.num2);
        }
38
        @Override
40      public int hashCode() {
            int hash = 17;
42          hash = hash * 31 + num1;
            hash = hash * 31 + num2;
44          return hash;
        }
46
        @Override
48      public String toString() {
            return "(" + num1 + ",␣" + num2 + ")";
50      }

52      @Override
        public int compareTo(Pwue.IntPair o) {
54          if (num1 < o.num1) {
                return -1;
56          }
            if (num1 > o.num1) {
58              return 1;
            }
60          if (num2 < o.num2) {
                return -1;
62          }
            if (num2 > o.num2) {
64              return 1;
            }
66          return 0;
        }
68
        public void setFirst(int num1) {
70          this.num1 = num1;
        }
72
        public void setSecond(int num2) {
74          this.num2 = num2;
        }
76
        public void swap() {
78          int temp = num1;
            num1 = num2;
80          num2 = temp;
        }
82  }

84  static Integer[] flipOp(Integer[] a, int i) {
        Integer[] b = new Integer[a.length - 1];
86      for (int j = 0; j < i - 1; j++) {
            b[j] = a[i - j - 2];
88      }
        for (int j = i; j < a.length; j++) {
90          b[j - 1] = a[j];
        }
92      return normalize(b);
    }
94
```

```
    static Integer[] revFlipOp(Integer[] a, int i, int n) {
96      i--;
        Integer[] b = new Integer[a.length + 1];
98      for (int j = 0; j < i; j++) {
            if (a[i - j - 1] >= n) {
100             b[j] = a[i - j - 1] + 1;
            } else {
102             b[j] = a[i - j - 1];
            }
104     }
        b[i] = n;
106     for (int j = i; j < a.length; j++) {
            if (a[j] >= n) {
108             b[j + 1] = a[j] + 1;
            } else {
110             b[j + 1] = a[j];
            }
112     }
        return normalize(b);
114 }

116 static Integer[] normalize(Integer[] a) {
        IntPair[] b = new IntPair[a.length];
118     for (int i = 0; i < a.length; i++) {
            b[i] = new IntPair(a[i], i);
120     }
        Arrays.sort(b);
122     for (int i = 0; i < a.length; i++) {
            b[i].setFirst(i);
124         b[i].swap();
        }
126     Arrays.sort(b);
        Integer[] c = new Integer[a.length];
128     for (int i = 0; i < a.length; i++) {
            c[i] = b[i].second();
130     }
        return c;
132 }

134 static Integer[] allFlipOps(int n) {
        Integer[] a = new Integer[n];
136     for (int i = 0; i < n; i++) {
            a[i] = i + 1;
138     }
        return a;
140 }

142 static IntPair[] allRevFlipOps(int n) {
        IntPair[] a = new IntPair[n * (n + 1)];
144     for (int i = 0; i < n; i++) {
            for (int j = 0; j <= n; j++) {
146             a[i * (n + 1) + j] = new IntPair(i + 1, j);
            }
148     }
        return a;
150 }

152 static Integer[] range(int n) {
        Integer[] a = new Integer[n];
154     for (int i = 0; i < n; i++) {
            a[i] = i;
156     }
        return a;
158 }

160 static Map<IntPair, Set<List<Integer>>> memo = new HashMap<>();

162

    static Set<List<Integer>> k(int n, int a, int depth) {
164     IntPair key = new IntPair(n, a);
        if (memo.containsKey(key)) {
166         return memo.get(key);
        }
```

```
168         if (a == 0) {
            Set<List<Integer>> result = new HashSet<>();
170         result.add(Arrays.asList(range(n)));
            memo.put(key, result);
172         return result;
        }
174     System.out.println("| ".repeat(depth) + "| k(" + n + ", " + a + ") "
                + ((int) Math.ceil(n / 1.5) - a + 1) * n * (n + 1)
176                     * k(n - 1, a - 1, depth + 1).size()
                + " Its");
178     HashSet<List<Integer>> result = new HashSet<>();
        for (IntPair rFlip : allRevFlipOps(n)) {
180
            for (List<Integer> seqL : k(n - 1, a - 1, depth + 1)) {
182             Integer[] seq = seqL.toArray(new Integer[0]);
                Integer[] rFlipped = revFlipOp(seq, rFlip.first(), rFlip.second());
184             if (!(a > 1 || !Arrays.equals(rFlipped, range(n)))) {
                    continue;
186             }
                boolean r1 = true;
188             boolean r2 = false;
                for (Integer flip : allFlipOps(n)) {
190                 for (int b = a - 1; b < Math.ceil(n / 1.5); b++) {
                        r2 = false;
192                     if (k(n - 1, b, depth + 1)
                                .contains(Arrays.asList(flipOp(rFlipped, flip)))) {
194                         r2 = true;
                            break;
196                     }
                    }
198                 r1 = r1 && r2;
                    if (!r1) {
200                     break;
                    }
202             }
                if (r1) {
204                 result.add(Arrays.asList(rFlipped));
                }
206         }
        }
208     System.out.println("| ".repeat(depth) + "| k(" + n + ", " + a + ") Fertig         ");
        memo.put(key, result);
210     return result;
    }
212
    static Optional<Integer[]> kHasSolution(int n, int a) {
214     for (IntPair rFlip : allRevFlipOps(n)) {
            for (List<Integer> seqL : k(n - 1, a - 1, 0)) {
216             Integer[] seq = seqL.toArray(new Integer[0]);
                Integer[] rFlipped = revFlipOp(seq, rFlip.first(), rFlip.second());
218             if (!(a > 1 || !Arrays.equals(rFlipped, range(n)))) {
                    continue;
220             }

222             boolean r1 = true;
                boolean r2 = false;
224             for (Integer flip : allFlipOps(n)) {
                    for (int b = a - 1; b < Math.ceil(n / 1.5); b++) {
226                     r2 = false;
                        if (k(n - 1, b, 0).contains(Arrays.asList(flipOp(rFlipped, flip)))) {
228                         r2 = true;
                            break;
230                     }
                    }
232                 r1 = r1 && r2;
                    if (!r1) {
234                     break;
                    }
236             }
                if (r1) {
238                 return Optional.of(rFlipped);
                }
240         }
```

```
            }
242         return Optional.empty();
        }
244
        public static void main(String[] args) {
246         System.out.println(Arrays.deepToString(allRevFlipOps(5)));
            Scanner scanner = new Scanner(System.in);
248         System.out.print("n: ");
            int n = scanner.nextInt();
250         scanner.close();
            for (int a = (int) Math.ceil(n / 1.5); a > 0; a--) {
252             Optional<Integer[]> result = kHasSolution(n, a);
                if (result.isPresent()) {
254                 System.out.println("max a: " + a);
                    System.out.println(Arrays.deepToString(result.get()));
256                 break;
                }
258         }

260     }
    }
```

Pwue.java