

Aufgabe 3: Pancake Sort

Teilnahme-ID: 66692

Bearbeiter dieser Aufgabe:
Sven Zimmermann

March 2023

Inhalt

1. Lösungsidee	1
1.1 Aufgabe 3a: A(S).....	2
1.2 Aufgabe 3b: PWUE-Zahl	2
1.2.1 Ausschließen von Permutationen	5
1.2.2 Rekursiver Ansatz	5
2 Umsetzung.....	7
2.1 Aufgabe 3a: A(S).....	7
2.1.1 Implementierung der PWUE-Funktion	7
2.1.2 Implementierung der Brute-Force-Methode zur Bestimmung der Lösungssequenz.....	7
2.1.3 Optimierung	8
2.2 Aufgabe 3b: PWUE-Zahl	8
2.2.1 Speicherung der Schichten	8
2.2.2 Generieren aller Stapelpermutationen	8
2.2.3 ‚Kürzen‘ der Liste	9
2.2.4 Generieren der rekursiven Schichten	9
3 Beispiele	10
3.1 Aufgabe 3a.....	10
3.2 Aufgabe 3b.....	10
4 Quellcode: Implementierung in Python.....	11
4.1 Aufgabe 3a.....	11
4.2 Aufgabe 3b.....	12

1. Lösungsidee

Bei der Aufgabe „Pancake-Sort“ sollte ein Stapel an Pfannkuchen mit unterschiedlichen Größen durch die Ausführung eines speziellen „Pfannkuchen-Wende-Und-Ess-Algorithmus“ (kurz: PWUE) von klein nach groß sortiert werden. Der Algorithmus kann an jeder Stelle des Stapels angewendet

werden und kehrt den Teilstapel vom obersten (ersten) bis zur gewählten Stelle um und entfernt das nun oberste Objekt. Das Ziel soll sein, den Pfannkuchen-Stapel mit so wenigen Funktions-Anwendungen wie möglich ($A(S)$ -Zahl) zu sortieren.

1.1 Aufgabe 3a: $A(S)$

Die minimale Anzahl an Anwendungen der PWUE-Funktion zur Sortierung eines Stapels S sei die Zahl $A(S)$. Festzustellen ist, dass mit jeder Durchführung der PWUE-Funktion sich die Stapelgröße um 1 verkleinert, sodass die maximale Menge an Durchführungen der Größe des Stapels minus 1 entspricht. Nach dieser Menge an Durchführungen ist zwangsweise nur noch ein Pfannkuchen im Stapel und er ist somit sortiert. Um nun die minimal nötige Menge an Anwendungen der PWUE-Funktion zur Sortierung eines bestimmten Stapels zu finden, ist ein Algorithmus denkbar, der alle Möglichkeiten, die PWUE-Funktion anzuwenden, durchprobiert, und dann die effektivste Ausgabe (Abb. 1.1):

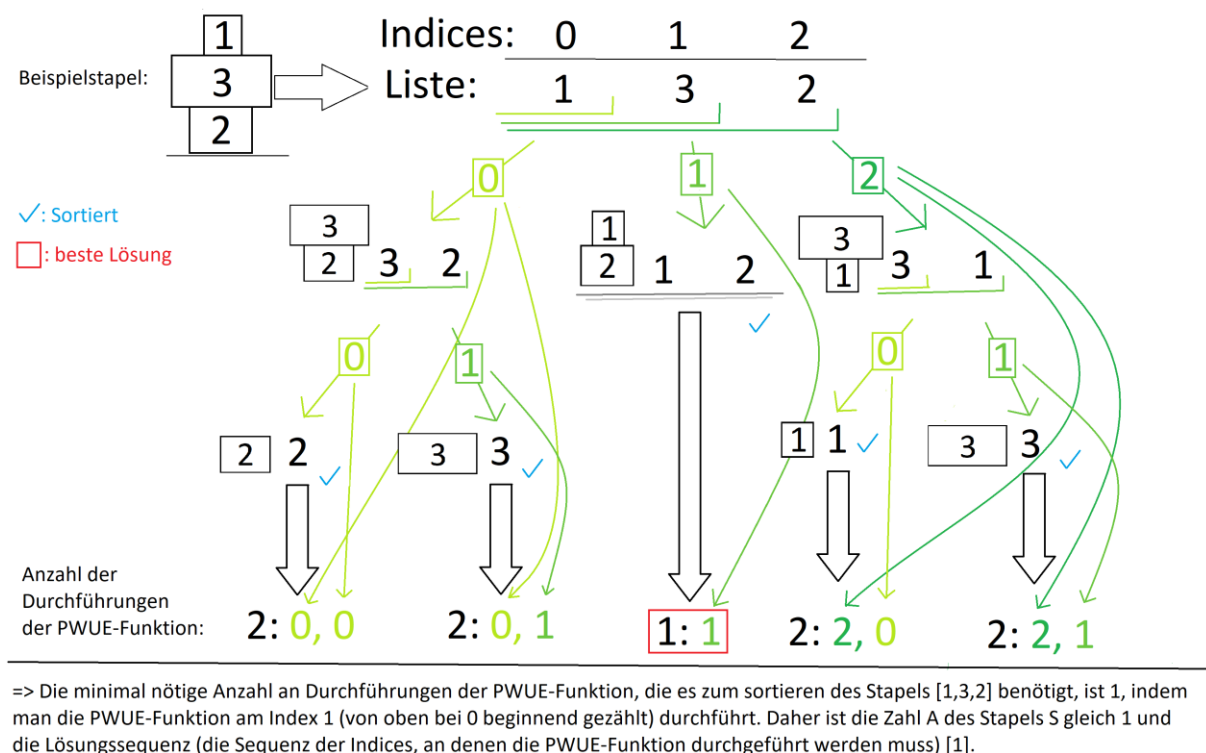


Abbildung 1.1: Beispielhafter Ablauf eines Algorithmus zur Bestimmung der Zahl A für den Stapel $S=[1,3,2]$.

1.2 Aufgabe 3b: PWUE-Zahl

Aufbauend auf dem Algorithmus zur Bestimmung der minimal nötigen Zahl $A(S)$ an Durchführungen der PWUE-Funktion zur Sortierung eines Stapels S , soll nun ein Algorithmus entwickelt werden, der für eine Stapelgröße n die maximale Größe für $A(S)$ bestimmt. Das heißt, es soll derjenige Stapel aus allen möglichen Stapeln mit der Länge n gefunden werden, dessen optimale Lösungssequenz zu seiner Sortierung am längsten ist und dann die Länge der Lösungssequenz, sowie der Stapel, ausgegeben werden. Es soll quasi der ‚am wenigsten sortierte‘ Stapel mit der Länge n gefunden werden.

Für die Reihe der PWUE-Zahlen ergeben sich folgende Erkenntnisse:

- Die PWUE-Zahl für $n=1$ ist 0, da ein Stapel mit nur einem Pfannkuchen unbedingt richtig sortiert ist.
- Da Pfannkuchen der gleichen Größe, wenn sie übereinander liegen, sortiert sind (z.B. $[1,2,1]$) und statt Dopplungen eine der Größen von der anderen verschieden sein könnte (z.B. $[1,2,3]$), wodurch eine zusätzliche Sortierung und somit potentiell eine längere Lösungssequenz nötig wäre, ist die Betrachtung ‚redundante Größen enthaltene‘ Stapel irrelevant. Es sind also lediglich alle $n!$ Permutationen der Pfannkuchen der Größe 1 bis n in einem Stapel der Größe n auf ihre Lösungssequenz zu untersuchen.
- Die PWUE-Zahl von n lässt sich rekursiv auf den Daten der Stapel mit der Länge $n-1$ aufbauen, da sich durch die Anwendung der PWUE-Funktion auf einen Stapel der Länge n die Länge des Stapels um 1 reduziert und der Stapel somit Teil aller möglichen Stapel der Länge $n-1$ sein muss. Hierbei ist zu beachten, dass die eigentlichen Größen der Elemente des Stapels irrelevant für die Sortierung sind. Nur das Verhältnis der Zahlen zueinander ist relevant (vgl. Abb. 1.2). Somit würde zwar der Stapel $[4,2,3,1]$ durch das Anwenden der PWUE-Funktion an Index 3 zu $[3,2,4]$, welcher nicht Teil der Permutationen der Größe 1 bis 3 im Stapel der Größe 3 ist, jedoch ist er von seinem Sortiervorgang identisch zu dem Stapel $[2,1,3]$.
- Die PWUE-Zahl von n ($P(n)$) kann sich von der PWUE-Zahl von $n-1$ ($P(n-1)$) nur um 0 oder +1 unterscheiden, da nach einer Durchführung der PWUE-Funktion (+1) mit einem Stapel zwangsweise ein Stapel S mit der Größe $n-1$ entsteht, welcher selber wiederum maximal eine Zahl $A(S)$ besitzt, die der PWUE-Zahl von $n-1$ entspricht:

$Q(n)$: Menge der Stapel S mit Länge n , $S_n \in Q(n), n \in \mathbb{N}^{\neq 0}$

$PWUE - Func(S_n, i) = f(S_n)$, Stapel S_n , Index i mit $i \in \mathbb{N}^{\leq n}$

$A(S)$: siehe Kapitel 1.1

$R(n)$: Menge aller Stapel S_n mit $A(S_n) = P(n)$, $R(n) \subseteq Q(n)$

$A(S_n) \leq P(n)$, $A(S_n) = 1 + A(f(S_n))$, $\Leftrightarrow i = \text{erste Zahl der Lösungssequenz}$

$f(S_n) \in Q(n-1) \Rightarrow A(f(S_n)) \leq P(n-1)$

$\Rightarrow 1. A(S_n) \leq P(n-1)$, $\forall S_n | f(S_n) \notin R(n-1)$

$\Rightarrow 2. A(S_n) = P(n-1) + 1$, $\forall S_n | f(S_n) \in R(n-1)$

$P(n) = P(n-1)$, $\vee P(n) = P(n-1) + 1$

- Da $P(n)$ nicht linear mit n wächst, sondern langsamer (vgl. Abb. 1.3), lässt sich kein Stapel, als nicht notwendig zur Bestimmung der Lösungssequenz eines größeren Stapels, ausschließen.

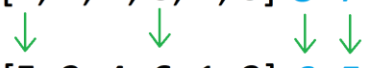
Beispielstapel: $[7, 2, 4, 8, 1, 3]$ $8 > 7 > 4 > 3 > 2 > 1$

 Ist für das Sortieren
 das Gleiche wie: $[5, 2, 4, 6, 1, 3]$ $6 > 5 > 4 > 3 > 2 > 1$

Abbildung 1.2: Relevanz der Absoluten Objektgrößen für das Sortieren.

n	1	2	3	4	5	6	7	8	9	10	11	12	13
$P(n)$	0	1	2	2	3	3	4	?	?	?	?	?	?

Abbildung 1.3: Vorgegebene $P(n)$ Zahlen bis $n=7$.

Es gilt also für einen Algorithmus lediglich zu prüfen, ob es einen Stapel der Länge n gibt, der eine längere Lösungssequenz hat, als der ‚unsortierteste‘ der nächst kürzeren Länge, da ohnehin bekannt ist, dass die PWUE-Zahl der Länge n mindesten der PWUE-Zahl der Länge $n-1$ entspricht.

Das Problem, bei der Bestimmung der PWUE-Zahlen, ist der rapide Anstieg in Rechenintensität mit steigender Größe n , da die Menge der zu überprüfenden Stapel mit der Fakultät wächst und da für jeden Stapel die Lösungssequenz bestimmt werden muss, wessen Rechenaufwand ebenfalls mit der Größe des Stapels rapide anwächst. Es wäre deutlich einfacher, wenn ein Algorithmus direkt die Stapel generieren könnte, welche die längste Lösungssequenz besitzen, sodass nicht $n!$ Stapel überprüft werden müssen, sondern eine konstante Menge. Würde es sich bei dem Problem um die reguläre Sortierung einer Liste handeln, so wäre man dazu geneigt, einfach nur diejenigen Listen zu überprüfen, die am ‚wenigsten sortiert‘ sind. Es wirkt nun zunächst einleuchtend, dass selbiges auch für die Sortierung mit der PWUE-Funktion gilt, da bereits teilweise sortierte Stapel weniger Anwendungen der PWUE-Funktion zur Sortierung benötigen (vgl. [3,2,1] oder [4,1,2,3]). Es zeigt sich auch, dass besonders unsortierte Stapel lange Lösungssequenzen besitzen, jedoch müssen jene nicht die Längsten sein. Generiert man zum Beispiel immer die Stapel, welche die größtmöglichen Größendifferenzen zwischen benachbarten Pfannkuchen haben (Beispielsweise für $n=10$: [6, 1, 7, 2, 8, 3, 9, 4, 10, 5]), so ergeben sich folgende PWUE-Zahlen:

n	$P(n)$	möglichst unsortierter Stapel	Menge der möglichen Stapel (Permutationen)
0	/	[]	1
1	0	[1]	1
2	1	[2,1]	2
3	2	[2,3,1]	6
4	2	[4,1,3,2]	24
5	3	[4,1,5,3,2]	120
6	3	[5,1,6,2,4,3]	720
7	4	[5,1,6,2,7,4,3]	5040
8	4	[6,1,7,2,8,3,5,4]	40320
9	5	[6,1,7,2,8,3,9,5,4]	362880
10	6	[5,10,4,9,1,7,2,8,6,3]	3628800 \approx 3,5 Mio.
11	6	[7,1,8,2,9,3,10,4,11,6,5]	39916800 \approx 40 Mio.
12	7	[7,1,8,2,9,3,10,4,11,5,12,6]	479001600 \approx 480 Mio.
13	7	[8,1,9,2,10,3,11,4,12,5,13,7,6]	6227020800 \approx 6,2 Mrd.
14	8	[9,1,10,2,11,3,12,4,13,5,14,6,8,7]	87178291200 \approx 87 Mrd.
15	9	[9,1,10,2,11,3,12,4,13,5,14,6,15,8,7]	1307674368000 \approx 1300 Mrd.

Das Problem ist, dass es keine Garantie gibt, dass diese Zahlen stimmen müssen und, wie sich später zeigen wird, ist mindestens die PWUE-Zahl von $n=8$ falsch.

Es ergibt sich demnach kein eindeutiges Muster für die Stapel mit der längsten Lösungssequenz.

Die einzige Möglichkeit ist somit, jede Permutation durchzuprüfen und ihre Lösungssequenz zu ermitteln.

1.2.1 Ausschließen von Permutationen

Wie sich gezeigt hat, ist es leider nicht möglich direkt die ‚am wenigsten sortierten‘ Stapel zu generieren, sodass die Permutationen einzeln geprüft werden müssen. Hierbei lassen sich jedoch einige von den $n!$ Permutationen ausschließen. So sind alle Permutationen, bei denen der größte Pfannkuchen ganz unten liegt, auszuschließen, da jene, wie ein Stapel der Größe $n-1$ zu sortieren ist, da der unterste Pfannkuchen bei der Sortierung ignoriert werden kann. Selbiges gilt nicht für den kleinsten Pfannkuchen mit der obersten Position, da beim Sortieren mit der PWUE-Funktion, die Positionen der oberen Pfannkuchen im Stapel vertauscht werden.

Somit verbleiben jedoch immer noch $(n-1)(n-1)!$ Permutationen, welche signifikant weiter reduziert werden können. Das Bestimmen der PWUE-Zahl für große n wäre somit immer noch sehr rechenintensiv.

1.2.2 Rekursiver Ansatz

Das Problem in der Rechenintensität liegt zum einen in der enormen Menge an Permutationen mit steigender Länge n und zum anderen in dem steigenden Rechenaufwand bei der rekursiven Bestimmung der Lösungssequenz der einzelnen Stapel. Statt nun zu versuchen einzelne Permutationen aufgrund verschiedener Kriterien für die Bestimmung der PWUE-Zahl auszuschließen, ist ein rekursiver Ansatz möglich der darauf aufbaut, dass durch das Anwenden der PWUE-Funktion auf einen Stapel der Länge n ein Stapel der Länge $n-1$ entsteht, welcher wiederum durch das Anwenden der PWUE-Funktion zu einem Stapel der Länge $n-2$ wird, und so weiter. So wäre die Bestimmung der Lösungssequenz eines Stapels der Länge n deutlich einfacher, wenn bereits die Lösungssequenzen aller Stapel der Länge $n-1$ bekannt wären, da lediglich die Länge der Lösungssequenzen aller, durch das Anwenden der PWUE-Funktion auf den Stapel erreichbaren, Stapel der Länge $n-1$ verglichen werden müssten und dann die kürzeste um die entsprechende Anwendung der PWUE-Funktion, die zu dem entsprechenden Stapel geführt hat, der Lösungssequenz des Stapels hinzugefügt werden müsste. Da nun für die Bestimmung der PWUE-Zahl lediglich die Längen der Lösungssequenzen relevant sind, lassen sich bei diesem Verfahren die Lösungssequenzen auf ihre Länge reduzieren. Die Längen der Lösungssequenzen der Stapel der Länge $n-1$ lassen sich nun ebenso rekursiv bestimmen, wie die Lösungssequenzen der Stapel der Länge n , sodass sich, lediglich mit dem Wissen, dass die Länge der Lösungssequenz des Stapels [1] gleich 0 ist und dass die Lösungssequenz bereits sortierter Stapel (Beispielsweise: [1,2,3,4,5,6]) ebenfalls gleich 0 ist, alle PWUE-Zahlen, unabhängig von den spezifischen Lösungssequenzen, bestimmen lassen (vgl. Abb. 1.4).

Permutationen für einen
Stapel der Größe $n=1$:

Permutationen
für einen Stapel
der Größe $n=2$:

Permutationen
für einen Stapel
der Größe $n=3$:

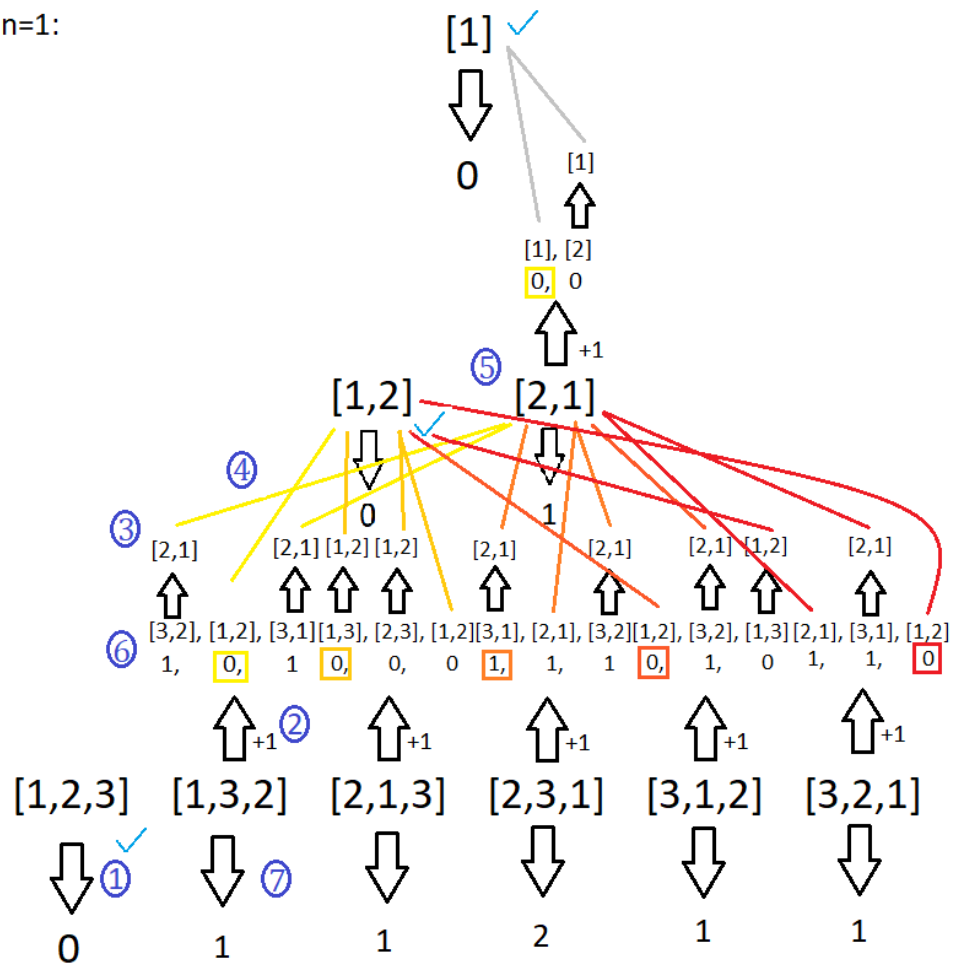


Abbildung 1.4: Rekursiver Ablauf zur Bestimmung der PWUE-Zahl für $n=3$.

1. Überprüfe ob der Stapel bereits sortiert ist und falls ja ist die Länge seiner Lösungssequenz 0.
2. Falls der Stapel nicht bereits sortiert ist, führe die PWUE-Funktion an jedem Index aus und speichere die dabei entstehenden Stapel.
3. Falls ein Stapel nicht maximal ‚gekürzt‘ ist, ersetze die Zahlen sodass ihre Verhältnisse untereinander gleich bleiben (vgl. Abb. 1.2), jedoch alle zwischen 1 und $n-1$ (n ist die ursprüngliche Stapelgröße und $n-1$ die Größe des Stapels nach dem einmaligen Ausführen der PWUE-Funktion) liegen.
4. Suche den Stapel in der vorherigen ‚Ebene‘ und übernehme die Länge seiner Lösungssequenz.
5. Falls die vorherige ‚Ebene‘ nicht die ‚Ebene‘ der Stapel der Größe 1 ist, werden zur Bestimmung der Länge der Lösungssequenzen dieser Stapel die Schritte 1-6 wiederholt. Falls die vorherige ‚Ebene‘ die ‚Ebene‘ der Stapel der Größe 1 ist, so nimm 0 als Länge der Lösungssequenz.
6. Wähle aus allen Lösungen der möglichen Stapel, die sich aus der unterschiedlichen Anwendung der PWUE-Funktion ergeben, die kürzeste Lösungssequenz.
7. Die Länge der Lösungssequenz der Permutation ist dann die kürzeste mögliche Lösungssequenz (Schritt 6) plus eins (einmaliges Ausführen der PWUE-Funktion).

Der Vorteil dieser Methode ist, dass nicht für jede Stapel-Permutation der Größe n , $n!$ Mögliche Anwendungen der PWUE-Funktion durchprobiert werden müssen (für die Bestimmung der Zahl $A(S)$), sondern lediglich n . Dafür müssen die vorherigen ‚Ebenen‘ generiert werden, was insgesamt trotzdem auf eine deutlich geringere Rechenleistung herauskommt:

Sei N die Menge der zu testenden Stapel für die PWUE – Zahl von n :

$$N_{BruteForce} = \sum_{k=1}^{n!} n! = n!^2$$

$$N_{Ebenen} = \sum_{k=1}^n k * k! \leq n^2 * n! \ll n!^2$$

n	$n^2 * n!$	$(n!)^2$
1	1	1
2	8	4
5	3000	14400
10	362880000 \approx 363 Mio.	13168189440000 \approx 13 Bio.
15	294226732800000 \approx 294 Bio.	1710012252724199424000000 \approx 1,7 Quadrillion
20	973160803270656000000 \approx 973 Trill	5919012181389927685417441689600000000 \approx 6 Sextillion

Abbildung 1.5: Vergleich des Wachstums von $n^2 * n!$ gegenüber $(n!)^2$.

2 Umsetzung

2.1 Aufgabe 3a: A(S)

2.1.1 Implementierung der PWUE-Funktion

Die PWUE-Funktion benötigt zwei Inputs. Zum einen den zu Verändernden Stapel und zum anderen den Index, an welchen die Funktion ansetzen soll. Das Verfahren ist dann folgendes:

1. Teile den Stapel am Index.
2. Invertiere die Reihenfolge der Größen des oberen Teilstapels.
3. Entferne das oberste Objekt des oberen Teilstapels.
4. Füge beide Teile wieder zusammen (oben oben und unten unten).

Da in Python 3.10 das invertieren eine Standardfunktion der Liste ist, nutze ich statt eines Stapels eine Liste, wobei das erste Objekt das oberste ist und das letzte Objekt das unterste. Desweiteren sind nur die Größen der Pfannkuchen relevant, sodass die Liste lediglich jene speichert.

2.1.2 Implementierung der Brute-Force-Methode zur Bestimmung der Lösungssequenz

Um die Lösungssequenz zur Sortierung einer Liste mit der PWUE-Funktion zu finden, ist die einfachste Methode die Brute-Force-Methode, wobei alle Möglichkeiten die PWUE-Funktion anzuwenden durchprobiert werden und die herausgefiltert werden, die funktionieren. Da sich die Listen nach jeder Anwendung der PWUE-Funktion verändern, man jedoch nicht ohne weiteres bestimmen kann, ob die Veränderung in die richtige Richtung geht, ist die Implementierung der Methode als rekursive Variante naheliegend. Hierbei nimmt die Methode die Liste als Input und ruft, falls die Liste nicht bereits sortiert ist, für jede Liste, die sich durch das Anwenden der PWUE-Funktion auf die ursprünglich Liste ergeben kann, sich selber mit jener auf:

1. Überprüfe ob die erhaltene Liste bereits sortiert ist und falls ja gebe dies zurück.
2. Falls nicht, generiere alle möglichen Listen durch Anwenden der PWUE-Funktion auf verschiedene Indices der erhaltenen Liste.
3. Wiederhole den Prozess (1-4) für jede dieser Listen.
4. Prüfe für jede Liste wie lang ihre Lösungssequenz ist und füge der kürzesten Lösungssequenz den Index hinzu, der nötig war zum Erreichen der entsprechenden Liste, und gebe die Gesamtsequenz zurück.

2.1.3 Optimierung

Durch die interne Abarbeitung des rekursiven Algorithmus durch den Computer, welche nicht gleichzeitig, sondern nacheinander erfolgt, lässt sich nach den ersten paar gefundenen Lösungssequenzen schon eine Tendenz für die Länge der besten Lösungssequenz erkennen. Denn wenn beispielsweise die kürzeste bisher gefundene Lösungssequenz 5 Ausführungen der PWUE-Funktion beinhaltet und in einem Lösungsversuch nach 7 Ausführungen der PWUE-Funktion immer noch keine Sortierung der Liste erfolgt ist, ist es unnötig überhaupt noch den Lösungsversuch fortzusetzen, da er ohnehin nicht kürzer als 5 Ausführungen werden kann. Dadurch minimiert sich der Rechenaufwand deutlich, da nach wenigen gefundenen Lösungssequenzen die nötige Menge an zu überprüfenden Kombinationen der Ausführungen der PWUE-Funktion stark sinkt.

2.2 Aufgabe 3b: PWUE-Zahl

Zur Umsetzung der rekursiven Ermittlung der PWUE-Zahlen, gilt es zunächst die Speicherung der verschiedenen ‚Ebenen‘ zu klären. Das Problem ist, dass das Zugreifen der höheren Rekursionsschichten auf die niedrigeren sehr langsam wäre, wenn immer die Stapel miteinander abgeglichen werden müssten. Die Lösung stellt eine Hash-Map beziehungsweise eine assoziative Liste dar, welche jedem Stapel einen eindeutigen Index zuordnet, sodass das Suchen nach dem entsprechenden Index in der Liste der Lösungssequenzlängen der niedrigeren Ebene wegfällt. Die Schwierigkeit besteht nun darin eine entsprechende Hash-Funktion zu erstellen, welche garantiert, dass kein Index nicht-eindeutig zugeordnet wird.

2.2.1 Speicherung der Schichten

Zur Umsetzung der Hash-Funktion zur eindeutigen Bezeichnung einer Liste der Größe n , ist die Darstellung der Listen in alternativen Zahlensystemen heranzuziehen, wobei die Indices der Liste mit der Stelle im Zahlensystem gleichzusetzen sind. In einer Liste der Größe n können, in unserem Fall, nur ‚Ziffern‘ der Größe n vorkommen, sodass bei einer Liste der Größe n das n -Zahlensystem zu verwenden wäre, wobei jede Größe k im Zahlensystem der Ziffer $k-1$ entspricht, da die Größen von 1 bis n vorkommen, während im Zahlensystem nur die Ziffern von 0 bis $n-1$ existieren (vgl. Abb. 2.1).

Größe	2	4	6	7	11
Liste	[2,1]	[3,2,4,1]	[4,3,5,1,2,6]	[3,5,1,7,2,4,6]	[1,2,5,10,3,8,4,9,6,11,7]
Darstellung in Zahlensystem	10_2	2130_4	324015_6	2406135_7	$014927385A6_{11}$
Umrechnung in Dezimalsystem	2	156	26795	304659	$3395493525 \approx 3,4 \text{ Mrd.}$

Abbildung 2.1: Beispielhafte Zuordnung von Zahlenwerten zu einer Liste, durch die Hash-Funktion.

Da das erste Objekt einer Liste in Normalfall mit dem Index 0 nummeriert wird, während das letzte Objekt einer Liste der Länge n mit $n-1$ nummeriert ist, wird in meiner Implementierung der Hash-Funktion dem Index 0 die Einerstelle im Zahlensystem zugeordnet, sodass die Darstellung im Zahlensystem der invertierten Liste, statt der Liste selbst entspricht.

2.2.2 Generieren aller Stapelpermutationen

Ähnlich wie beim Generieren aller Möglichkeiten zur Sortierung eines Stapels durch die PWUE-Funktion (vgl. Kapitel 2.1), ist auch die Generierung aller Stapelpermutationen rekursiv möglich. Hierfür wird beim Zusammenstellen der Listen an jedem Index die bisherige Liste für jede, noch nicht

eingefügte, Zahl kopiert und dann jeweils eine der Möglichkeiten einer der Listen hinzugefügt. Anschließend wird das Verfahren für jede der Listen wiederholt, bis alle Zahlen bis n in jeder Liste genau einmal verwendet wurden:

1. Bisherige Liste für jede noch nicht verwendete Zahl kopieren.
2. An jeweils eine der kopierten Listen eine noch nicht verwendete Zahl anfügen.
3. Prozedur (1-3) für jede Liste solange wiederholen, bis alle Zahlen bis n verwendet wurden.

2.2.3 ‚Kürzen‘ der Liste

Beim Ausführen der PWUE-Funktion auf einen Stapel der Größe n ist es sehr wahrscheinlich, dass der dabei entstehende Stapel nicht nur Elemente von 1 bis $n-1$ enthält, wie es bei der vorherigen Schicht der Fall war. Daher würde beim Ausführen der Hash-Funktion ein falscher Index generiert, wodurch die falsche Lösungssequenzlänge aus der vorherigen Schicht übernommen würde (falls überhaupt eine existiert). Es ist daher nötig, die, durch die PWUE-Funktion generierten, Listen vor dem Generieren des Hash-Schlüssels zu kürzen. Dafür wird jeder Zahl der Liste ihr Index in einer sortierten Version der Liste zugeordnet (vgl. Abb. 2.2).

	ungekürzte Liste	sortierte Liste	gekürzte Liste
Indices	0 1 2 3 4 5	0 1 2 3 4 5	0 1 2 3 4 5
Liste	[7, 2, 4, 8, 1, 3]	[1, 2, 3, 4, 7, 8]	[4, 1, 3, 5, 0, 2]

Abbildung 2.2: Exemplarischer Ablauf von 'Kürzen' einer Liste.

2.2.4 Generieren der rekursiven Schichten

Aufbauend auf die bisherigen Algorithmen (vgl. Kapitel 2.2.1-2.2.3) kann nun eine Methode entwickelt werden, durch die das Bestimmen der PWUE-Zahl einer Größe n deutlich schneller ist, als das iterative durchrechnen aller Lösungssequenzen der Permutationen mit Größe n . Die Methode lässt sich in folgende vier Schritte einteilen (vgl. Abb. 1.4):

1. Generieren der vorherigen Schicht, durch wiederholen der Schritte 1-4 mit der Größe $n-1$, bis $n=1$.
2. Bestimmen aller möglichen Stapel der Größe n (siehe Kapitel 2.2.2).
3. Generieren und kürzen aller Stapel die sich aus jedem Stapel (ausgenommen des bereits sortierten Stapel $[1,2,\dots,n]$, für den gilt die Länge der Lösungssequenz gleich 0) durch das Anwenden der PWUE-Funktion ergeben.
4. ‚Nachschlagen‘ der Länge der Lösungssequenz der generierten Stapel in der vorherigen Schicht durch den Hash-Key des Stapels und übernehmen der Länge der kürzesten Lösungssequenz je Stapel plus eins.

Die Implementierung der Methode in Pseudocode sähe ungefähr so aus:

```

1 Algorithmus: generiere_PWUE-Zahl_von_n
2 Eingabe: n
3 alle_Stapel <- generiere_alle_Stapelpermutationen //vgl. Kapitel 2.2.2
4 falls n gleich 1, tue
5     gebe eine assoziative Liste mit der Länge der Lösungssequenz von 0 am Index 0 zurück
6 vorherige_Schicht <- führe generiere_PWUE-Zahl_von_n mit n-1 aus
7 neue_Schicht <- leere assoziative Liste
8 für jeden Stapel S in alle_Stapel, tue

```

```

9         falls S bereits sortiert ist, tue
10             neue_Schicht am Index des Hash-Funktionswert von S <- 0
11         falls S nicht sortiert ist, tue
12             kürzeste_Lösungssequenz <- Länge von S //"maximalwert", da jede Lösungssequenz kürzer ist als die Länge von S
13             für jeden Index i in S, tue
14                 neuer_Stapel <- gekürzter Stapel S nach dem Anwenden der PWUE-Funktion am Index i
15                 falls die Länge der Lösungssequenz am Index des Hash-Funktionswert von neuer_Stapel kleiner ist als
16 kürzeste_Lösungssequenz, tue
17                     kürzeste_Lösungssequenz <- Länge der Lösungssequenz am Index des Hash-Funktionswert von
18 neuer_Stapel
19             neue_Schicht am Index des Hash-Funktionswert von S <- kürzeste_Lösungssequenz + 1
20 gebe neue_Schicht zurück

```

3 Beispiele

3.1 Aufgabe 3a

Ausgaben meines Programms zu den Beispieleingaben der BWInfo-Webseite (<https://bwinf.de/bundeswettbewerb/41/2/>):

pancace0.txt:

Output:

Lösungssequenz: [5, 3] Länge: 2

pancace1.txt:

Output:

Lösungssequenz: [3, 4, 5] Länge: 3

pancace2.txt:

Output:

Lösungssequenz: [2, 3, 3, 5] Länge: 4

pancace3.txt:

Output:

Lösungssequenz: [1, 2, 3, 3, 3, 6] Länge: 6

pancace4.txt:

Output:

Lösungssequenz: [1, 1, 11, 1, 4, 7, 1] Länge: 7

pancace5.txt:

Output:

Lösungssequenz: [1, 13, 6, 8, 4, 5] Länge: 6

pancace6.txt:

Output:

Lösungssequenz: [2, 2, 5, 6, 11, 1, 5, 1] Länge: 8

pancace7.txt:

Output:

Lösungssequenz: [12, 3, 4, 10, 12, 10, 2, 4] Länge: 8

3.2 Aufgabe 3b

Ausgaben meines Programms zur Bestimmung der PWUE-Zahl bis n=11:

Für n=2 ist die PWUE-Zahl: 1, exemplarischer Stapel: [2, 1]

Für n=3 ist die PWUE-Zahl: 2, exemplarischer Stapel: [2, 3, 1]

Für n=4 ist die PWUE-Zahl: 2, exemplarischer Stapel: [1, 2, 4, 3]

Für n=5 ist die PWUE-Zahl: 3, exemplarischer Stapel: [1, 4, 2, 5, 3]

Für n=6 ist die PWUE-Zahl: 3, exemplarischer Stapel: [1, 2, 5, 3, 6, 4]

Für n=7 ist die PWUE-Zahl: 4, exemplarischer Stapel: [1, 2, 7, 4, 6, 5, 3]

Für n=8 ist die PWUE-Zahl: 5, exemplarischer Stapel: [4, 6, 1, 7, 2, 8, 5, 3]

Für n=9 ist die PWUE-Zahl: 5, exemplarischer Stapel: [1, 2, 7, 3, 8, 4, 9, 6, 5]

Für n=10 ist die PWUE-Zahl: 6, exemplarischer Stapel: [1, 4, 9, 3, 8, 2, 7, 10, 6, 5]

Für n=11 ist die PWUE-Zahl: 6, exemplarischer Stapel: [1, 2, 5, 10, 3, 8, 4, 9, 6, 11, 7]

4 Quellcode: Implementierung in Python

4.1 Aufgabe 3a

Nachfolgend ist das Hauptprogramm meiner Implementierung in Python 3.10:

```
def generate_Loesungssequenz(liste):
    """
    generate_Loesungssequenz generiert die Lösungssequenz für eine Liste an Pfannkuchen in Form von Indices,
    an welchen die PWUE-Funktion in der entsprechenden Reihenfolge angewendet wird.
    (Wrapper-Funktion für die rekursive, interne Funktion)
    """
    alpha = [len(liste)] #Intiierung des Abbruchfaktors alpha als maximal
    pWUESequence = generate_Loesungssequenz_intern(liste, [], alpha) #Aufruf der Rekursiven Funktion
    return len(pWUESequence), pWUESequence #Rückgabe der Lösung der internen Funktion

def generate_Loesungssequenz_intern(liste, actions, alpha):
    """
    generate_Loesungssequenz_intern probiert rekursiv alle möglichkeiten die PWUE-Funktion auf die, ihm übergebene,
    Liste an Pfannkuchen anzuwenden und dadurch zu sortieren.
    """
    #1. Überprüfe ob die Liste bereits sortiert ist und falls ja, gebe die durchgeführten actions zurück (Rekursionsanker)
    if len(liste) > 1 and not is_sorted(liste):
        #2. Generiere alle möglichen Listen durch Anwenden der PWUE-Funktion auf alle Indices der erhaltenen Liste
        shortest_action = None
        for n in range(0, len(liste)):
            #Optimierung 1: Breche das Suchen ab, falls bereits mehr actions durchgeführt wurden, als die bisher beste
            #Lösungssequenz
            #benötigt hat, da die momentanige Sequenz nicht besser sein kann.
            if len(actions)+1 < alpha[0]:
                new_actions = actions.copy()
                new_actions.append(n)
                #3. Rufe mit der neuen Liste und der aktualisierten Lösungssequenz wieder sich selber auf
                (Rekursionsschritt)
```

```

new_actions, alpha)

new_shortest_action_recursive = generate_Loesungssequenz_intern(pWUEFunc(liste, n),

if new_shortest_action_recursive == None:
    continue
#4. Bestimme die kürzeste Lösungssequenz und...
if shortest_action == None or len(shortest_action) > len(new_shortest_action_recursive):
    shortest_action = new_shortest_action_recursive
#Optimierung 2: Falls die neue Lösungssequenz kürzer ist als die bisher kürzeste,

aktualisiere die kürzeste.

if len(shortest_action) < alpha[0]:
    alpha[0] = len(shortest_action)

#4. ...gebe sie zurück.
return shortest_action
else:
    return actions

```

4.2 Aufgabe 3b

Nachfolgend ist das Hauptprogramm meiner Implementierung in Python 3.10:

```

def generiere_PWUEZahl_von_n(size):
    """
    generiere_PWUEZahl_von_n generiert die PWUE-Zahl von size durch die Nutzung der Längen der Lösungssequenzen
    der vorherigen Schicht (size-1).
    """
    all_stapel = generate_stapelcombinations(size)
    #1. Rekursives Generieren aller vorherigen Schichten bis n=1.
    if size == 1:
        return {0: 0}, 0, [1]
    previous_layer = generiere_PWUEZahl_von_n(size-1)[0]
    new_layer = {}
    #2a. Der einzig sortierte Stapel ist der Stapel des Formats S=[1,2,...,size], welcher von generate_stapelcombinations
    #als erster generiert wird:
    new_layer[hash_func(all_stapel[0])] = 0
    #2b. Für alle anderen Stapel...
    best_stapel = None
    for stapel in all_stapel[1:]:
        best_PWUEZahl = size
        #3a. generiere alle Stapel, die sich durch das einmalige Anwenden der PWUE-Funktion auf den Stapel ergeben
        for i in range(len(stapel)):
            #3b. und 'kürze' sie.
            new_stapel = reduce_stapel(pWUEFunc(stapel.copy(), i))
            #4a. Schlage die Länge der Lösungssequenz für die neuen Stapel in der vorherigen Schicht nach und
            #bestimme dir kürzeste eines jeden Stapels.
            if previous_layer[hash_func(new_stapel)]+1 < best_PWUEZahl:
                best_PWUEZahl = previous_layer[hash_func(new_stapel)]+1
        #Speicherung der PWUE-Zahl & des korrespondierenden Stapels
        if best_stapel == None or best_stapel[0] < best_PWUEZahl:
            best_stapel = [best_PWUEZahl, stapel]
        #4b. Speichere die kürzeste Länge einer Lösungssequenz (+1) für den Stapel im new_layer.
        new_layer[hash_func(stapel)] = best_PWUEZahl
    #Rückgabe des new_layer, der PWUE-Zahl und des 'unsortiertesten' Stapels.
    return new_layer, best_stapel[0], best_stapel[1]

```