Aufgabe 3: Pancake Sort

Teilnahme-ID: 67739

Bearbeiter/-in dieser Aufgabe:
Robert Vetter

17. April 2023

Inhaltsverzeichnis

1 Lösungsidee

Um die beschriebene Erweiterung des Pancake-Sort-Problems zu lösen, habe ich mir zuerst die verschiedenen Schritte überlegt, die ich mit dem Pfannenwender ausführen kann. Jeder Schritt besteht aus dem Wenden eines Teilstapels und dem anschließendem Essen des obersten Pfannkuchens. Mein Ziel ist es, eine möglichst kurze Abfolge von Schritten zu finden, die den Stapel in einer von unten aufsteigenden Reihenfolge sortiert. Ein Ansatz zur Lösung dieses Problems besteht darin, einen Suchalgorithmus zu verwenden, der systematisch verschiedene Abfolgen von Schritten ausprobiert und diejenige auswählt, die den Stapel am schnellsten sortiert. Eine Möglichkeit, dies effizient zu tun, ist den A*-Suchalgorithmus zu benutzen, der eine Heuristik verwendet, um vielversprechende Abfolgen von Schritten zu priorisieren. Als Heuristik wählte ich die Anzahl der aufeinanderfolgenden Pfannkuchen, die in absteigender Reihenfolge angeordnet sind. Dieses vereinfacht das Problem erheblich, da man so schon vor der eigentlichen Berechnung die Anzahl der Schritte abschätzen kann, die erforderlich sind, um den Stapel zu sortieren. Hierbei benötigt jeder dieser Paare mindestens einen weiteren Schritt, um in die richtige Reihenfolge gebracht zu werden. Wenn die Heuristik für einen bestimmten Teilstapel genau Null ist, kann man daraus schlussfolgern, dass der Stapel bereits sortiert ist. In dem nachfolgenden Beispiel wäre dabei das Ergebnis der Heuristik zwei, da es zwei Paare von adjazenten Pancakes mit aufsteigender Größe gibt.

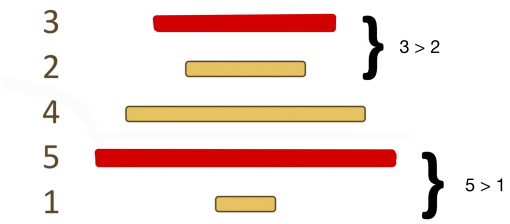

Abbildung 1: Veranschaulichung der Heuristik

In dem schon angesprochenenen A*-Algorithmus wird der gegebene Stapel von Pfannkuchen in seiner ursprünglichen Reihenfolge verwendet. Alle möglichen Nachfolgerstapel, die sich aus dem Wenden eines Teilstapels ergeben, werden generiert. Dabei wird für jede mögliche Position des Pfannenwenders der Stapel an dieser Position gewendet, um neue Konfigurationen zu erhalten. Für jeden neu entstandenen Nachfolgerstapel wird die Heuristik berechnet, die eine Schätzung der noch erforderlichen Schritte zur Sortierung des Stapels darstellt. Zudem habe ich eine kombinierte Bewertungsfunktion erstellt, die sowohl die Heuristik als auch die bisherigen Schritte verwendet, um den vielversprechendsten Stapel zur weiteren Untersuchung auszuwählen. Die Auswahl wird in einer Prioritätswarteschlange gespeichert, die die Stapel nach ihrer Bewertungsfunktion sortiert. Dabei wird der Prozess kontinuierlich wiederholt, indem der am besten bewertete Stapel aus der Prioritätswarteschlange entfernt wird, alle seine möglichen Nachfolger generiert und sie ebenfalls in die Warteschlange eingefügt werden. Sobald auf einen sortierten Stapel gestoßen wird, der alle Pfannkuchen in aufsteigender Reihenfolge enthält, hat der Algorithmus sein Ziel erreicht. In der folgenden Abbildung habe ich einen Ausgangsstapel an Pancakes in der Mitte abgebildet. Der A*-Algorithmus geht jetzt also in Schritt 1 jede mögliche Wende-Und-Ess-Operation durch und nimmt eine Bewertung anhand der Heuristik vor. Dabei fällt auf, dass für einen Stapel die Heuristik Null ist, was bedeutet, dass er schon vollständig sortiert ist. Somit lässt sich der Ausgangsstapel in einem Schritt sortieren:

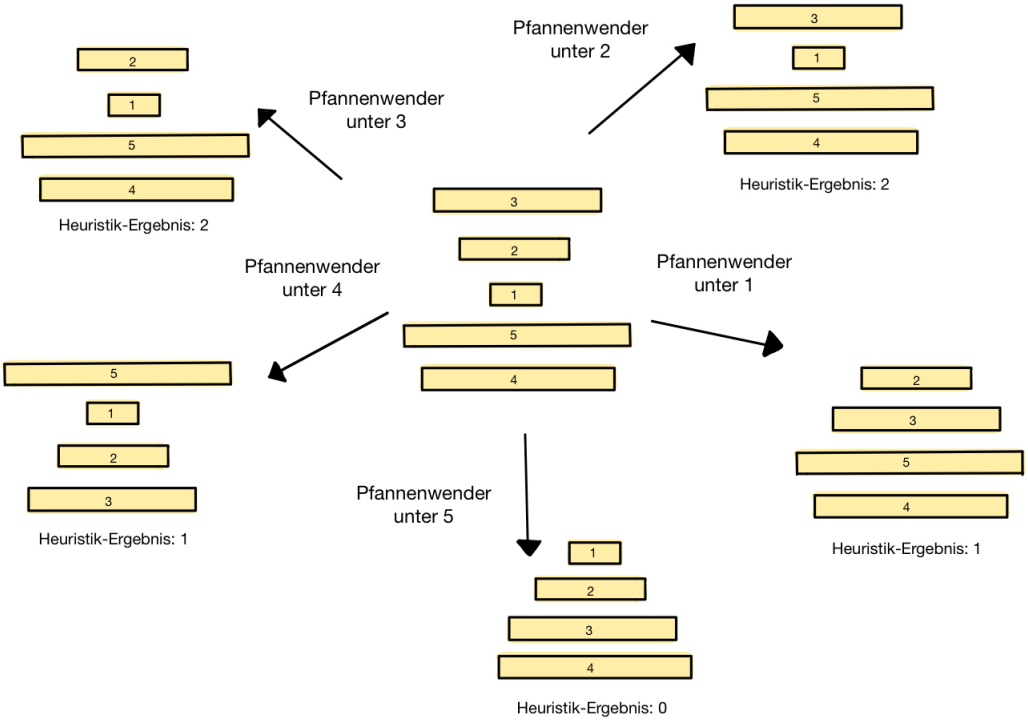

Abbildung 2: Veranschaulichung der Funktionsweise des A*-Algorithmus

In Teilaufgabe b) ist gefordert, für verschiedene Stapelgrößen die maximale PWUE-Zahl zu berechnen, also die größtmögliche Anzahl an möglichst wenig Wende-Und-Ess-Operationen. Dabei müssen lediglich alle möglichen Permutationen der jeweiligen Stapelgrößen berechnet werden und auf alle der A*-Algorithmus angewendet werden. Da es n! mögliche Permutationen gibt und die Laufzeit für große n somit sehr hoch werden kann, hatte ich die Idee, Multithreading, eine Datenbank als auch die Erkennung von Symmterien und Gemeinsamkeiten zu implementieren.

2 Umsetzung

Die Umsetzung der Lösungsidee in Programmcode basiert auf der Implementierung des A*-Algorithmus zur Lösung des Pfannkuchen-Problems. Dabei wählte ich die Programmiersprache Python. Anfänglich erstellte ich eine Methode ”read_pancakes_from_file”, welche lediglich die Größen der Pfannkuchen aus der TXT-Datei ausliest. Wie bereits ausführlich in dem vorherigen Kapitel beschrieben, erarbeitete ich zuerst eine Methode ”heuristic” mit einem Stapel als Parameter. Sie berechnet die Anzahl aufeinanderfolgender aufsteigender Paare, um eine Abschätzung der verbleibenden Schritte zu erhalten. Um die Nachfolgerstapel zu berechnen, schrieb ich eine Methode ”successors”. Diese nimmt als Parameter einen Ausgangsstapel entgegen und generiert alle möglichen Nachfolgerstapel, die durch das Wenden eines Teilstapels entstehen. Dies ermöglicht es dem Algorithmus, alle möglichen Schritte zu erkunden, um den optimalen Pfad zu finden. Nun werde ich mich dem Kernstück meines Programmes widmen, dem A*-Algorithmus. Er basiert auf der Idee, einen Pfad vom Startknoten zum Zielknoten zu finden, indem er die Kosten der bereits zurückgelegten Strecke (g-Score) und die geschätzten Kosten bis zum Zielknoten (h-Score) berücksichtigt. Der Algorithmus kombiniert die Vorteile der günstigsten Kostensuche (Dijkstra) und der heuristischen Suche (Greedy Best-First Search). Um den A*-Algorithmus effizient umzusetzen, wird eine Heap-Datenstruktur verwendet. Heaps ermöglichen es, schnell den Knoten mit dem niedrigsten f-Score (Summe aus g- und h-Score) zu finden und zu entfernen. Der Algorithmus untersucht dann die Nachfolger des Knotens und aktualisiert deren g- und f-Scores entsprechend. Insgesamt bietet der A*-Algorithmus somit eine effiziente und zielgerichtete Suche nach der optimalen Lösung für das Pfannkuchenproblem, indem er den Suchraum systematisch durchläuft und dabei sowohl die bisherigen Kosten als auch die geschätzten zukünftigen Kosten berücksichtigt. Nachfolgend habe ich ihn nochmal in einem Struktogramm dargestellt:

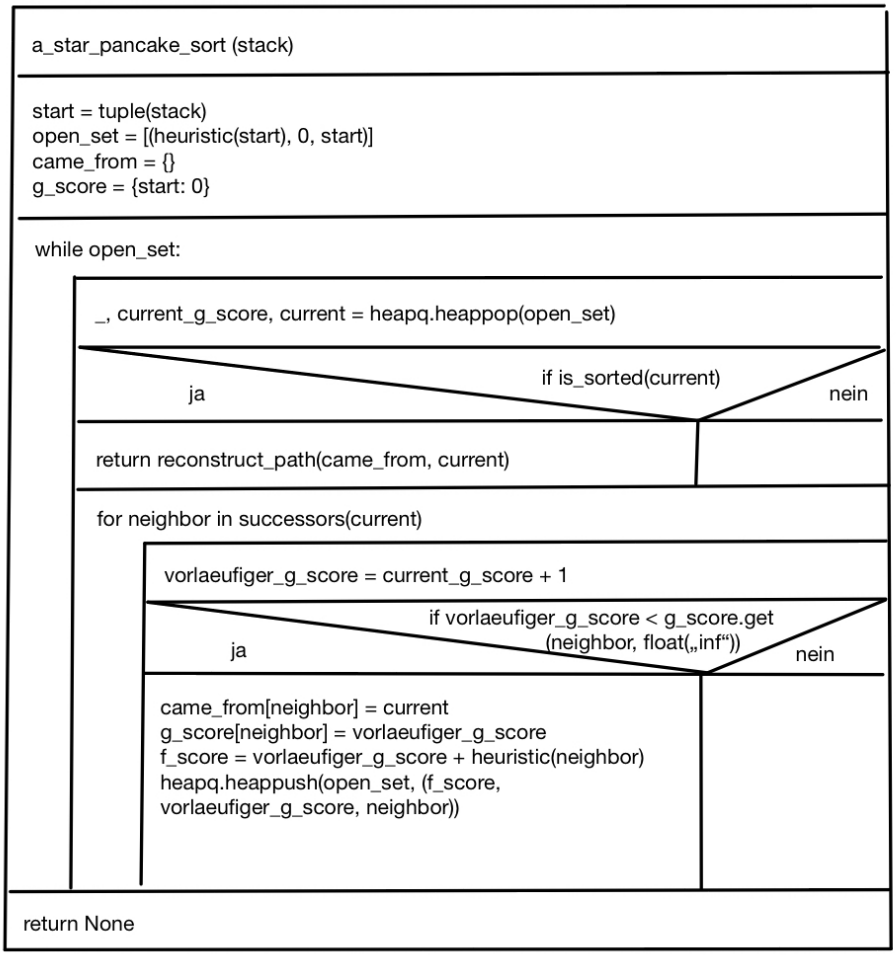

Abbildung 3: Struktogramm des A*-Algorithmus

Der Algorithmus ruft die Methode ”reconstruct_path”, welche den gefundenen Pfad vom Anfangszustand bis zum Zielzustand rückwärts rekonstruiert. Sie nimmt zwei Parameter entgegen: das Dictionary came_from, welches die Vorgänger für jeden untersuchten Stapel enthält, und den aktuellen sortierten Stapel. Die Methode geht durch das came_from-Dictionary und fügt den aktuellen Stapel dem Pfad hinzu, bis sie den Startzustand erreicht. Anschließend wird der Pfad umgekehrt, um die richtige Reihenfolge der Schritte von Anfang bis Ende zu erhalten. Die Funktion ”print_steps” verarbeitet hingegen den rekonstruierten Pfad und gibt die einzelnen Schritte in einer verständlichen und informativen Form aus. Sie nimmt den sortierten Pfad als Eingabe und erstellt eine Liste von Schritten, die den Index des umzudrehenden Teilstapels und den resultierenden Stapel nach dem Umkehren und Essen des obersten Pfannkuchens enthalten. Die Liste der Schritte wird zurückgegeben und kann zur Ausgabe oder weiteren Analyse verwendet werden. Somit wäre Teilaufgabe a) gelöst. Für Teilaufgabe b), bei der die Pancake-Wende-Und-Ess (PWUE)-Zahlen für n Pfannkuchen berechnet werden sollen, wurde die bereits entwickelte A*-Implementierung weiter optimiert, um die Berechnungen effizienter und schneller durchzuführen. Wie bereits in meiner Lösungsidee beschrieben, habe ich in einer Methode ”generate_permutations” alle möglichen Permutationen für eine Anzahl n an Pfannkuchen generieren lassen. Hierbei ist jedoch zu beachten, dass die Anzahl der zu überprüfenden Permutationen mit n! steigt. So hätte man bei 10 Pfannkuchen schon

$$N_{Permutationen}(10) = 10! = 3628800 \tag{1}$$

mögliche Permutationen. Ich fand jedoch keinen Weg, wie ich bestimmte Permutationen schon im Vorhinein ausschließen kann. Also versuchte ich, meinen Algorithmus über verschiedene Ansätze zu optimieren.

1. Multithreading: Um die Berechnungszeit zu reduzieren, wurde Multithreading verwendet, um mehrere Stapel gleichzeitig zu bearbeiten. Mit ProcessPoolExecutor aus dem concurrent.futures-Modul wurden parallele Prozesse erstellt, die jeweils die A*-Berechnungen für verschiedene Stapel durchführen. Dies führt zu einer erheblichen Zeitersparnis, insbesondere bei größeren n-Werten.

2. Memoisierung: Die Memoisierungstechnik wurde angewendet, um bereits berechnete Ergebnisse für bestimmte Stapel zu speichern und später wiederverwenden zu können, anstatt die gleichen Berechnungen erneut durchzuführen. Mit Hilfe der "lru_cache"'-Funktion aus dem functools-Modul wurde die "memoized_a_star_pancake_sort"-Funktion erstellt, die die Ergebnisse der A*-Berechnungen zwischenspeichert.

In der "find_pwue_number"-Funktion wurden alle Optimierungen implementiert. Zuerst werden alle möglichen Stapel generiert. Anschließend werden mithilfe des Multithreadings paralelllaufende Prozesse realisiert und die memoisierte A*-Funktion verwendet, um die Anzahl der Wende-Und-Ess-Operationen zu berechnen. Schließlich wird die maximale Anzahl der erforderlichen Operationen und ein Beispielstapel für die gegebene n-Größe gefunden und zurückgegeben. Weiterhin kann man noch einige Erweiterungen hinzufügen:

1. Schätzung der PWUE-Zahl
   Angenommen, P(5) ist bekannt, dann kann P(6) geschätzt werden. In der Menge der Stapel mit Höhe 5 gibt es mindestens einen Stapel, bei dem A(S) = P(5) zutrifft. Darüber hinaus gibt es einige Stapel, bei denen A(S) = P(5)-1 ist. Mit jeder durchgeführten Operation verringert sich die Stapelhöhe um 1. In den Varianten der Stapel mit Höhe 6 werden einige Stapel die Eigenschaft A(S) = P(5) aufweisen, und möglicherweise gibt es Varianten, bei denen A(S) = P(5)+1 zutrifft. P(6) ist daher entweder gleich P(5) oder P(5)+1. Wenn bei der Berechnung der Permutationen ein Stapel gefunden wird, bei dem A(S) = P(n-1)+1 zutrifft, muss nicht weiter nach einem anderen Stapel gesucht werden, der die insgesamt höchste A(S)-Nummer besitzt.

2. Verwendung einer Datenbank
   Das Analyse von kleineren Stapel mit A(S) = P(n) führt zu einem geringeren Aufwand bei der Sortierung im Vergleich zu hohen Stapelhöhen, da für letztere eine besonders hohe Anzahl an Berechnungen notwendig ist, um alle möglichen Sortierwege zu ermitteln. Falls jedoch die vorherigen P(n)-Werte bekannt sind und eine Datenbank vorhanden ist, die die Anzahl der Operationen für jeden Stapel verzeichnet, könnte auf diese zurückgegriffen werden, um die Laufzeit erheblich zu senken. Somit müssten lediglich alle resultierenden Stapel betrachtet werden, die bei der ersten Wende-Und-Ess-Operation entstehen könnten. Weiterhin müssten ggf. noch die Größen der Pancakes auf die der in der Datenbank vorhandenen Stapel abgestimmt werden, da der größte Pfannkuchen lediglich die Größe n-1 besitzt. Dieses habe ich nachfolgend exemplarisch abgebildet:

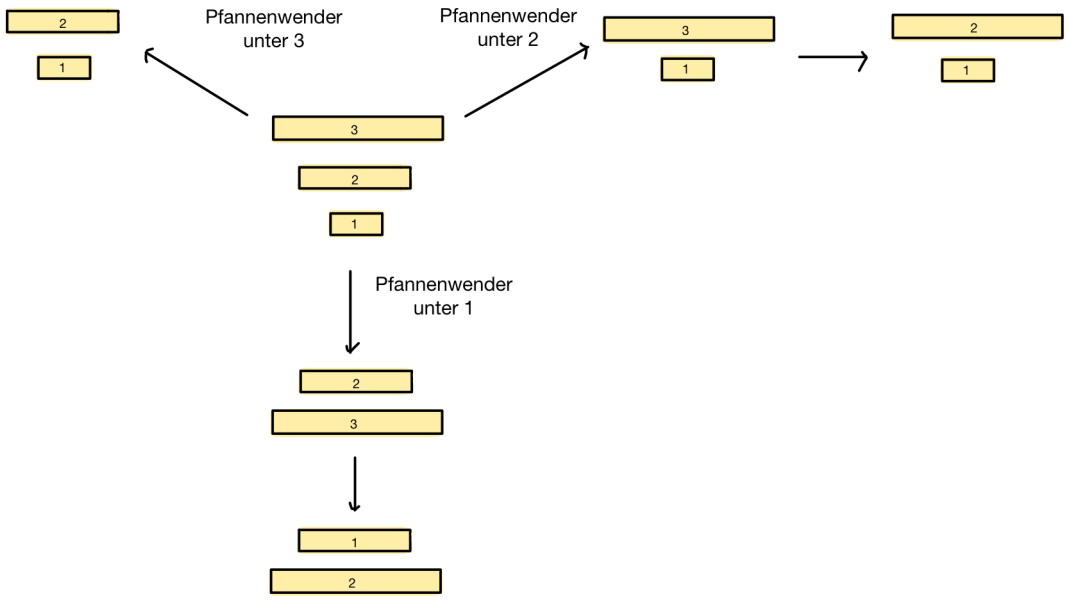

Abbildung 4: Veranschaulichung des Anpassungsprozesses in der Datenbank

Angenommen, der Ausgangsstapel ist absteigend sortiert mit den Größen 3, 2 und 1. Schiebt man den Pfannenwender unter die Position 3 und führt die Wende-Und-Ess-Operation aus, erhält man den Stapel 2, 1. Für diesen Stapel könnte man jetzt die schnellste Lösung aus der Datenbank herauslesen, vorausgesetzt, man hat alle möglichen Operationen bei n=2 schonmal betrachtet. Würde man den Pfannenwender nun unter den Pancake mit der Größe 2 oder 1 schieben, würde in beiden resultierenden Stapeln die Größe 3 vorkommen, welche so noch nicht in der Datenbank enthalten ist. Aus diesem Grund erweist es sich als sinnvoll, diesen in eine Pancake-Größe umzuformen, die schon einmal betrachtet wurde, in diesem Fall die 2. Nachdem man also die Lösung dieses Stapels aus der Datenbank herausgelesen hat, könnte man am Ende die Substitution wieder rückgängig machen und das Ergebnis weiterverwenden. Letztlich habe ich jedoch nur Erweiterung 1 beibehalten, da die Laufzeit des Programmes durch das Suchen in einer Datenbank und die Substitution eher gestiegen anstatt gesunken ist. Dabei verwendete ich für die Datenbank das Python-Modul sqlite3. Da mein Programm außerdem mit der Suche aufhört, sobald es eine Lösung gefunden hat, die um 1 größer als die PWUE-Zahl des vorherigen n ist, würde sowieso nur ein Teil der möglichen Wende-Und-Ess-Operationen in der Datenbank gespeichert werden. Zuletzt habe ich eine main-Methode erarbeitet, welche über einfache Fallunterscheidungen den Benutzer fragt, ob er

1. Die minimale Anzahl an Wende-Und-Ess-Operationen für ein eigenes Beispiel berechnen lassen will bzw. automatisiert ein eigenes Beispiel generieren möchte,
2. die geringste Anzahl an Wende-Und-Ess-Operationen für ein BWInf-Beispiel berechnen lassen möchte oder
3. die PWUE-Zahl für ein bestimmtes n ermitteln will.

3 Beispiele

Zuerst habe ich alle gegebenen BWInf-Beispiele getestet.

Ursprünglicher Stapel	Umformungsschritte	Kurzform der Schritte	Programmlaufzeit
[3, 2, 4, 5, 1]	2	[(5, [5, 4, 2, 3]), (4, [2, 4, 5])]	0,0 Sekunden

Die Kurzform der Schritte meint dabei die benötigten Umformungsschritte. So wurde der Pfannenwender im Ausgangsstapel zuerst unter dem fünften Pancake eingeschoben, danach alle darüberliegenden Pfannkuchen gedreht und der Oberste gegessen, wonach anschließend der Pfannenwender unter den Pancake mit der Position 4 (also der unterste Pancake mit der Größe 3) geschoben wurde und die Wende-Und-Ess-Operation noch einmal ausgeführt wurde. Nachfolgend habe ich alle weiteren Beispiele des BWInf abgebildet:

Ursprünglicher Stapel	Umformungsschritte	Kurzform der Schritte	Programmlaufzeit
[6, 3, 1, 7, 4, 2, 5]	3	[(6, [4, 7, 1, 3, 6, 5]), (5, [3, 1, 7, 4, 5]), (3, [1, 3, 4, 5])]	0,0 Sekunden
[8, 1, 7, 5, 3, 6, 4, 2]	4	[(8, [4, 6, 3, 5, 7, 1, 8]), (6, [7, 5, 3, 6, 4, 8]),... (1, [3, 5, 7, 8])]	0,0 Sekunden
[5, 10, 1, 11, 4, 8, 2, 9, 7, 3, 6]	6	[(10, [7, 9, 2, 8, 4, 11, 1, 10, 5, 6]), ...(2, [1, 4, 5, 7, 9])]	0,01 Sekunden
[7, 4, 11, 5, 10, 6, 1, 13, 12, 9, 3, 8, 2]	7	[(12, [3, 9, 12, 13, 1, 6, 10, 5, 11, 4, 7, 2]),..., (2, [4, 5, 6, 9, 12, 13])]	0,03 Sekunden
[4, 13, 10, 8, 2, 3, 7, 9, 14, 1, 12, 6, 5, 11]	7	[(10, [14, 9, 7, 3, 2, 8, 10, 13, 4, 12, 6, 5, 11]),...,(1, [2, 3, 4, 5, 6, 9, 14])]	0,07 Sekunden
[14, 8, 4, 12, 13, 2, 1, 15, 7, 11, 3, 9, 5, 10, 6]	8	[(15, [10, 5, 9, 3, 11, 7, 15, 1, 2, 13, 12, 4, 8, 14]),...,(1, [1, 2, 3, 4, 5, 10, 14])]	0,14 Sekunden
[8, 5, 10, 15, 3, 7, 13, 6, 2, 4, 12, 9, 1, 14, 16, 11]	8	..., (1, [2, 4, 7, 8, 9, 10, 14, 16])]	0,93 Sekunden

Es fällt auf, dass alle vorgegebenen Beispiele unter einer Sekunde gelöst werden können. Natürlich ist aufgrund der Wahl der Heuristik nicht garantiert, dass wirklich die optimale Lösung gefunden wird. Es ist jedoch dennoch ein sehr guter Weg, um in sehr kurzer Zeit eine passende Lösung zu entwickeln. Nachfolgend habe ich noch einige eigene Beispiele dargestellt:

Ursprünglicher Stapel	Länge des Stapels	Umformungsschritte	Programmlaufzeit
[3, 8, 18, 13, 6, 17, 2, 14, 11, 9, 1, 12, 7, 5, 10, 4, 15, 16]	18	9	1,3 Sekunden
[7, 6, 14, 13, 19, 15, 17, 5, 11, 3, 16, 4, 9, 1, 12, 8, 2, 10, 18]	19	10	7,57 Sekunden
[3, 19, 15, 8, 5, 7, 11, 1, 4, 17, 2, 13, 6, 9, 20, 10, 12, 16, 14, 18]	20	10	6,28 Sekunden
[5, 4, 2, 20, 15, 18, 13, 7, 14, 12, 16, 1, 3, 11, 8, 17, 9, 6, 10, 21, 19]	21	11	43,46 Sekunden

Nun möchte ich mich noch den PWUE-Zahlen widmen. Da mein Algorithmus nicht alle Möglichkeiten ausprobiert, sondern lediglich basierend auf der Heuristik Entscheidungen trifft, stimmt die PWUE-Zahl für n=6 nicht mit der vorgegebenen PWUE-Zahl überein. Nachfolgend habe ich die Ergebnisse der Berechnung abgebildet:

Anzahl n an Pancakes	Berechnete PWUE-Zahl	Beispiel	Programmlaufzeit
1	0	[1]	0,01 Sekunden
2	1	[2, 1]	0,12 Sekunden
3	2	[2, 3, 1]	0,67 Sekunden
4	2	[1, 2, 4, 3]	1,16 Sekunden
5	3	[1, 4, 2, 5, 3]	1,45 Sekunden
6	4	[3, 5, 1, 6, 4, 2]	1,87 Sekunden
7	4	[1, 2, 3, 4, 7, 6, 5]	5,23 Sekunden
8	5	[1, 2, 7, 8, 6, 5, 4, 3]	11,34 Sekunden
9	5	[1, 2, 3, 4, 5, 8, 6, 9, 7]	57,01 Sekunden
10	6	[1, 2, 3, 4, 7, 10, 9, 8, 6, 5]	7,08 Minuten
11	6	[2, 4, 3, 5, 9, 6, 10, 7, 11, 8, 1]	56,5 Minuten
12	7	[2, 4, 10, 3, 7, 11, 6, 9, 5, 12, 8, 1]	ca. 2 Stunden
13	7	[2, 4, 3, 5, 8, 12, 7, 10, 6, 11, 9, 13, 1]	ca. 11 Stunden

Es ist klar erkennbar, dass bei hohem n die Laufzeit des Algorithmus stark zunimmt. Dieses ist auf die steigende Anzahl der Permutationen als auch auf die wachsende Länge des Stapels zurückzuführen. Jetzt wo man jedoch schon einmal die PWUE-Zahlen hat, könnte man das erste Beispiel zurückgeben, dass der Algorithmus findet, bei welchem PWUE(n) Operationen benötigt werden. Dies würde wesentlich weniger Zeit in Anspruch nehmen.

4 Quellcode

Da es in dem vorgegebenen LaTeX-Dokument so angegeben war, habe ich meinen gesamten Quellcode nachfolgend eingefügt:

```
import os
import heapq
import itertools
import time
from concurrent.futures import ProcessPoolExecutor
from functools import lru_cache
import random

def read_pancakes_from_file(filename):
    with open(filename, "r") as file:
        num_pancakes = int(file.readline())
        pancakes = [int(line.strip()) for line in file.readlines()]
    return pancakes

# Funktion zum Ueberpruefen, ob der Stapel sortiert ist. Die Funktion gibt True zurueck, wenn der Stapel sortiert ist, andernfalls False.
def is_sorted(stack):
    return all(stack[i] <= stack[i + 1] for i in range(len(stack) - 1))

# Heuristik, die die Anzahl der ungeordneten Paare in einem Stapel berechnet. Die Heuristik gibt eine Schaetzung der verbleibenden Schritte zurueck, die benoetigt werden, um den Stapel zu sortieren.
def heuristic(stack):
    return sum(1 for i in range(len(stack) - 1) if stack[i] > stack[i+1])

# Funktion zum Generieren von Nachfolgern (moeglichen Schritten) fuer einen gegebenen Stapel. Die Funktion gibt eine Sequenz von Stapeln zurueck, die durch Anwenden einer Wendeoperation auf den gegebenen Stapel erzeugt wurden.
def successors(stack):
    for i in range(1, len(stack) + 1):
        flipped_stack = stack[:i][::-1][1:] + stack[i:]
        yield flipped_stack

# Funktion zur Rekonstruktion des Pfades von A*-Suche. Die Funktion gibt eine Liste von Stapeln zurueck, die den Pfad von der Startposition zur Zielsortierung repraesentiert.
def reconstruct_path(came_from, current):
    path = [current]
```

```
    while current in came_from:
        current = came_from[current]
        path.append(current)
    return path[::-1]


# A*-Suche zur Sortierung von Pancake-Stapeln
def a_star_pancake_sort(stack):
    start = tuple(stack)

    open_set = [(heuristic(start), 0, start)]
    came_from = {}
    g_score = {start: 0}

    while open_set:
        _, current_g_score, current = heapq.heappop(open_set)

        if is_sorted(current):
            return reconstruct_path(came_from, current)

        for neighbor in successors(current):
            vorlaeufiger_g_score = current_g_score + 1

            if vorlaeufiger_g_score < g_score.get(neighbor, float('inf')):
                came_from[neighbor] = current
                g_score[neighbor] = vorlaeufiger_g_score
                f_score = vorlaeufiger_g_score + heuristic(neighbor)
                heapq.heappush(open_set, (f_score, vorlaeufiger_g_score, neighbor))

    return None

# Funktion zum Erstellen einer Liste von Schritten und resultierenden Stapeln aus einem geordneten Pfad
def print_steps(sorted_path):
    steps = []
    for i in range(len(sorted_path) - 1):
        prev_state, next_state = sorted_path[i], sorted_path[i + 1]
        flipped_index = next(i for i in range(len(prev_state)) if prev_state[:i+1][::-1][1:] + prev_sta
        steps.append((flipped_index, list(next_state)))
    return steps

# Funktion zum Generieren aller moeglichen Permutationen eines Stapels der Groesse n.
def generate_permutations(n):
    return itertools.permutations(range(1, n + 1))

# Memoisierte A* Pancake-Sortierfunktion. Die Funktion verwendet das Python-Dekorator
@lru_cache, um Berechnungen zu speichern und die Leistung zu verbessern.
@lru_cache(maxsize=None)
def memoized_a_star_pancake_sort(stack):
    return len(a_star_pancake_sort(stack)) - 1

# Funktion zum Finden der PWUE-Nummer fuer n Pfannkuchen
def find_pwue_number(n, previous_pwue_number=None):
    max_operations = -1
    max_example = None

    stacks = list(itertools.permutations(range(1, n + 1)))

    with ProcessPoolExecutor() as executor:
        futures = [executor.submit(memoized_a_star_pancake_sort, stack) for stack in stacks]

        for i, future in enumerate(futures):
            steps = future.result()
            if steps > max_operations or max_example is None:
                max_operations = steps
                max_example = stacks[i]

            if previous_pwue_number is not None and max_operations >= previous_pwue_number + 1:
                break

    return max_operations, max_example

# Funktion zum Berechnen der PWUE-Zahlen fuer Pfannkuchenstapel von der Groesse 'start' bis zur
Groesse 'end'
```

```
111 def calculate_pwue_numbers(start, end):
    pwue_numbers = {}
113     previous_pwue_number = None
    for n in range(start, end + 1):
115         pwue_number, example = find_pwue_number(n, previous_pwue_number)
        pwue_numbers[n] = (pwue_number, example)
117         print(f"Kurzform: P({n}) = {pwue_number}, Beispiel: {example}")
        previous_pwue_number = pwue_number
119
    return pwue_number
121
  def generate_random_stack(n):
123     pancake_sizes = list(range(1, n+1))
    random.shuffle(pancake_sizes)
125     return pancake_sizes

127 def main():
    pancake_list = []
129     decision = int(input("Moechten Sie die minimale Anzahl an Wende-Und-Ess-Operationen fuer einen
    Stapel ausgeben (1) oder stattdessen die PWUE-Zahl berechnen?(2) "))
131     if decision == 1:
        bwinf_or_own = int (input("Moechten Sie fuer ein eigenes Beispiel die Anzahl an Wende-Und-Ess-
133         Operationen berechnen (1) oder fuer ein BWInf-Beispiel? (2) "))
        if bwinf_or_own == 1:
135             random_or_own = int (input("Moechten Sie zufaellig einen Stapel generieren lassen (1)
            oder stattdessen einen Eigenen erstellen? (2) "))
137             if random_or_own == 1:
                numbers = int(input("Wie viele Pfannkuchen soll ihr Stapel besitzen? "))
139                 pancake_list = generate_random_stack(numbers)
            else:
141                 numbers = int(input("Wie viele Pfannkuchen soll ihr Stapel besitzen? "))
                for i in range(1, numbers + 1):
143                     number = int(input(str("Geben Sie Zahl " + str(i) + " ein: ")))
                    pancake_list.append(number)
145         if bwinf_or_own == 2:

147             file_found = False
            # solange kein gueltiger Dateiname eingegeben wurde
149             while not file_found:
                file_name = str(input("Geben Sie die Bezeichnung der txt-Datei mit den Pancakes an
151                 (Bsp.: pancake0.txt). \nDie Datei muss ich dafuer in demselben Verzeichnis wie dieses
                    Skript
153                 befinden.: "))
                # Datei im aktuellen Verzeichnis finden
155                 current_directory = os.path.dirname(os.path.abspath(__file__))
                file_path = os.path.join(current_directory, file_name)
157
                # Ueberpruefen, ob die Datei existiert
159                 if os.path.isfile(file_path):
                    pancake_list = read_pancakes_from_file(file_path)
161                     file_found = True
                else:
163                     print("Datei nicht gefunden. Bitte geben Sie einen gueltigen Dateinamen ein.")
        print("Urspruenglicher Stapel: ", pancake_list)
165         start = time.time()
        sorted_path = a_star_pancake_sort(pancake_list)
167         if sorted_path:
            steps = print_steps(sorted_path)
169             print("Schritte:", len(steps))
            print("Kurzform: ", steps)
171             for i, (flip_index, resulting_stack) in enumerate(steps, start=1):
                print(f"Schritt {i}: Der Pfannenwender wird unter den {flip_index}ten Pfannkuchen
173                 geschoben, umgedreht und der oberste gegessen. Resultierender Stapel: {resulting_stack}
        else:
175             print("Keine Loesung gefunden")
    else:
177         pwue_num_start = int(input("Geben Sie die Anzahl n an Pfannkuchen an, ab der Sie die PWUE-Zahl
        pwue_num_end = int(input("Geben Sie die Anzahl n an Pfannkuchen an, bis zu welcher Sie die PWUE
179         start = time.time()
        calculate_pwue_numbers(pwue_num_start, pwue_num_end)
181
    end = time.time()
183     duration = end - start
```

```
    print("Die Programmlaufzeit betraegt ", round(duration, 2), " Sekunden")

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