Aufgabe 2: Pancake Sort

Teilnahme-ID: 64903

Bearbeiter/-in dieser Aufgabe:
Alessio Caputo

8. April 2023

Inhaltsverzeichnis

Lösungsidee

Teilaufgabe A

Das gegebene Problem ist eine Erweiterung des “pancake sort” Problems. Bei diesem müssen eine Reihe an unterschiedlichen Pfannkuchen der Größe nach sortiert werden. Ziel ist es dabei, wie bei der Aufgabenstellung zu A3, die kürzeste Anzahl an Wendeloperationen zu erzielen. Dabei gibt es anders als bei der gegebenen Problemstellung jedoch keine „Essoperation“.

Eine mögliche Lösung des Problems ist ein iterativer Bruteforce-Algorythmus, der alle möglichen Wendeoperationen durchführt. Dieser würde immer ein Optimales Ergebnis berechnen. Dabei wird Stufenweise jede mögliche Wendeoperationenreinfolge getestet.(zuerst alle der ersten Wendeoperation, danach alle der zweiten Wendeoperation, …). Zu beachten ist, dass dieser Algorithmus auch viele verschiedene Lösungen mit der selben PUWE Zahl berechnen kann. Er bedarf jedoch eine Menge an Rechenkapazität und Speicherkapazität um bei größeren Eingaben eine Lösung innerhalb kürzerer Zeiten zu finden.

Werden jedoch die Lösungen betrachtet, die der Algorithmus ausgibt, so zeigt sich eine Art Muster in den verschiedenen Lösungswegen. In jeder Wendeoperation kommt es zu einer Verlängerung der direkt aufeinanderfolgenden Pancakes. Auch wird deshalb

zwischen zwei bereits ideal nebeneinanderliegenden Pancakes keine Wendeoperation durchgeführt. Ein solcher Schritt würde kontraproduktive Auswirkungen haben.

Dieses Verhalten kann genutzt werden, um einen neuen Algorithmus auf Basis des Bruteforce Algorithmus zu entwerfen. In dieser Lösung wird jeder Pfannkuchen-stapel auf Basis der auf- und absteigenden Zahlenfolgen bewertet, um einen Wert zu erhalten, der Aufschluss darüber gibt, wie nahe dieser an einer Potenziellen Lösung ist. Wenn nun in jeder Etappe lediglich die besten Bewertungen verfolgt werden, kann das die Geschwindigkeit und vor allem den Speicherbedarf deutlich verbessern.

Teilaufgabe B

Hier muss für ein gegebenes n eine Zahlenfolge generiert werden, die die längstmögliche Anzahl an Wendeoperationen zu folge hat. Wenn diese mit einem fehlerfreien Algorithmus gelöst wird, muss eine Optimale Lösung für diese Zahlenfolge gefunden werden. Mit den Wissen über die Zahlenfolgen aus Teilaufgabe A kann die folgende Prozedur genutzt werden, um die Zahlenfolge zu generieren, deren PUWE Zahl am Höchsten ist.

Zunächst wird die aufsteigende Zahlenfolge von 1 bis n in Zwei Hälften geteilt. Sollte n Ungerade sein, wird der hinteren Hälfte die mittlere Zahl zugeschlagen.

1 2 3 4 5 6 7 8 9

1 2 3 4 - 5 6 7 8 9

Anschließend werden diese Zahlenfolgen nach dem Reisverschlussprinzip wie Folgt angeordnet: $b_1; a_1; b_2; a_2; b_3; a_3 \ldots$

1 2 3 4
5 6 7 8 9

Wichtig ist dabei, dass mit der zweiten Liste begonnen wird, da keine 1 vorne stehen darf. Wenn sie vorne stünde, würde die PUWE-Zahl um 1 zu klein sein. Der entstehende Pfannkuchenstapel muss nun lediglich durch den Bruteforce-Algorythmus gelöst werden, um die Gefrage PUWE-Zahl und den Lösungsweg zu berechnen.

5 1 6 2 7 3 8 4 9

Da nun ein „schwerer Pfannkuchenstapel“ vorliegt, kann nun dieser gelöst werden. Eine PUWE Zahl $P(n)$ muss immer die Bedingung $P(n) \geq P(n+1)$ erfüllen. Außerdem kann eine PUWE-Zahl maximal 2-Mal auftreten. Also muss auch gelten $P(n) \neq P(n+2)$.

Dieses wissen kann genutzt werden, um eine untere Schranke für n mit $P(n-1)$ zu definieren. Dadurch müssen alle PUWE-Zahlen gleichzeitig berechnet werden, oder bei der Eingabe die vorherige PUWE-Zahl festgelegt werden.

Nun wird für ein gegebenes n lediglich der Baum rekursiv bis zur Tiefe $P(n-1)$ gescannt. Solle in dieser Tiefe keine Lösung gefunden werden, muss die Suchtiefe $P(n-1)+1$ gewählt werden. Indem durch den rekursiven Aufruf nicht der gesamte Baum, sondern nur der tatsächlich interessante Teil des Baumes gescannt wird, kann Laufzeit und Speicher eingespart werden.

NP-schwere

Das TSP-Problem ist bekanntermaßen NP-schwer. Um eine Instanz des TSP-Problems in eine Instanz des Pfannkuchenproblems zu reduzieren, kann für jede Stadt eine Größe des Pfannkuchens definiert werden. Wenn eine Reihenfolge gefunden wird, in der der Verkäufer jede Stadt genau einmal besucht und zum Ausgangspunkt zurückkehrt, entspricht dies einer Sortierung des Stapels der Pfannkuchen in aufsteigender Größe. Umgekehrt, wenn eine Sortierung des Stapels der Pfannkuchen in aufsteigender Größe gefunden wird, können die Größen der Pfannkuchen als Reihenfolge der besuchten Städte interpretiert werden und geprüft werden, ob der Verkäufer jede Stadt genau einmal besucht und zum Ausgangspunkt zurückkehrt. Da das TSP-Problem NP-schwer ist und eine Reduktion auf das Pfannkuchenproblem durchgeführt werden kann, ist das Pfannkuchenproblem ebenfalls NP-schwer.

Implementierung Teilaufgabe A

Die beschienene Lösungsidee wird in die Programmiersprache Python implementiert. Es wurde sich für Python entschieden, da die Dynamischen listen der Sprache eine ideale Basis für das Speichern von Pfannkuchenstapeln bilden.

Pancake Klasse

Für das Speichern von Pfannkuchenstapeln wird die Klasse Pancake (im Code nur „P“) erstellt, die es ermöglicht, alle Informationen über den Stapel zu ordnen. Diese Klasse beinhaltet die Funktionen „flip“ um eine Wendeoperation durchzuführen, „isValid“, um zu überprüfen, ob dieser Stapel korrekt ist, „copy“ um eine Kopie von sich selbst zu erstellen, „getFlipIndexPoints“ um alle Punkte zu erhalten an denen „flip“ aufgerufen werden kann und „printHistory“ um die Wendeoperationen in der Konsole anzuzeigen. Der eigentliche Stapel ist in Form einer Liste von Integern „stack“ gespeichert. Um die eigene Historie zu Speichern wird nicht jeder Zustand von „stack“ gespeichert, sondern lediglich die Indices der Wendeoperationen. Mit diesen kann der Weg, den der Ausgangsstapel verfolgt hatte rekonstruiert und ausgegeben werden. Diese Herangehensweise ist deutlich Ressourceneffizienter als das Speichern aller bisherigen Zustände.

Bewertung der Stapel

Wie in der Lösungsidee beschrieben, müssen auf- und absteigende Zahlenfolgen innerhalb der Liste gefunden werden. Das Problem ist dabei, dass die auf- und absteigenden Zahlenfolgen innerhalb der Liste gesucht werden, und nicht direkt aufeinanderfolgende Zahlenfolgen. Es wird nur die Zahlenmenge der Liste betrachtet. Um diese Zahlenfolgen innerhalb der Zahlenmenge zu finden werden zunächst alle aufsteigenden Zahlenfolgen in der Liste gesucht, indem für jedes Element (mit ausnahme des Letzten) überprüft wird, ob $a_i > a_{i+1}$. Wenn die Überprüfungen in folge richtig waren, wird diese Sequenz in einer Liste „secuences” gespeichert. Anschließend wird durch diese Liste iteriert und für jedes Element die größte und die kleinste Zahl ermittelt. Da mögliche Sequenzen Lücken in den Zahlenreihen haben können werden alle möglichen Zahlen ermittelt, die innerhalb

der Sequenz in die Lücken passen. Es wird nun überprüft, ob eine dieser Zahlen im Rest der ursprünglichen Zahlenmenge ist. Wenn dies nicht der Fall ist, dann wird die Sequenz am ende zurückgegeben. Wenn Eine der Lücken der Sequenz jedoch in dem Rest der ursprünglichen Zahlenmenge ist, dann wird mit einem rekursivem Aufruf der selben Funktion der Bereich vor und nach der Lücke überprüft. Dies ist notwendig, da eventuell noch weitere Sequenzen in dieser „Ursequenz“ existieren können (Bsp. [1, 2, 3, 5, 6, 7, 8, 4]). Ebenfalls werden alle Zahlen, nicht nur die der Untersequenz, an die rekursive Funktion weitergegeben damit die Sequenzlücken auf jede Zahl überprüft werden können. Durch das Rekursive verhalten, kann die gesamte Liste in Unterlisten aufgeteilt werden und mit den oben stehenden Bedingungen auf Zahlenfolgen innerhalb der Liste überprüft werden. Nun werden lediglich die Listen mit der Länge 1 aus den entstandenen Listen gefiltert, da diese nicht tatsächlich aufsteigend sind. Um die absteigenden Sequenzen zu ermitteln wird die Funktion für die aufsteigen Sequenzen aufgerufen mit invertierten Eingaben aufgerufen. Auch werden die Ergebnisse zusätzlich Invertiert. Dieses vorgehen ist nicht Ideal bzw. effizient, da das Invertieren Rechenzeit beansprucht. (Ich habe mich dennoch dafür entschieden, da in dieser Funktion zu viele Operatoren zu verändern wären ... :D)

Da nun die Sequenzen errechnet wurden muss nur noch aus diesen Sequenzen ein „Rating“ ermittelt werden. Dazu wurden verschiedene Methoden getestet, die besten Ergebnisse wurden mit dieser Methode erzielt:

Zunächst werden die Summen der Aufsteigenden und Absteigen Listen miteinander Addiert. Anschließend wird die Größere der beiden Zahlen zurückgegeben. Je größer die Zahl ist, desto „besser“ ist der Stapel. Es wird bewusst nur einer der beiden Zustände betrachtet, da nahe einer Lösung nur auf-, bzw. absteigende Zahlenfolgen zu finden sind.

(letztendlich werden die invertierten Ausgaben der Funktion, die die absteigenden Sequenzen Sequenzen berechnet also nicht genutzt :D Dies ist hauptsächlich ein Relikt aus anderen Bewertungsalgorithmen)

main()

Die „main“-Funktion steuert alle Berechnungen des Algorithmus. (ach ne…) Zunächst wird die Datei eingelesen und in den Konstruktor der „Pancake“ Klasse überführt. Dabei wird die erste Zeile der Datei nicht eingelesen, da sie lediglich die Länge der Liste beschreibt. Für die Programmiersprache Python ist diese Information jedoch unerheblich, da das Listenobjekt seine Größe dynamisch anpasst.

Um die in der Lösungsidee beschriebene etappenweise Implementierung des Algorithmus umzusetzen werden zwei Listen erstellt. Durch die erste Liste wird Iteriert, und die zweite Liste beinhaltet alle mutierten Pfannkuchenstapel. Diese wird später in die erste Liste kopiert, um das nächste Layer zu scanen. Während der Iterierung durch die erste Liste werden für jedes Element der Liste alle möglichen Wendeoperationen durchgeführt. Nach jeder Wendeoperation wird überprüft, ob die entstandene Liste valide ist. Wenn sie dies nicht ist, wird sie von dem Bewertungsalgorithmus aufgegriffen und Bewertet. Die besten n (n=Parameter von main()) werden der zweiten Liste hinzugefügt. Nach allen Iterationen durch die erste Liste liegt die zweite Liste mit

einigen Mutationen der vorangegangenen Generation vor. Von dieser werden die besten m (m=Parameter von main()) in die erste Liste kopiert. Die zweite Liste wird gelöscht.

n / m – Parameter

Durch Veränderung der Parameter n und m der Main-Funktion kann die Laufzeit und die Genauigkeit des Algorithmus modifiziert werden. Je kleiner die Beiden Zahlen desto geringer die Laufzeit und Genauigkeit. Das verändern des m Parameters beeinflusst vor allem auch den Speicherbedarf enorm, da er entscheidet, wie viele Pfannkuchenstapel in der nächsten Iteration überprüft werden. Relativ große n / m Parameter können ebenfalls dafür sorgen, dass auf kleineren Eingaben eine vollständige Bruteforce-Suche angewendet wird. Diese Implementierung wurde gewählt, da sie nach ausgiebigem Testen bessere Ergebnisse hervorgebracht hatte als der m-Parameter alleine.

Beim Ausführen des Scripts kann zwischen verschiedenen Presets von n/m- Parametern gewählt werden, die für verschiedene Eingaben optimiert sind. (Geschwindigkeit; Ausgeglichen; Genauigkeit; „Super-Genauigkeit“)

Implementierung Teilaufgabe B

Für die Berechnung der PUWE Zahlen wird wie vorher in der Lösungsidee beschrieben ein Optimaler Algorithmus gewählt. Anders als für Teilaufgabe A wird dieser Code in C++ geschrieben und mit -Ofast kompiliert, um möglichst hohe Geschwindigkeiten zu Gewährleiten.

Die Logik der Pancake bzw. P – Klasse wird in den C++ Code übertragen.

Es wird eine Klasse *Calc* definiert, die die Lösung für einen Stapel berechnet. In dieser befindet sich die Funktion *main*(), die den rekursiven Aufruf der Funktion *find*() mit den richtigen Parametern startet.

Die *find*()-Funktion überprüft zunächst, ob die Anzahl der Wendeoperationen mit der gewünschten PUWE-Zahl übereinstimmt. Wenn ja, dann wird der Konstruktionsweg ausgegeben. Wenn nein, dann wird für jede weitere mögliche Mutation ein rekursiver Aufruf getätigt.

Beispiele - Teilaufgabe A

In den hier aufgeführten Beispielen muss zwischen den m/n Presets der Ausgaben unterschieden werden. Auch kommt gut hervor, wie die unterschiedlichen Parameter Einfluss auf die Ausgaben nehmen.

pancake0.txt

Fast; Normal	**Accurate**
`USING SCAN PRESET: fast` `-- init_len:5; flip_operations:3 --`	`USING SCAN PRESET: accurate` `-- init_len:5; flip_operations:2 --`

```
-- : [3, 2, 4, 5, 1]
0  : [2, 4, 5, 1]
2  : [4, 2, 1]
2  : [2, 4]
PUWE: 3

time: 0.001s

(„normal“: 0.001s)
```

```
-- : [3, 2, 4, 5, 1]
4  : [5, 4, 2, 3]
3  : [2, 4, 5]
PUWE: 2

time: 0.001s
```

pancake1.txt

Fast; Normal; Accurate

```
USING SCAN PRESET: fast
-- init_len:7; flip_operations:3 --

-- : [6, 3, 1, 7, 4, 2, 5]
2  : [3, 6, 7, 4, 2, 5]
3  : [7, 6, 3, 2, 5]
4  : [2, 3, 6, 7]
PUWE: 3

time: 0.002s

(“Normal”: 0.003s, “Accurate”: 0.005s)
```

pancake2.txt

Fast; Normal; Accurate

```
USING SCAN PRESET: fast
-- init_len:8; flip_operations:4 --

-- : [8, 1, 7, 5, 3, 6, 4, 2]
1  : [8, 7, 5, 3, 6, 4, 2]
3  : [5, 7, 8, 6, 4, 2]
3  : [8, 7, 5, 4, 2]
4  : [4, 5, 7, 8]
PUWE: 4

time: 0.005s

(“Normal”: 0.008s, “Accurate”: 0.023s)
```

pancake3.txt

Fast; Normal

```
USING SCAN PRESET: fast
-- init_len:11;
flip_operations:6 --

-- : [5, 10, 1, 11, 4, 8,
       2, 9, 7, 3, 6]
5  : [4, 11, 1, 10, 5, 2,
```

Accurate

```
USING SCAN PRESET: accurate
-- init_len:11;
flip_operations:6 --

-- : [5, 10, 1, 11, 4, 8, 2, 9,
7, 3, 6]
3  : [1, 10, 5, 4, 8, 2, 9, 7,
```

Superaccurate

```
USING SCAN PRESET: superaccurate
-- init_len:11; flip_operations:6 --

-- : [5, 10, 1, 11, 4, 8, 2, 9, 7, 3,
       6]
8  : [9, 2, 8, 4, 11, 1, 10, 5, 3, 6]
```

```
       9, 7, 3, 6]
0  : [11, 1, 10, 5, 2, 9,
       7, 3, 6]
1  : [11, 10, 5, 2, 9, 7,
       3, 6]
7  : [3, 7, 9, 2, 5, 10,
       11]
3  : [9, 7, 3, 5, 10, 11]
3  : [3, 7, 9, 10, 11]
PUWE: 6

time: 0.034s
```

```
       3, 6]
7  : [9, 2, 8, 4, 5, 10, 1, 3,
       6]
1  : [9, 8, 4, 5, 10, 1, 3, 6]
3  : [4, 8, 9, 10, 1, 3, 6]
4  : [10, 9, 8, 4, 3, 6]
5  : [3, 4, 8, 9, 10]
PUWE: 6

0.113s
```

```
9  : [3, 5, 10, 1, 11, 4, 8, 2, 9]
4  : [1, 10, 5, 3, 4, 8, 2, 9]
2  : [10, 1, 3, 4, 8, 2, 9]
5  : [8, 4, 3, 1, 10, 9]
4  : [1, 3, 4, 8, 9]
PUWE: 6

time: 3.819s
```

pancake4.txt

Fast

```
USING SCAN PRESET: fast
-- init_len:13; flip_operations:8 --

-- : [7, 4, 11, 5, 10, 6, 1, 13, 12, 9, 3, 8, 2]
10 : [9, 12, 13, 1, 6, 10, 5, 11, 4, 7, 8, 2]
9  : [4, 11, 5, 10, 6, 1, 13, 12, 9, 8, 2]
1  : [4, 5, 10, 6, 1, 13, 12, 9, 8, 2]
9  : [8, 9, 12, 13, 1, 6, 10, 5, 4]
4  : [13, 12, 9, 8, 6, 10, 5, 4]
5  : [6, 8, 9, 12, 13, 5, 4]
4  : [12, 9, 8, 6, 5, 4]
5  : [5, 6, 8, 9, 12]
PUWE: 8

time: 0.117s
```

Normal

```
USING SCAN PRESET: normal
-- init_len:13; flip_operations:7 --

-- : [7, 4, 11, 5, 10, 6, 1, 13, 12, 9, 3, 8, 2]
10 : [9, 12, 13, 1, 6, 10, 5, 11, 4, 7, 8, 2]
0  : [12, 13, 1, 6, 10, 5, 11, 4, 7, 8, 2]
10 : [8, 7, 4, 11, 5, 10, 6, 1, 13, 12]
2  : [7, 8, 11, 5, 10, 6, 1, 13, 12]
4  : [5, 11, 8, 7, 6, 1, 13, 12]
0  : [11, 8, 7, 6, 1, 13, 12]
5  : [1, 6, 7, 8, 11, 12]
PUWE: 7

time: 0.375s
```

Accurate

```
USING SCAN PRESET: accurate
-- init_len:13; flip_operations:7 --

-- : [7, 4, 11, 5, 10, 6, 1, 13, 12, 9, 3, 8, 2]
3  : [11, 4, 7, 10, 6, 1, 13, 12, 9, 3, 8, 2]
10 : [3, 9, 12, 13, 1, 6, 10, 7, 4, 11, 2]
9  : [4, 7, 10, 6, 1, 13, 12, 9, 3, 2]
7  : [12, 13, 1, 6, 10, 7, 4, 3, 2]
3  : [1, 13, 12, 10, 7, 4, 3, 2]
0  : [13, 12, 10, 7, 4, 3, 2]
6  : [3, 4, 7, 10, 12, 13]
PUWE: 7

time: 2.467s
```

pancake5.txt

Fast

```
USING SCAN PRESET: fast
-- init_len:14; flip_operations:7 --

-- : [4, 13, 10, 8, 2, 3, 7, 9, 14, 1, 12, 6, 5,
       11]
9  : [14, 9, 7, 3, 2, 8, 10, 13, 4, 12, 6, 5, 11]
```

Normal

```
USING SCAN PRESET: normal
-- init_len:14; flip_operations:7 --

-- : [4, 13, 10, 8, 2, 3, 7, 9, 14, 1, 12, 6, 5,
       11]
8  : [9, 7, 3, 2, 8, 10, 13, 4, 1, 12, 6, 5, 11]
```

```
12 : [5, 6, 12, 4, 13, 10, 8, 2, 3, 7, 9, 14]
2  : [6, 5, 4, 13, 10, 8, 2, 3, 7, 9, 14]
7  : [2, 8, 10, 13, 4, 5, 6, 7, 9, 14]
3  : [10, 8, 2, 4, 5, 6, 7, 9, 14]
0  : [8, 2, 4, 5, 6, 7, 9, 14]
0  : [2, 4, 5, 6, 7, 9, 14]
PUWE: 7

time: 0.091s
```

```
9  : [1, 4, 13, 10, 8, 2, 3, 7, 9, 6, 5, 11]
11 : [5, 6, 9, 7, 3, 2, 8, 10, 13, 4, 1]
3  : [9, 6, 5, 3, 2, 8, 10, 13, 4, 1]
5  : [2, 3, 5, 6, 9, 10, 13, 4, 1]
7  : [13, 10, 9, 6, 5, 3, 2, 1]
7  : [2, 3, 5, 6, 9, 10, 13]
PUWE: 7

time: 0.439s
```

Accurate

```
USING SCAN PRESET: accurate
-- init_len:14; flip_operations:7 --

-- : [4, 13, 10, 8, 2, 3, 7, 9, 14, 1, 12, 6, 5, 11]
8  : [9, 7, 3, 2, 8, 10, 13, 4, 1, 12, 6, 5, 11]
4  : [2, 3, 7, 9, 10, 13, 4, 1, 12, 6, 5, 11]
6  : [13, 10, 9, 7, 3, 2, 1, 12, 6, 5, 11]
8  : [12, 1, 2, 3, 7, 9, 10, 13, 5, 11]
8  : [13, 10, 9, 7, 3, 2, 1, 12, 11]
0  : [10, 9, 7, 3, 2, 1, 12, 11]
6  : [1, 2, 3, 7, 9, 10, 11]
PUWE: 7

time: 3.018s
```

pancake6.txt

Fast

```
USING SCAN PRESET: fast
-- init_len:15; flip_operations:9 --

-- : [14, 8, 4, 12, 13, 2, 1, 15, 7, 11, 3, 9,
       5, 10, 6]
8  : [15, 1, 2, 13, 12, 4, 8, 14, 11, 3, 9, 5,
       10, 6]
11 : [9, 3, 11, 14, 8, 4, 12, 13, 2, 1, 15, 10,
       6]
0  : [3, 11, 14, 8, 4, 12, 13, 2, 1, 15, 10, 6]
6  : [12, 4, 8, 14, 11, 3, 2, 1, 15, 10, 6]
0  : [4, 8, 14, 11, 3, 2, 1, 15, 10, 6]
3  : [14, 8, 4, 3, 2, 1, 15, 10, 6]
5  : [2, 3, 4, 8, 14, 15, 10, 6]
6  : [15, 14, 8, 4, 3, 2, 6]
6  : [2, 3, 4, 8, 14, 15]
PUWE: 9

time: 0.305s
```

Normal

```
USING SCAN PRESET: normal
-- init_len:15; flip_operations:9 --

-- : [14, 8, 4, 12, 13, 2, 1, 15, 7, 11, 3, 9, 5,
       10, 6]
12 : [9, 3, 11, 7, 15, 1, 2, 13, 12, 4, 8, 14, 10,
       6]
3  : [11, 3, 9, 15, 1, 2, 13, 12, 4, 8, 14, 10, 6]

5  : [1, 15, 9, 3, 11, 13, 12, 4, 8, 14, 10, 6]
2  : [15, 1, 3, 11, 13, 12, 4, 8, 14, 10, 6]
7  : [4, 12, 13, 11, 3, 1, 15, 14, 10, 6]
3  : [13, 12, 4, 3, 1, 15, 14, 10, 6]
7  : [14, 15, 1, 3, 4, 12, 13, 6]
7  : [13, 12, 4, 3, 1, 15, 14]
5  : [1, 3, 4, 12, 13, 14]
PUWE: 9

time: 2.097s
```

Accurate

```
USING SCAN PRESET: accurate
-- init_len:15; flip_operations:8 --

-- : [14, 8, 4, 12, 13, 2, 1, 15, 7, 11, 3, 9, 5, 10, 6]
8  : [15, 1, 2, 13, 12, 4, 8, 14, 11, 3, 9, 5, 10, 6]
6  : [4, 12, 13, 2, 1, 15, 14, 11, 3, 9, 5, 10, 6]
9  : [3, 11, 14, 15, 1, 2, 13, 12, 4, 5, 10, 6]
7  : [13, 2, 1, 15, 14, 11, 3, 4, 5, 10, 6]
```

```
0  : [2, 1, 15, 14, 11, 3, 4, 5, 10, 6]
4  : [14, 15, 1, 2, 3, 4, 5, 10, 6]
8  : [10, 5, 4, 3, 2, 1, 15, 14]
6  : [1, 2, 3, 4, 5, 10, 14]
PUWE: 8

time: 5.179s
```

pancake7.txt

Fast

```
USING SCAN PRESET: fast
-- init_len:16; flip_operations:10 --

-- : [8, 5, 10, 15, 3, 7, 13, 6, 2, 4, 12, 9, 1,
       14, 16, 11]
1  : [8, 10, 15, 3, 7, 13, 6, 2, 4, 12, 9, 1, 14,
       16, 11]
3  : [15, 10, 8, 7, 13, 6, 2, 4, 12, 9, 1, 14,
       16, 11]
11 : [1, 9, 12, 4, 2, 6, 13, 7, 8, 10, 15, 16,
       11]
3  : [12, 9, 1, 2, 6, 13, 7, 8, 10, 15, 16, 11]
4  : [2, 1, 9, 12, 13, 7, 8, 10, 15, 16, 11]
10 : [16, 15, 10, 8, 7, 13, 12, 9, 1, 2]
9  : [1, 9, 12, 13, 7, 8, 10, 15, 16]
1  : [1, 12, 13, 7, 8, 10, 15, 16]
2  : [12, 1, 7, 8, 10, 15, 16]
0  : [1, 7, 8, 10, 15, 16]
PUWE: 10

time: 0.447s
```

Normal

```
USING SCAN PRESET: normal
-- init_len:16; flip_operations:9 --

-- : [8, 5, 10, 15, 3, 7, 13, 6, 2, 4, 12, 9, 1,
       14, 16, 11]
1  : [8, 10, 15, 3, 7, 13, 6, 2, 4, 12, 9, 1, 14,
       16, 11]
3  : [15, 10, 8, 7, 13, 6, 2, 4, 12, 9, 1, 14,
       16, 11]
11 : [1, 9, 12, 4, 2, 6, 13, 7, 8, 10, 15, 16,
       11]
3  : [12, 9, 1, 2, 6, 13, 7, 8, 10, 15, 16, 11]
11 : [16, 15, 10, 8, 7, 13, 6, 2, 1, 9, 12]
5  : [7, 8, 10, 15, 16, 6, 2, 1, 9, 12]
4  : [15, 10, 8, 7, 6, 2, 1, 9, 12]
8  : [9, 1, 2, 6, 7, 8, 10, 15]
0  : [1, 2, 6, 7, 8, 10, 15]
PUWE: 9

time: 2.923s
```

Accurate

```
USING SCAN PRESET: accurate
-- init_len:16; flip_operations:8 --

-- : [8, 5, 10, 15, 3, 7, 13, 6, 2, 4, 12, 9, 1, 14, 16, 11]
1  : [8, 10, 15, 3, 7, 13, 6, 2, 4, 12, 9, 1, 14, 16, 11]
3  : [15, 10, 8, 7, 13, 6, 2, 4, 12, 9, 1, 14, 16, 11]
11 : [1, 9, 12, 4, 2, 6, 13, 7, 8, 10, 15, 16, 11]
3  : [12, 9, 1, 2, 6, 13, 7, 8, 10, 15, 16, 11]
5  : [6, 2, 1, 9, 12, 7, 8, 10, 15, 16, 11]
4  : [9, 1, 2, 6, 7, 8, 10, 15, 16, 11]
9  : [16, 15, 10, 8, 7, 6, 2, 1, 9]
8  : [1, 2, 6, 7, 8, 10, 15, 16]
PUWE: 8

time: 7.054s
```

Theoretische Analyse – Teilaufgabe A

Der Algorithmus bietet ohne zweifel eine Annäherung an die Optimalen Lösungen für die gestellten Beispieleingaben des BWINF. Dennoch handelt es sich nur um eine Annäherung an eine Optimale Lösung für jeden beliebigen Pfannkuchenstapel. Die Heuristik ist dennoch sehr gut, da auch auf größere Eingaben relativ gute Ergebnisse in recht kurzer Zeit gefunden werden.

Dass unter unterschiedlichen Presets unterschiedliche Ergebnisse mit der selben PUWE-Zahl gefunden werden zeigt, ebenfalls, dass die Lösung nicht optimal ist. Jedes Preset überprüft einen Großteil des ihm vorangegangenen Presets. Deshalb sollten idealerweise die selben Lösungswege ausgegeben werden. Da dies nicht der Fall ist, muss davon ausgegangen werden, dass es zu besseren Ergebnissen kommen kann, wenn nicht nur die Pfannkuchenstapel mit der höchsten Bewertung für eine Wendeoperation herangezogen werden.

Es fällt auf, dass die Lösungen des Programms mit dem “fast”-Preset vor allem an den kleineren Stapeln (< ~12) hervorragende Ergebnisse liefert. (mit zufällig generiertem Datensatz getestet) Auf den Sachverhalt bezogen ist dies ein recht gutes Ergebnis, da ich mir keinen Pfannkuchenstapel mit über 20 Pfannkuchen vorstellen kann (auch wenn ich gerne einen hätte! Auch wird durch die Ungenauigkeit des Programms erzwungen mehr Pfannkuchen als notwendig zu Essen). Diese “Genauigkeit” auf kleineren Eingaben ist vor allem auf die verwendete Heuristik zurückzuführen. Diese kann zwar auf größeren Eingaben ebenfalls zu guten Ergebnissen führen, jedoch werden die Wahrscheinlichkeiten dafür mit steigender Eingabegröße unwahrscheinlicher.

Die Laufzeit des Algorithmus hängt von verschiedenen Faktoren ab. Vereinfachter weise wird hier die variable n_{main} weggelassen und nur die Limitierung durch m_{main} berücksichtigt. Da für jede mögliche Iteration maximal m_{main} Pfannkuchenstapel berechnet werden kann dieser Prozess mit $O(m_{main}(n-1))$ dargestellt werden. Das Iterieren durch die verschiedenen Punkte der „getFlipIndexPoints“ dauert $O(n)$ und das Bewerten des Pfannkuchenstapel benötigt ebenfalls etwa $O(n)$ Zeit. Dadurch ergibt sich eine Gesamtlaufzeit von $O(m_{main}(n-1)n^2)$. (Keine Erfahrung mit der Big-O-Notation. Gut möglich, dass etwas falsch ist.)

Der Bruteforce Algorithmus zeigt auf kleineren Eingaben immer innerhalb von angemessener Zeit die korrekte Lösung für das Problem. Jedoch ist dieser mit einer Laufzeit von $O(2^n)$ vor allem für größere Eingaben ungeeignet. Selbst wenn er mit einer Heuristik ergänzt werden würde, die eine Wendeoperation innerhalb einer korrekten Sequenz des Pfannkuchens verhindern würde, wäre die Laufzeit immer noch nahezu Exponentiell.

Durch die Limitierung der Anzahl der Stapel in dem Programm kann der Speicheraufwand mit $m_{main}n$ dargestellt werden. Die einzig relevante Größe bilden die Länge der Pfannkuchenstapel und der Pfannkuchenstapel selbst, da diese durch die Variable m_{main} limitiert sind. Bei kleineren Eingaben ist der Speicheraufwand sogar noch geringer, da das Limit von m_{main} nicht erreicht wird. Mit pancake7.txt auf dem “accurate”-Preset wurde ein Maximum von etwa ~12Mib Speicherbelegung festgestellt.

Mit der Bruteforce-Lösung ist der Speicherbedarf deutlich höher als $m_{main} n$. Da für jeden Pfannkuchenstapel neue Pfannkuchen erstellt und gespeichert werden müssen. Hier wäre ein Rekursive Herangehensweise von Nöten, bei der nicht die erste Lösung die beste wäre. (Die Notationen für den Speicheraufwand beinhalten nur eine Grobe Schätzung. Variablen innerhalb der “Pancake”-Klasse werden beispielsweise nicht berücksichtigt.)

Beispiele – Teilaufgabe B

```
CALC:  P(8)
-- 8 4 7 3 6 2 5 1 --

-- init_len:8; flip_operations:4 --
-- : 8 4 7 3 6 2 5 1
1 : 8 7 3 6 2 5 1
6 : 5 2 6 3 7 8
0 : 2 6 3 7 8
1 : 2 3 7 8
PUWE: 4
PUWE(8)=4 in 0ms calculated

CALC:  P(10)
-- 10 5 9 4 8 3 7 2 6 1 --

-- init_len:10; flip_operations:5 --
-- : 10 5 9 4 8 3 7 2 6 1
1 : 10 9 4 8 3 7 2 6 1
8 : 6 2 7 3 8 4 9 10
3 : 7 2 6 8 4 9 10
4 : 8 6 2 7 9 10
3 : 2 6 8 9 10
PUWE: 5
PUWE(10)=5 in 1ms calculated

CALC:  P(12)
-- 12 6 11 5 10 4 9 3 8 2 7 1 --

-- init_len:12; flip_operations:6 --
-- : 12 6 11 5 10 4 9 3 8 2 7 1
1 : 12 11 5 10 4 9 3 8 2 7 1
10 : 7 2 8 3 9 4 10 5 11 12
3 : 8 2 7 9 4 10 5 11 12
6 : 10 4 9 7 2 8 11 12
1 : 10 9 7 2 8 11 12
4 : 2 7 9 10 11 12
PUWE: 6
PUWE(12)=6 in 22ms calculated

CALC:  P(14)
-- 14 7 13 6 12 5 11 4 10 3 9 2 8 1 --

-- init_len:14; flip_operations:8 --
-- : 14 7 13 6 12 5 11 4 10 3 9 2 8 1
0 : 7 13 6 12 5 11 4 10 3 9 2 8 1
0 : 13 6 12 5 11 4 10 3 9 2 8 1
1 : 13 12 5 11 4 10 3 9 2 8 1
10 : 8 2 9 3 10 4 11 5 12 13
3 : 9 2 8 10 4 11 5 12 13
```

```
CALC:  P(9)
-- 9 4 8 3 7 2 6 1 5 --

-- init_len:9; flip_operations:5 --
-- : 9 4 8 3 7 2 6 1 5
0 : 4 8 3 7 2 6 1 5
0 : 8 3 7 2 6 1 5
1 : 8 7 2 6 1 5
5 : 1 6 2 7 8
1 : 1 2 7 8
PUWE: 5
PUWE(9)=5 in 0ms calculated
CALC:  P(11)
-- 11 5 10 4 9 3 8 2 7 1 6 --

-- init_len:11; flip_operations:6 --
-- : 11 5 10 4 9 3 8 2 7 1 6
0 : 5 10 4 9 3 8 2 7 1 6
1 : 5 4 9 3 8 2 7 1 6
7 : 7 2 8 3 9 4 5 6
4 : 3 8 2 7 4 5 6
1 : 3 2 7 4 5 6
2 : 2 3 4 5 6
PUWE: 6
PUWE(11)=6 in 1ms calculated

CALC:  P(13)
-- 13 6 12 5 11 4 10 3 9 2 8 1 7 --

-- init_len:13; flip_operations:7 --
-- : 13 6 12 5 11 4 10 3 9 2 8 1 7
0 : 6 12 5 11 4 10 3 9 2 8 1 7
1 : 6 5 11 4 10 3 9 2 8 1 7
9 : 8 2 9 3 10 4 11 5 6 7
6 : 4 10 3 9 2 8 5 6 7
1 : 4 3 9 2 8 5 6 7
4 : 2 9 3 4 5 6 7
1 : 2 3 4 5 6 7
PUWE: 7
PUWE(13)=7 in 21ms calculated
CALC:  P(15)
-- 15 7 14 6 13 5 12 4 11 3 10 2 9 1 8 --

-- init_len:15; flip_operations:8 --
-- : 15 7 14 6 13 5 12 4 11 3 10 2 9 1 8
1 : 15 14 6 13 5 12 4 11 3 10 2 9 1 8
2 : 14 15 13 5 12 4 11 3 10 2 9 1 8
3 : 13 15 14 12 4 11 3 10 2 9 1 8
6 : 11 4 12 14 15 13 10 2 9 1 8
1 : 11 12 14 15 13 10 2 9 1 8
```

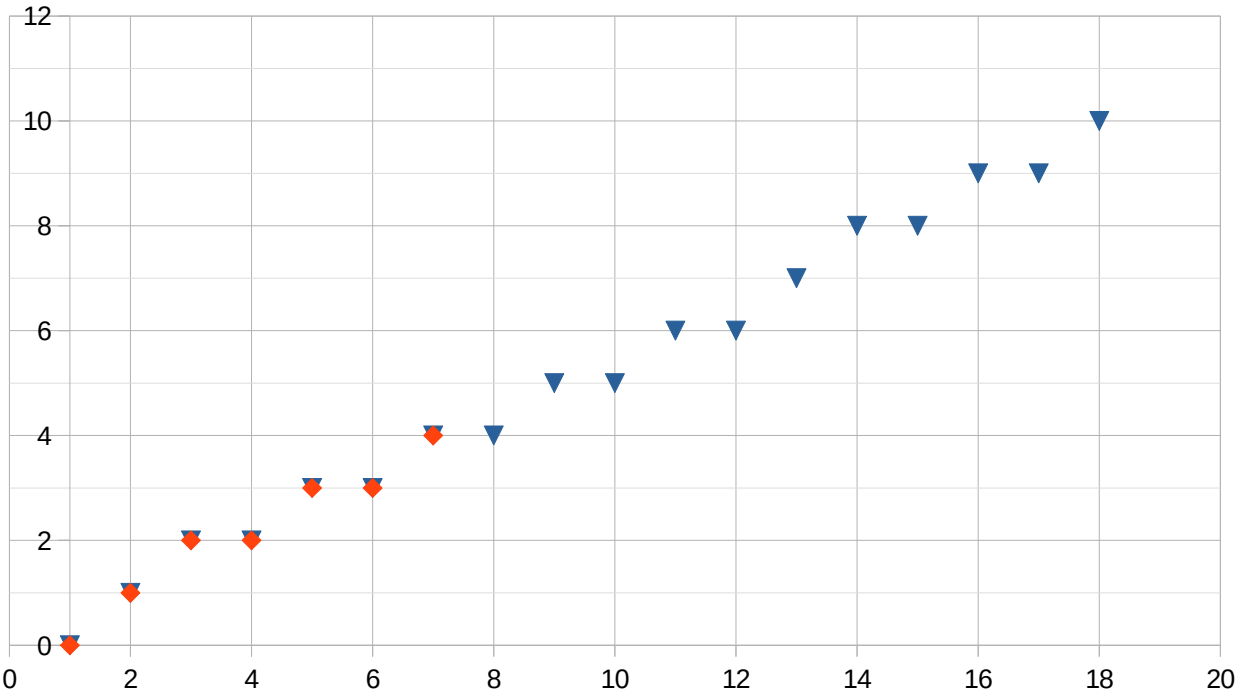
```
6 : 11 4 10 8 2 9 12 13
1 : 11 10 8 2 9 12 13
4 : 2 8 10 11 12 13
PUWE: 8
PUWE(14)=8 in 3231ms calculated
CALC:  P(16)
-- 16 8 15 7 14 6 13 5 12 4 11 3 10 2 9 1 --

-- init_len:16; flip_operations:9 --
-- : 16 8 15 7 14 6 13 5 12 4 11 3 10 2 9 1
0 : 8 15 7 14 6 13 5 12 4 11 3 10 2 9 1
1 : 8 7 14 6 13 5 12 4 11 3 10 2 9 1
11 : 10 3 11 4 12 5 13 6 14 7 8 9 1
1 : 10 11 4 12 5 13 6 14 7 8 9 1
4 : 12 4 11 10 13 6 14 7 8 9 1
1 : 12 11 10 13 6 14 7 8 9 1
9 : 9 8 7 14 6 13 10 11 12
5 : 6 14 7 8 9 10 11 12
1 : 6 7 8 9 10 11 12
PUWE: 9
PUWE(16)=9 in 5861ms calculated
CALC:  P(18)
-- 18 9 17 8 16 7 15 6 14 5 13 4 12 3 11 2 10 1 --


-- init_len:18; flip_operations:10 --
-- : 18 9 17 8 16 7 15 6 14 5 13 4 12 3 11 2 10 1

0 : 9 17 8 16 7 15 6 14 5 13 4 12 3 11 2 10 1
3 : 8 17 9 7 15 6 14 5 13 4 12 3 11 2 10 1
6 : 6 15 7 9 17 8 5 13 4 12 3 11 2 10 1
1 : 6 7 9 17 8 5 13 4 12 3 11 2 10 1
4 : 17 9 7 6 5 13 4 12 3 11 2 10 1
12 : 10 2 11 3 12 4 13 5 6 7 9 17
6 : 4 12 3 11 2 10 5 6 7 9 17
1 : 4 3 11 2 10 5 6 7 9 17
4 : 2 11 3 4 5 6 7 9 17
1 : 2 3 4 5 6 7 9 17
PUWE: 10
PUWE(18)=10 in 332093ms calculated
```

```
4 : 15 14 12 11 10 2 9 1 8
8 : 1 9 2 10 11 12 14 15
1 : 1 2 10 11 12 14 15
PUWE: 8
PUWE(15)=8 in 3777ms calculated
CALC:  P(17)
-- 17 8 16 7 15 6 14 5 13 4 12 3 11 2 10 1 9 --

-- init_len:17; flip_operations:9 --
-- : 17 8 16 7 15 6 14 5 13 4 12 3 11 2 10 1 9
1 : 17 16 7 15 6 14 5 13 4 12 3 11 2 10 1 9
4 : 15 7 16 17 14 5 13 4 12 3 11 2 10 1 9
1 : 15 16 17 14 5 13 4 12 3 11 2 10 1 9
4 : 14 17 16 15 13 4 12 3 11 2 10 1 9
7 : 12 4 13 15 16 17 14 11 2 10 1 9
1 : 12 13 15 16 17 14 11 2 10 1 9
5 : 17 16 15 13 12 11 2 10 1 9
9 : 1 10 2 11 12 13 15 16 17
1 : 1 2 11 12 13 15 16 17
PUWE: 9
PUWE(17)=9 in 122147ms calculated
P(19)

P(19) hatte 4h Zeit um eine Antwort zu liefern.
Der Prozess wurde gestoppt, da die Lüfter mich
Nachts wach halten.
```

Theoretische Analyse – Teilaufgabe B

Identische PUWE-Zahlen

In der abgebildeten Grafik sind die PUWE-Zahlen abgebildet. (X-Achse: n; Y-Achse: P(n))
Rot: Bereits bekannte PUWE-Zahlen des BWINF; Blau: Berechnete PUWE-Zahlen



Sofort fällt eine Art linearer Zusammenhang zwischen n und P(n) auf. Auch ist die selbe PUWE-Zahl, wie in der Lösungsidee beschrieben, maximal 2 mal vorhanden. Dies liegt an dem Zusammenhang zwischen geraden und ungeraden Zahlen. Eine ungerade Zahl ist in der Regel nur ein Teil des Problems für die nächstgerade Zahl, da bei dieser einfach „eine Zahl stehen bleibt“. Eine Ausnahme bildet der Wert 13, ab dem sich dieses Phänomen umzukehren scheint. Die 13 ist der einzige Wert innerhalb der Zahlenliste, der aus der reihe tanzt und keinen anderen Stapel mit der Selben PUWE Zahl hat. Womöglich hat dieser Zusammenhang etwas mit Primzahlen zu tun, jedoch wird ein größeres Datenset benötigt, um eine konktete Aussage zu treffen.

Laufzeit und Speicher

Die Laufzeit des Algorithmus für eine PUWE-Zahl beträgt im schlimmsten Fall $O((n*n-1*n-2...*P(n-1))+(n*n-1*n-2...*P(n-1)+1))$, da hier zunächst alle Wege für die vorherige PUWE-Zahl abgefragt wird und anschließend im letzten möglichen Weg der Tiefe $P(n-1)+1$ die Lösung gefunden wird. Für den Fall, dass $P(n)=P(n+1)$ fällt der Zweite Summand weg.

Der Speicheraufwand dieses Algorithmus relativ gering verglichen mit anderen Optimalen Algorithmen, wie beispielsweise einer Bruteforce Lösung. (Natürlich von der Implementierung abhängig) Dies ist der Fall, da durch die Rekursion nie der gesamte Baum, sondern nur die Aktuell gefragte stelle des Baumes geladen wird. (maximaler Speicherbedarf beim Testen:~60Mib)

Geht da noch was?

Der Algorithmus kann nochmal deutlich beschleunigt werden, indem Parallelität angewendet werden würde. Aktuell findet die gesamte Berechnung innerhalb eines Threads statt. Die Form des Algorithmus erlaubt es jedoch, einzelne Zweige in separaten Theads zu berechnen, was einen deutlichen Leistungsschub verspricht. Dies wurde hier nicht implementiert, da der Algorithmus wie er aktuell arbeitet Problemlos und in relativ kurzer Zeit P(8)-P(18) berechnen kann und die Anforderungen P(8)-P(11) waren.

mögliche Erweiterungen

Mit dem aktuellem Ansatz ist es durch eine Modifizierung der „rateStack“ Funktion recht einfach das Verhalten des Sortieralgorithmus zu verändern.

Erweiterungsidee 1: Möglichst weit oben Flippen

Auf den Sachverhalt bezogen kann es recht schwer sein, einen Pfannkuchenstapel der Länge n > 15 von unten zu wenden. Eine mögliche Erweiterung wäre es, die Wendeoperationen so zu gestalten, dass möglichst nur oberhalb einer bestimmten grenze eine Wendeoperation durchgeführt werden muss. Dies Funktioniert natürlich nicht immer, da die untere Hälfte des Stapels auch noch sortiert werden muss, aber es kann beispielsweise zuerst die obere Hälfte des Stapels und anschließend die untere Hälfte sortiert werden.

Erweiterungsidee 2: Pfannkuchen dazu packen

In dieser Erweiterungsidee kocht der Koch die Pfannkuchen und Packt alle x Wendeoperationen einen zusätzlichen Pfannkuchen der nächst großen Größe oben drauf. Ziel ist wieder ein Stapel, der sortiert ist, und weggenommen werden kann, so dass ein „neuer Stapel“ vom Koch begonnen werden kann. Dieses Problem ist nochmal weitaus komplexer, da das Hinzufügen eines neuen Pfannkuchens die im vorgestellten Algorithmus verwendete Heuristik nicht mehr anwendbar macht, da der Bewertungsalgorithmus nur mit statischen Längen arbeiten kann.

Eine Möglichkeit, den Bewertungsalgorithmus zu modifizieren, damit er die gewünschte Funktionalität hat, wäre es die Position und Werte der Zahlensequenzen zu berücksichtigen und nicht nur die Längen der Listen. Zusätzlich kann der Selbe stapel abhängig von dem Intervall x in dem ein Neuer Pfannkuchen hinzugefügt wird anders bewertet werden. Die Laufzeit des Algorithmus würde keine großen Veränderungen davontragen, da lediglich die „rateStack“ Funktion grundlegend verändert werden müsste. Zusätzlich müsste natürlich noch das hinzufügen der Pfannkuchen berücksichtigt werden. Der Speicheraufwand dürfte ebenfalls ähnlich ausfallen, da keine Variablen grundlegend verändert werden müssen. Es kommt nur der Speicheraufwand für den zusätzlichen Pfannkuchen hinzu.

Quellcode – Pancake Klasse

```python
class P: #P = PancakeStack
    def __init__(self, stack, history=[], initStack=[]):
        self.stack = stack
        self.history = history
        if(history == []): self.initStack = stack
        else: self.initStack = initStack

    def flip(self, i):
        stack = self.stack[:]
        for j in range((i + 1) // 2):
            stack[j], stack[i - j] = stack[i - j], stack[j]
        self.history.append(i)
        stack.pop(0)
        self.stack = stack[:]

    def printHistory(self):
        print("-- init_len:"+ str(len(self.initStack))+ ";
flip_operations:"+ str(len(self.history))+ " --\n")
        print("-- : "+str(self.initStack))
        stack = P(self.initStack)
        for flipindex in self.history:
            stack.flip(flipindex)
            flipindex = str(flipindex)
            if (len(flipindex) ==1): flipindex+=" "
            print(str(flipindex)+" : "+str(stack.stack))

        print("PUWE: "+str(len(self.history)))

    def isValid(self):
        lst = list(self.stack)
        return all(lst[i] < lst[i+1] for i in range(len(lst)-1))

    def copy(self):
        return P(self.stack.copy(), self.history.copy(),
                                        initStack=self.initStack)

    def getFlipIndexPoints(self):
        return list(range(0, len(self.stack)))

def flip_and_eat(stack, idx):
    stack[:idx+1] = reversed(stack[:idx+1])
    stack.pop(0)
    return stack
```

Quellcode – Teilaufgabe A

```
#main-Funktion des Optimierten Lösungsverfahrens
def main(filename, n, m):
    with open(filename, "r") as file:
        num=[]
        for line in file.read().splitlines()[1:]:
            num.append(int(line))

    initStack = P(num, [])
    del num

    pkListA = [initStack.copy()]
    pkListB:list = []
    while pkListA:
        for pancake in pkListA:
            flipPoints = pancake.getFlipIndexPoints()
            mutationPancakes = []
            for i in flipPoints:
                newPancake = pancake.copy()
                newPancake.flip(i)
                if newPancake.isValid():
                    newPancake.printHistory()
                    return 0
                mutationPancakes.append((newPancake,
rateStack(newPancake)))
                del newPancake
            mutationPancakes.sort(key = lambda x: x[1], reverse=True)
            pkListB += (first(n, mutationPancakes))
        pkListB.sort(key=lambda x: x[1], reverse=True)
        pkListB = list(map(lambda x: x[0], pkListB))
        pkListA = first(m, pkListB)
        pkListB = []

# Bewertungsfunktion
def rateStack(stack):
    stack = stack.stack.copy()
    sums = [
        sum(map(lambda x : len(x),
                _getAscendingSequences(stack)
            )),
        sum(map(lambda x : len(x),
                _getDescendingSequences(stack)
            ))
        ]
    return max(sums)+(0.2*min(sums))
```

```python
#aufsteigende / absteigende Zahlenfolgen

def _getAscendingSequences(numbers, n=[], numbers_copy=[]):
    sequences = []
    current_sequence = [numbers[0]]
    for i in range(1, len(numbers)):
        if numbers[i] > numbers[i-1]:
            current_sequence.append(numbers[i])
        else:
            sequences.append(current_sequence)
            current_sequence = [numbers[i]]
    sequences.append(current_sequence)

    del current_sequence, i

    returnObj = []
    for sequence in sequences:
        valid=True
        sequenceGap = list(range(min(sequence), max(sequence)+1))
        for i in sequence:sequenceGap.remove(i)
        for i in sequenceGap:
            if i in numbers or i in n:
                valid = False
                break
        if valid:
            returnObj.append(sequence)
        else:
            new_n = numbers_copy+numbers
            for j in sequence: new_n.remove(j)
            sequence.insert(_getInsertIndex(sequence, i), -1)
            returnObj += _getAscendingSequences(sequence, new_n,
numbers)

    for i in range(len(returnObj)):
        returnObj[i] = list(filter(lambda x: x != -1 ,returnObj[i]))
    return [lst for lst in returnObj if len(lst) > 1]


def _getDescendingSequences(numbers):
    numbers.reverse()
    listreverse = _getAscendingSequences(numbers)
    for i in listreverse:i.reverse()
    return listreverse
```

Quellcode – Teilaufgabe B

```
class Calc
{
public:
    int n;
    std::vector<int> predictedPUWE;
    int d;
    Calc(int n, std::vector<int> predictedPUWE) : n(n),
                                predictedPUWE(predictedPUWE), d(0) {}

    int main()
    {
        std::cout << "\n\n-----------------\nCALC:  P(" << n << ")\n";

        std::vector<int> num = reisverschluss(n);

        P initStack(num);
        std::cout << "-- ";
        for (int i : num)
            std::cout << i << ' ';
        std::cout << "--\n";

        for (int i : predictedPUWE)
        {
            int a = find(initStack, i);
            if (std::find(predictedPUWE.begin(), predictedPUWE.end(), a)
                                                != predictedPUWE.end())
                return a;
        }
        return -1;
    }
private: int find(P stack, int wantedPUWE)
    {
        int stackPUWE = stack.history.size();
        if (stackPUWE < wantedPUWE)
        {
            for (int i : stack.getFlipIndexPoints())
            {
                P newStack = stack.copy();
                newStack.flip(i);
                int status = find(newStack, wantedPUWE);
                if (status == -1)
                    continue;
                else if (status == wantedPUWE)
                    return wantedPUWE;
```

```
            }
        }
        else if (stackPUWE == wantedPUWE)
        {
            if (stack.isValid())
            {
                std::cout << "\n";
                stack.printHistory();
                return stackPUWE;
            }
            else
                return -1;
        }
        else
            return -1;
        return -1;
    }
};
```

```
int main()
{
    std::vector<int> last_predictedPUWE = {4, 5, 6};
    std::vector<int> history = {4};
    for (int i = 8; i <= 100; ++i)
    {
        Calc calculator(i, last_predictedPUWE);
        auto start = std::chrono::high_resolution_clock::now();
        int puwe_i = calculator.main();
        auto end = std::chrono::high_resolution_clock::now();
        auto int_ms =
  std::chrono::duration_cast<std::chrono::milliseconds>(end - start);
        std::cout << "PUWE(" << i << ")=" << puwe_i << " in " <<
                          int_ms.count() << "ms calculated" << std::endl;
        history.push_back(puwe_i);
        if (history.back() == history[history.size() - 2])
        {
            last_predictedPUWE = {puwe_i + 1, puwe_i + 2, puwe_i + 3};
        }
        else
        {
            last_predictedPUWE = {puwe_i, puwe_i + 1, puwe_i + 2};
        }
    }
    return 0;
}
```