Aufgabe 3: Pancake Sort

Teilnahme-Id: 65438

Bearbeiter/-in dieser Aufgabe:
Raphael Gaedtke

9. April 2023

Inhaltsverzeichnis

1 Lösungsidee

1.1 Eigenschaften von Pfannkuchenstapeln

Es werden zunächst einige allgemeine Definitionen und Beobachtungen zu Pfannkuchenstapeln vorgestellt, bevor die Berechnung der Funktionen A und P in den Kapiteln 1.2 und 1.3 beschrieben wird. Zunächst soll es möglich sein, einen gegebenen Pfannkuchenstapel nach einer Wendeoperation zu vereinfachen:

Definition 1. *Es sei S ein Stapel mit $k \in \mathbb{N}$ Pfannkuchen, von denen der größte die Größe $l \in \mathbb{N}$ habe, wobei $l \geq k$ gelten muss. Es sei dann E die Funktion, die S auf einen Stapel S' abbildet, der ein Stapel von k Pfannkuchen mit den Größen 1 bis k ist.*
Dieser Stapel S' sei der Pfannkuchenstapel, der entsteht, wenn der kleinste Pfannkuchen in S durch einen mit Größe 1, der zweitkleinste durch 2 usw. ersetzt wird. Der Pfannkuchen mit Größe l wird dann durch einen mit Größe k ersetzt. Außerdem sollen die größten Pfannkuchen, die schon korrekt einsortiert am unteren Ende des Stapels S liegen, aus dem Stapel gelöscht sein.

Beispiel 1. *Ist $S = (2, 4, 5, 1)$ und $T = (2, 4, 5, 1, 6)$, so ist $E(S) = E(T) = (2, 3, 4, 1)$.*

Lemma 1. *Liegt der größte Pfannkuchen eines Stapels ganz unten, muss er für eine optimale Sortierung nicht bewegt werden.*

Beweis. Gäbe es in einer optimalen Sequenz von PWUE-Operationen zum Sortieren des Stapels eine, die den untersten Pfannkuchen bewegt, so könnte diese stattdessen auch erst beim zweituntersten mit Größe z beginnen. Der unterste Pfannkuchen ist dann nämlich schon korrekt einsortiert.
Dadurch fehlt bei den folgenden Operationen lediglich der Pfannkuchen mit Größe z, der aufgegessen wird. Das Fehlen eines Pfannkuchens kann die Anzahl der nötigen PWUE-Operationen aber nicht erhöhen. □

Lemma 2. *Es gilt $A(S) = A(E(S))$ für jeden Pfannkuchenstapel S.*

Beweis. Für den Stapel S hängt der Wert von $A(S)$ von der Reihenfolge der Größen der Pfannkuchen in S ab. Diese Reihenfolge ist in S und $E(S)$ aber gleich, weshalb die zu beweisende Aussage gelten muss. Außerdem sind die vom unteren Ende des Stapels entfernten Pfannkuchen nach Lemma 1 irrelevant für die Berechnung.
Um $A(S)$ zu berechnen, kann also auch direkt $A(E(S))$ berechnet werden, was in der Regel eine Vereinfachung darstellt. □

Mithilfe dieser Vereinfachung von Pfannkuchenstapeln kann dann auch eine Funktion für die Wende- und Ess-Operationen auf einem Stapel definiert werden:

Definition 2. *Es sei S ein Pfannkuchenstapel und $k \in \mathbb{N}$. Dann sei $W(S, k)$ der Stapel, der entsteht, wenn die obersten k Pfannkuchen gewendet, der dann ganz oben liegende Pfannkuchen gegessen und auf den verbleibenden Stapel die Funktion E angewandt wird.*

Es besteht dabei ein Zusammenhang zwischen $A(S)$ und den für die Sortierung von S nach einer Wendeoperationen benötigten PWUE-Operationen:

Theorem 1. *Für jeden nicht vollständig sortierten Stapel S mit $n \in \mathbb{N}$ Pfannkuchen ist*

$$A(S) = 1 + \min_{k \in \mathbb{N}_{\leq n}} A(W(S, k)) \tag{1}$$

Beweis. Es sei zunächst $l \in \mathbb{N}$ mit

$$A(W(S, l)) = \min_{k \in \mathbb{N}_{\leq n}} A(W(S, k)) \tag{2}$$

und $P = W(S, l)$. Der Stapel P ist unter allen durch eine einzige PWUE-Operation von S aus erreichbaren Stapeln also derjenige, der am schnellsten sortierbar ist.
Dann muss zunächst $A(S) \leq A(P) + 1$ sein, schließlich lässt sich S durch eine Operation in P überführen und dann in weiteren $A(P)$ Operationen vollständig sortieren. Es muss aber auch $A(S) \geq A(P) + 1$ sein - schließlich muss es zum Sortieren von S eine Folge von $A(S)$ PWUE-Operationen geben. Die erste dieser Operationen führt aber zu einem Stapel S' mit $\exists m \in \mathbb{N} : W(S, m) = S'$ und $A(S') = A(S) - 1$. Es kann also ein Stapel P mit $P = S'$ gewählt werden.
Aus $A(S) \leq A(P) + 1$ und $A(S) \geq A(P) + 1$ folgt aber $A(S) = A(P) + 1$. □

Außerdem lässt sich die Funktion P für konkrete Stapelgrößen nach oben abschätzen:

Theorem 2. *Es ist $\forall n \in \mathbb{N} : P(n) \leq 2 \cdot \lceil \frac{n}{3} \rceil$.*

Beweis. Die Aussage wird bewiesen durch Angabe eines Algorithmus, der jeden beliebigen Stapel mit n Pfannkuchen in maximal $2 \cdot \lceil \frac{n}{3} \rceil$ Schritten sortiert.
Dafür werden immer zwei Operationen durchgeführt, bis der Stapel sortiert ist. Betrachtet wird nur der obere Teil des Pfannkuchenstapels, der noch nicht sortiert ist (liegen die größten Pfannkuchen in der richtigen Reihenfolge ganz unten, müssen sie nach Lemma 1 nicht mehr beachtet werden).

1. Liegt der größte noch nicht richtig einsortierte Pfannkuchen an Position k (von oben) im Stapel S, so wird S zunächst auf $W(S, k+1)$ gesetzt. Danach liegt dieser größte Pfannkuchen ganz oben (liegt er vorher schon ganz oben, kann dieser Schritt übersprungen werden).
2. Muss der jetzt oben liegende Pfannkuchen an Position l einsortiert werden, wird S auf $W(S, l)$ gesetzt.

Dieser Algorithmus korrigiert mit zwei Operationen die Positionen von drei Pfannkuchen: Der größte noch nicht korrekt einsortierte, der danach an der richtigen Position liegt, und zwei weitere, die während dieser Operation gegessen werden (Wird der erste Schritt übersprungen, sind es zwei Pfannkuchen in einem Schritt).
Bei zwei Operationen für drei Pfannkuchen kann der Algorithmus aber maximal $2 \cdot \lceil \frac{n}{3} \rceil$ Schritte benötigen. □

Beispiel 2. *Der Algorithmus soll am Beispiel von $S = (3, 2, 4, 5, 1)$ illustriert werden:*

1. *Zu Beginn ist 5 der größte Pfannkuchen und liegt an vierter Position. Der Stapel S wird also durch eine Wende-und-Ess-Operation zu $S' = W(S, 5) = (4, 3, 1, 2)$.*
2. *Der nun oben liegende, größte Pfannkuchen wird an die unterste Position verschoben, also ist $S'' = W(S', 4) = (1, 2, 3)$.*
3. *Der Stapel S'' ist dann nach zwei Operationen sortiert. Es ist aber $2 \leq 2 \cdot \lceil \frac{5}{3} \rceil = 4$.*

Theorem 3. *Es ist $\forall n \in \mathbb{N}_{>1} : P(n-1) \leq P(n) \leq P(n-1) + 1$.*

Beweis. Es kann nicht $P(n) < P(n-1)$ sein - andernfalls könnte ein Stapel S mit $(n-1)$ Pfannkuchen und $A(S) = P(n-1)$ durch Hinzufügen eines Pfannkuchens der Größe n am unteren Ende schneller sortiert werden.
Es kann aber auch nicht $P(n) > (P(n-1)+1)$ sein, schließlich lässt sich jeder Stapel mit n Pfannkuchen durch eine einzige PWUE-Operation in einen Stapel mit $(n-1)$ Pfannkuchen überführen. Dieser lässt sich aber schlechtestenfalls in $P(n-1)$ Schritten sortieren.
Insgesamt folgt $P(n) \geq P(n-1)$ und $P(n) \leq P(n-1) + 1$. □

1.2 Aufgabenteil a: Berechnung der Funktion A

Für einen gegebenen Stapel S wird die Anzahl der für seine Sortierung benötigten PWUE-Operationen $A(S)$ durch eine optimierte Breitensuche -also durch einen Complete-Search-Ansatz- ermittelt. Zentral hierfür ist das Theorem 1.

1.2.1 Berechnung

Es wird eine Warteschlange Q verwaltet, die zu Beginn nur den gegebenen Stapel S enthält. Nach dem i-ten Schritt soll Q alle Stapel enthalten, die sich durch genau i (aber nicht weniger) PWUE-Operation von S aus errreichen lassen.
Dazu werden im i-ten Schritt zunächst alle Stapel aus der Warteschlange genommen. Für jeden solchen Stapel P mit $k \in \mathbb{N}$ Pfannkuchen werden alle Stapel $P' = W(P, l)$ mit $l \in \mathbb{N}_{\leq k}$ erzeugt und in die Warteschlange eingefügt. So wird gleichzeitig sichergestellt, dass auch alle beabsichtigten Stapel in der Warteschlange enthalten sind.
Dies wird durchgeführt, bis ein sortierter Stapel gefunden wird. Geschieht dies im j-ten Schritt, so ist $A(S) = j$. Ob ein Stapel sortiert ist, wird dabei implizit während der Berechnung der Funktion E ermittelt, die bei Anwendung auf einen sortierten einen leeren Pfannkuchenstapel zurückliefert.

Da die Funktion E hier nach jeder einzelnen Wendeoperation angewandt wird, muss sie lediglich für Stapel mit k Pfannkuchen mit unterschiedlichen Größen zwischen 1 und $(k+1)$ berechnet werden. In diesen Fällen fehlt also jeweils genau ein Pfannkuchen.
Durch einmaliges Iterieren über den gesamten Stapel kann die Summe aller enthaltenen Pfannkuchengrößen berechnet werden. Die Differenz zwischen dieser Summe und der Summe aller Zahlen von 1 bis $(k+1)$, also $\sum_{i=1}^{k+1} i = \frac{i(i+1)}{2}$, ist dann genau die Größe des fehlenden Pfannkuchens. Alle Pfannkuchen, die dann größer sind als dieser, werden bei einem zweiten Durchgehen des Stapels um 1 dekrementiert - es verbleibt ein Stapel mit k Pfannkuchen mit unterschiedlichen Größen von 1 bis k in der richtigen Reihenfolge.

Anschließend kann der unterste Pfannkuchen entfernt werden, solange er der größte ist (in diesem Fall ist seine Größe auch immer die Größe des Stapels). Verbleibt nach dieser Prozedur ein leerer Stapel, so ist der Stapel sortiert. In jedem anderen Fall wird der verbleibende Stapel in die Warteschlange eingefügt (Das Anwenden der Funktion E ist ja nötig, um die Definition von W umzusetzen).

Die praktische Laufzeit dieses Verfahrens kann durch Optimierung aber noch stark verbessert werden. Problematisch im oben beschriebenen Vorgehen ist schon die Tatsache, dass der gleiche Stapel mehrmals berechnet werden kann. Dies geschieht, wenn zwei verschiedene Stapel in einem Schritt durch das Anwenden der PWUE-Operation in den gleichen Stapel überführt werden können. Dies wird umso bedeutsamer, weil die Funktion E nominell unterschiedliche Stapel wie etwa $(5, 4, 3)$ und $(3, 2, 1)$ gleichsetzt, was auch der eigentliche Grund für die Verwendung dieser Funktion ist.

Beispiel 3. *Ist* $S = (2, 1, 5, 4, 3)$ *und* $P = (1, 2, 5, 4, 3)$, *so ist*

$$W(S, 2) = W(P, 2) = (1, 4, 3, 2). \tag{3}$$

Die Auswirkungen dieser Mehrfachberechnungen bedeuten einen starke Laufzeitsteigerung, weil auch alle von den mehrfach eingefügten Stapeln aus erreichbaren mehrfach berechnet werden usw. Dieses Problem wird gelöst, indem eine Hashtabelle H von allen Stapeln, die zu irgendeinem Zeitpunkt in Q eingefügt wurden, verwaltet wird.
Soll dann ein neuer Stapel eingefügt werden, wird zunächst überprüft, ob er schon in H steht. Ist dies der Fall, wird er verworfen, was die Mehrfachberechnung vermeidet. In jedem anderen Fall wird er in H und in Q eingefügt.

1.2.2 Rekonstruktion der Wendeoperationen

Für einen gegebenen Stapel S soll allerdings nicht nur der Wert der Funktion $A(S)$, sondern auch eine Liste von $A(S)$ PWUE-Operationen, die S sortieren, ausgegeben werden. Zu diesem Zweck wird die Breitensuche über die Menge der Stapel mit Backtracking kombiniert.
Konkret wird für jeden Stapel S, der in Q eingefügt wird, in einer Hashmap B gespeichert, von welchem Stapel aus dieser durch eine PWUE-Operation erreicht werden kann. Konkret bedeutet dies für jeden jemals in Q eingefügten Stapel S:

$$\exists k \in \mathbb{N} : W(B[S], k) = S. \tag{4}$$

Wurde nun ein sortierter Stapel gefunden, so kann eine Liste der in der optimalen Sequenz von PWUE-Operationen erreichten Stapel rekonstruiert werden:
Es sei dazu der sortierte Stapel der Stapel T. An die gesuchte Liste von Stapel wird dann zunächst T angehängt und anschließend T auf $B[T]$ gesetzt. Dies wird durchgeführt, bis der ursprünglich vom Nutzer vorgegebene Stapel angehängt wurde. Anschließend wir die Reihenfolge in der neuen Liste von Stapeln noch umgekehrt.
Aus dieser Liste können dann die nötigen PWUE-Operationen ermittelt werden. Dazu wird für jeden Stapel in der Liste ausprobiert, welche der möglichen Wendeoperationen zum darauffolgenden Stapel führt. Beachtet werden muss hierbei noch die Tatsache, dass die Liste nicht originale Stapel, sondern die durch sukzessive Anwendung von E vereinfachten Stapel enthält.

1.2.3 Laufzeitüberlegungen

Zu Beginn wird genau ein Stapel mit n Pfannkuchen in Q eingefügt. Für diesen Stapel können durch Wendeoperationen bis zu n verschiedene Stapel mit jeweils bis zu $(n-1)$ Pfannkuchen erreicht werden. Diese müssen ebenfalls alle in Q eingefügt werden.
Für jeden dieser Pfannkuchen können jeweils bis zu $(n-1)$ mit bis zu $(n-2)$ Pfannkuchen usw. entstehen. Insgesamt müssen im i-ten Schritt im schlechtesten Fall $(n-i+1)$-mal mehr Pfannkuchen eingefügt werden als im $(i-1)$-ten Schritt, insgesamt also etwa $n!$ viele.
Für jeden so eingefügten Pfannkuchen müssen dann alle möglichen Wendeoperationen durchgeführt werden. Anzahl und Laufzeit dieser Wendeoperationen ist dabei jeweils proportional zur Größe des Stapels, die Laufzeit aller Wendeoperationen also insgesamt quadratisch in Bezug auf diese Größe.
Es wird dabei davon ausgegangen, dass die verwendeten Operationen (also Einfügen, Herausholen und Nachschlagen) der Warteschlange Q, der Hashtabelle H und der Hashmap B in konstanter Zeit durchgeführt werden können. Dann liegt die Laufzeit des gesamten Verfahrens etwa in $O(n^2 \cdot n!)$.

Dies ist allerdings nur eine äußerst ungenaue obere Schranke, die die durch die Anwendung der Funktion E und das Verwenden der Hashtabelle H erzielten Verbesserungen vernachlässigt. In der Realität führen nämlich viele Wendeoperationen unterschiedlicher Stapel (nach dem Anwenden der Funktion E) auf die gleichen Stapel, die dann nicht mehrfach berechnet werden.
Außerdem müssen nach Theorem 2 nicht n, sondern schlechtestenfalls $2 \cdot \lceil \frac{n}{3} \rceil$ Schritte durchgeführt werden.
Bei den gegebenen Eingabegrößen von $n \leq 16$ führen die genannten Einschränkungen und Optimierungen also zu akzeptablen Laufzeiten.

1.3 Aufgabenteil b: Berechnung der Funktion P

Definition 3. *Es sei $M_{k,l}$ die Menge aller Pfannkuchenstapel S mit k Pfannkuchen mit den Größen 1 bis k, für die $A(S) = l$ gilt. Die Menge $M_{k,l}$ soll aber nicht direkt diese Stapel, sondern ihre Bilder unter der Funktion E enthalten.*

Beispiel 4. *Es ist*

$$M_{2,0} = \{E((1,2))\} = \{()\}, \tag{5}$$

$$M_{2,1} = \{E((2,1))\} = \{(2,1)\} \tag{6}$$

und

$$M_{2,2} = \emptyset \tag{7}$$

Dabei ist

$$\forall n \in \mathbb{N} : M_{n,0} = \{()\} \tag{8}$$

-diese Mengen enthalten also jeweils einen sortierten Stapel- und

$$\forall n \in \mathbb{N}, k \in \mathbb{N}_{>P(n)} : M_{n,k} = \emptyset. \tag{9}$$

1.3.1 Berechnung

Es wird zunächst angegeben, wie die Menge $M_{k,l}$ aus den Mengen $M_{(k-1),y}$ mit $y \geq (l-1)$ berechnet werden kann. Anschließend wird aufgezeigt, wie es möglich ist, mithilfe dieser Operation $P(n)$ zu ermitteln und ein Beispiel für einen entsprechenden Stapel auszugeben.

Dafür seien zunächst diese Mengen $M_{(k-1),y}$ gegeben. Das Anwenden einer beliebigen PWUE-Operation auf einen beliebigen Stapel in $M_{k,l}$ muss einen Stapel in einer dieser Mengen ergeben, und für jeden Stapel in $M_{k,l}$ muss es mindestens eine PWUE-Operation geben, die ihn auf einen Stapel in $M_{(k-1),(l-1)}$ überführt.
Wäre die erste Bedingung für einen Stapel $S \in M_{k,l}$ nicht erfüllt, so müsste der aus einer Operation resultierende Stapel P in einer Menge $M_{(k-1),j}$ mit $j < (l-1)$ liegen. Dann wäre aber auch $A(S) \leq j+1 < l$ und somit $S \notin M_{k,l}$ (Dabei wird hier und im Folgenden der Einfachheit halber $S \in M$ geschrieben, wenn eigentlich $E(S) \in M$ gemeint ist). Wäre die zweite Bedingung für S nicht erfüllt, so wäre $A(S) > (l-1)+1 = l$ und somit wiederum $S \notin M_{k,l}$.

Deswegen kann $M_{k,l}$ ermittelt werden, indem zunächst alle Pfannkuchenstapel T in $M_{(k-1),(l-1)}$ durchgegangen werden. Für jeden dieser Stapel werden alle Stapel mit k Pfannkuchen betrachtet, die durch eine PWUE-Operation in T übergehen können. Dazu wird jeweils die Reihenfolge der ersten i Pfannkuchen in T umgekehrt und an der $(i+1)$-ten Stelle ein Pfannkuchen der Größe j eingefügt. Anschließend werden alle Pfannkuchen, die vorher eine Größe $\geq j$ hatten, um 1 inkrementiert - es wird also die „Umkehrung“ einer PWUE-Operation durchgeführt. Auf diese Weise werden also alle Pfannkuchenstapel durchgegangen, für die die zweite Bedingung gilt.
Für den aus einer solchen Umkehrung resultierenden Stapel werden dann alle für ihn möglichen PWUE-Operationen ausprobiert und getestet, ob ein jeder der daraus resultierenden Stapel in einer der vorgegebenen Mengen liegt. Ist dies der Fall, so liegt der getestete Stapel in $M_{k,l}$, mit dieser Methode wird aber auch jeder Stapel in $M_{k,l}$ gefunden.

Es ist also bekannt, wie $M_{k,l}$ aus den entsprechenden $M_{(k-1),y}$ berechnet werden kann. Außerdem sind nach Gleichung 8 und Gleichung 9 alle Mengen $M_{k,l}$ mit $l > P(k)$ - diese sind leer - und alle Mengen mit $M_{n,0}$ bekannt - diese enthalten jeweils einen sortierten Stapel. Dies kann nun aber benutzt werden, um $P(k)$ für alle $k \leq n$ zu berechnen.

Dazu wird zunächst $P(1) = 0$ gesetzt, was nach Aufgabenstellung wahr sein muss. Dann werden die $P(k)$ in aufsteigender Reihenfolge durchgegangen:
Aus den Mengen $M_{k-1,y}$ wird zunächst $M_{k,P(k-1)+1}$ berechnet. Ist $M_{k,P(k-1)+1} \neq \emptyset$, so ist nach Theorem 3 immer $P(k) = P(k-1)+1$. Außerdem liegt $M_{k,P(k)}$ wegen $M_{k,P(k)} = M_{k,P(k-1)+1}$ schon vor. In jedem anderen Fall ist nach Theorem 3 sicher $P(k) = P(k-1)$. Die Menge $M_{k,P(k)}$ muss dann für die weiteren Rechnungen ermittelt werden - es ist aber nicht gesichtert, dass die dafür nötige Menge $M_{(k-2),P(k-1)-2}$ schon berechnet wurde.
Nötigenfalls wird diese berechnet, wofür möglicherweise auch $M_{(k-2),P(k-1)-2}$ neu bestimmt werden muss. Dafür benötigt das Programm möglicherweise auch $M_{(k-3),P(k-1)-3}$ als neue Menge usw. Die Berechnung läuft also rekursiv weiter, bis nur noch schon bekannte Basismengen benötigt werden.

Um diese Berechnungen laufzeitsparend durchführen zu können, werden die Mengen $M_{k,l}$ in zwei Containern gespeichert: Einerseits in einer Listenstruktur, um möglichst schnell über alle Elemente einer spezifischen dieser Mengen iterieren zu können, andererseits in einer Hashmap, die für jeden bereits berechneten Stapel seine minimale Anzahl an zum Sortieren benötigten PWUE-Operationen speichert (und mit der daher die Zugehörigkeit eines Stapels mit k Pfannkuchen zu einer bestimmten Menge $M_{k,l}$ überprüft werden kann).
Um für jede berechnete Zahl $P(k)$ einen Stapel mit k Pfannkuchen auszugeben, für den tatsächlich genau $P(k)$ PWUE-Operationen zum Sortieren nötig sind, kann dann ein beliebiges Element der Menge $M_{k,P(k)}$ ausgewählt werden (wobei beachtet werden muss, dass diese Menge Bilder der Funktion E enthält).

1.3.2 Optimierung

Ist bei der Berechnung der Werte $P(1)$ bis $P(n)$ wegen $k = n$ die Zahl $P(k)$ der letzte zu berechnende Funktionswert und ist $P(k) = P(k-1)$, so wird durch das oben beschrieben Verfahren noch die Menge $M_{k,P(k)}$ berechnet. Dies ist aber eigentlich nicht nötig, weil diese Menge nur zum Berechnen weiterer Werte $P(j)$ mit $j > k = n$ benötigt werden würde.
Die Laufzeit wird auf diese Weise signifikant erhöht, weil die Mengen $M_{k,l}$ für steigende Werte von k stark wachsen können und außerdem weitere solcher Mengen für kleinere k als Basis des rekursiven Vorgehens berechnet werden müssen.

Um dies zu umgehen, wird die Berechnung der Menge $M_{k,P(k)}$ in diesem Fall nicht durchgeführt. Die Angabe eines beispielhaften Stapels S mit $A(S) = P(k)$ ist wegen $P(k) = P(k-1)$ auch trivial: Ein solcher wird erreicht durch einen Stapel aus der Menge $M_{k-1,P(k-1)}$, bei dem ein Pfannkuchen mit Größe k ganz unten eingefügt wurde.
Durch das Einfügen des neuen Pfannkuchens kann der Stapel dabei nicht schneller sortiert werden, also wurde so ein Stapel mit der richtigen Anzahl an nötigen Sortieroperationen gefunden.

1.3.3 Laufzeitüberlegungen

Bei dieser Teilaufgabe müssen im schlimmsten Falle alle $k!$ möglichen Stapel der Größe k in den Mengen $M_{k,l}$ verteilt gespeichert werden.
Für jeden dieser Stapel müssen alle „Umkehrungen“ von PWUE-Operationen und für jeden resultierenden Stapel alle PWUE-Operationen getestet werden. Beides sind für einen Stapel der Größe k etwa k Stück, deren Laufzeit etwa proportional zur Größe dieses Stapels ist. Es kann dabei davon ausgegangen werden, dass das Einfügen eines Stapels in eine Menge und die Abfrage, ob ein Stapel in einer bestimmten Hashtabelle enthalten ist, in konstanter Zeit ablaufen.
Insgesamt ergibt sich so eine Laufzeitkomplexität in $O(n^3 \cdot n!)$, was aber nur eine ungenaue obere Schranke ist - die Aufteilung der möglichen Stapel hat schließlich den Sinn, nicht alle mögichen Stapel berechnen zu müssen. In der Praxis ist die Anzahl der zu berechnenden Stapel daher deutlich niedriger.

1.4 Erweiterung: Andere Operationen

Das vorgegebene Problem beschäftigt sich mit PWUE-Operationen, also eine Wende-Operation gefolgt von einer Ess-Operation. Dabei sind andere „Zusammenstellungen“ dieser Operationen ebenfalls interessant.

1.4.1 PW-Operationen

PW-Operationen sind Operationen, in denen nur die obersten k für eine positive ganze Zahl k Pfannkuchen gewendet werden. Das Sortieren eines Pfannkuchenstapels mithilfe dieser Operation hat den Namen „Pancake Sorting“ (Übereinstimmungen mit dem Titel dieser Aufgabe sind sicherlich reiner Zufall) und wurden in der Literatur schon häufiger betrachtet.
Beispielhaft dafür ist ein Eintrag in der „On-line encyclopedia of integer sequences“, in dem die Folge der Zahlen $P_w(i)$ mit $i \in \mathbb{N}$ angegeben ist, wobei $P_w(i)$ die größtmögliche Anzahl von PW-Operationen bezeichne, die bei optimalem Vorgehen benötigt werden kann, um einen Stapel mit i Pfannkuchen zu sortieren.[1]
Diese beginnt laut OEIS mit

i	1	2	3	4	5	6	7	8
$P_w(i)$	0	1	3	4	5	7	8	9

Da dieses Problem also schon untersucht und diese Untersuchungen publiziert wurden, wird es hier nicht näher betrachtet.

1.4.2 PEUW-Operationen

Im Vergleich zu den PWUE-Operationen kann die Reihenfolge der beiden Elemente (Wenden und Essen) auch umgekehrt werden - die resultierenden Operationen sind die PEUW-Operationen, bei denen zuerst der oberste Pfannkuchen gegessen und dann eine Wendeoperation umgesetzt wird.
Analog zur PWUE-Operation sei $A_R(S)$ für einen Pfannkuchenstapel S die kleinstmögliche Anzahl von PEUW-Operationen, mit denen der Stapel sortiert werden kann. Außerdem sei $P_R(i)$ der größtmögliche Wert von $A_R(S)$, wenn S genau i Pfannkuchen enthält.

Die Berechnung von A_R und P_R folgt aber mit nur geringen Modifikationen aus der Berechnung der Funktionen A und P: Neben der offensichtlich nötigen Vertauschung der beiden Operationen muss nämlich keine Änderung am Programm vorgenommen werden. Damit bleibt auch z.B. die Laufzeitkomplexität des resultierenden Algorithmus gleich.
Der Grund hierfür ist die Tatsache, dass PWUE- und PEUW-Operationen alle für die Lösung relevanten Eigenschaften gemeinsam haben. Insbesondere sinkt die Anzahl der Pfannkuchen in einem Stapel durch beide Operationen um jeweils 1, während$P(1) = P_R(1) = 0$ gilt.
Es lässt sich auch zeigen, dass $P(i)$ und $P_R(i)$ immer nahe beieinander liegen:

Theorem 4. *Für jede natürliche Zahl i gilt $P_R(i) \leq P(i) + 1$ und $P(i) \leq P_R(i) + 1$.*

Beweis. Es wird nur $P_R(i) \leq P(i) + 1$ gezeigt, die andere Aussage folgt analog. Wird von einem beliebigen Stapel mit i Pfannkuchen der oberste Pfannkuchen gegessen, so entsteht ein Pfannkuchenstapel mit $(i-1)$ Pfannkuchen.
Dieser kann mit maximal $P(i-1)$ PWUE-Operationen sortiert werden. Wird am Ende dann noch eine Wendeoperation hinzugefügt, die nur den obersten Pfannkuchen wendet (und den Stapel damit unverändert lässt), so liegen zusammen mit den $P(i-1)$ PWUE-Operationen und dem anfänglichen Essen des Pfannkuchens insgesamt $2 \cdot P(i-1) + 2 = 2(P(i-1)+1)$ Ess- oder Wendeoperationen vor.
Die erste Operation ist das Essen eines Pfannkuchens, außerdem folgen Ess- und Wendeoperationen immer abwechselnd - die Kette von Operationen lässt sich also durch $(P(i-1)+1)$ PEUW-Operationen darstellen, die den Stapel sortieren.
Also ist $P_R(i) \leq P(i-1) + 1$. Wie im Beweis für Theorem 3 begründet, ist aber $P(i) \geq P(i-1)$ und damit $P_R(i) \leq P(i) + 1$. □

1.4.3 PE-Operationen

Bei PE-Operationen wird immer nur der oberste Pfannkuchen vom Stapel gegessen. Analog zu den vorherigen Kapiteln seien für einen Stapel S dann $A_E(S)$ die kleinstmögliche Zahl von PE-Operationen, nach denen S sortiert ist und $P_E(i)$ für eine positive Zahl i die größte Zahl, die $A_E(S)$ für einen Stapel S mit i Elementen annehmen kann.
Die Funktion P_E wird hier nicht in einem Programm umgesetzt, weil sie sich durch eine explizite, geschlossene Formel berechnen lässt:

[1] https://oeis.org/A058986, zuletzt abgerufen am 06.04.2023

Theorem 5. *Es gilt* $\forall i \in \mathbb{N} : P_E(i) = i - 1$.

Beweis. Es ist zunächst immer $P_E(i) \leq i - 1$, was durch Angabe eines entsprechenden Stapels gezeigt wird. Ein solcher ist beispielsweise ein vollständig sortierter Stapel, bei dem der kleinste Pfannkuchen von der obersten in die unterste Position verschoben wurde. Solange weniger als $(i - 1)$ PE-Operationen auf den Stapel angewandt wurden, liegt dieser kleinste Pfannkuchen unter einem anderen, womit der Stapel nicht korrekt sortiert ist.
Gleichzeitig muss aber auch $i - 1 \leq P_E(i)$ sein. Nach $(i - 1)$ PE-Operationen enthält der Stapel nämlich nur noch einen einzigen Pfannkuchen und ist somit sicher sortiert.
Aus $P_E(i) \leq i - 1$ und $i - 1 \leq P_E(i)$ folgt aber genau $P_E(i) = i - 1$. □

In ein Programm umgesetzt wird hier also nur die Berechnung der Funktion A_E. Wichtig hierfür ist die (implizit im vorherigen Beweis schon verwendete) Beobachtung, dass eine Folge von k PE-Operationen die obersten k Pfannkuchen aus dem Stapel entfernt.
Für einen Stapel mit i Pfannkuchen verbleiben also immer die untersten $(i - k)$ Pfannkuchen. Gesucht ist also die kleinste Zahl k, für die die untersten $(i - k)$ Pfannkuchen korrekt sortiert sind. Dies lässt sich aber untersuchen, indem vom unteren Ende des Stapels ausgegangen wird: Solange der betrachtete Pfannkuchen (der oberste ist oder) größer ist als der zuletzt gefundene, ist der unterste Teil des Stapels sortiert und es kann der um 1 weiter oben liegende Pfannkuchen untersucht werden.
Sobald ein Pfannkuchen gefunden wurde, der kleiner ist als der vorherige (oder festgestellt wurde, dass der gesamte Stapel sortiert ist), wurden die $i - k$ Pfannkuchen und damit auch die kleinstmögliche Zahl k bestimmt. Das Programm kann abgebrochen werden, was somit in einer Laufzeit geschieht, die proportional zur Anzahl i der Pfannkuchen im Stapel ist (Konkret liegt die Laufzeitkomplexität in $O(i)$).

2 Umsetzung

Die Lösungsidee wird in C++ umgesetzt.

2.1 Umsetzung der Pfannkuchenstapel

Pfannkuchenstapel werden im Programm generell in Containern vom Typ `vector<short>` gespeichert. Dabei sollen Einträge dieses Vektors mit niedrigeren Indizes die Größen von weiter oben liegenden Pfannkuchen in diesem Stapel darstellen. In einer separaten Header-Datei werden dann Operationen auf diesen Stapeln umgesetzt:

Zunächst gibt es zwei Funktionen, die die Funktion E zur Vereinfachung von Pfannkuchenstapeln darstellen. Die erste hat die Bezeichnung `simplify()`, nimmt eine Referenz auf den zu vereinfachenden Stapel entgegen und setzt dann das in Kapitel 1.2.1 beschriebene Verfahren um. Insbesondere wird dabei angenommen, dass es sich bei den übergebenen Stapeln um solche handelt, in denen für eine vollständige Permutation nur ein einziger Pfannkuchen fehlt.
Bei der Ausgabe in den folgenden Programmteilen ist es bisweilen aber nötig, einen Stapel ohne diese Einschränkung zu vereinfachen. Hierfür wird die Funktion `outputSimplify()` verwendet. Diese kopiert zuerst den übergebenen Stapel in den Container `pancopy`, sortiert diesen und geht dann die so sortieren Pfannkuchen durch.
Der i-te dabei gefundene Pfannkuchen muss später auf die Größe i reduziert werden, was in einer Hashmap `dict` gespeichert wird. Anschließend wird der originale, per Referenz übergebene Stapel durchgegangen und jeder Pfannkuchen durch den Wert seiner Ausgangsgröße in `dict` ersetzt.
Während der Berechnungen wird aber ausschließlich die Funktion `simplify()` verwendet. Dies führt einerseits zu korrekten Ergebnissen, da sie nach jeder PWUE-Operation angewandt wird und daher auch nie mehr als ein Pfannkuchen fehlt, andererseits hat sie eine bessere Laufzeit als die Funktion `outputSimplify()`, die das Array nicht nur durchgehen, sondern auch kopieren, sortieren und eine gesamte Hashmap anlegen muss.

Die PWUE-Operation $W(S, k)$ und ihre „Umkehrung“ werden in den Funktionen `flipOperation()` und `insertCake()` umgesetzt. Diese kehren zunächst die Reihenfolge der ersten Pfannkuchen um - die genaue Anzahl der davon betroffenen Pfannkuchen wird vom Nutzer vorgegeben.
Für `insertCake()` werden außerdem Größe und Position eines neu einzufügenden Pfannkuchens als Argumente übergeben. Alle Pfannkuchen, die vorher nicht kleiner als diese Größe waren, werden in einer `for`-Schleife um 1 erhöht, bevor der entsprechende Pfannkuchen an der richtigen Position eingefügt wird.

In `flipOperation()` wird lediglich der erste Pfannkuchen entfernt und dann die Funktion `simplify()` angewandt.

Im Programm werden Pfannkuchenstapel häufig in Hashtabellen eingefügt oder als Key-Werte von Hashmaps verwendet. C++ erlaubt für diese Zwecke bei den entsprechenden Datenstrukturen in der STL (wie zum Beispiel `unordered_map`) keine Key-Elemente des Typs `vector<short>`. Deshalb, und um die Laufzeit zu verbessern, sollen die Pfannkuchenstapel auf IDs abgebildet werden.
Die ID eines solchen Stapels muss dabei zwei wichtige Eigenschaften erfüllen: Die ID eines Stapels und der Stapel zu einer gegebenen ID müssen eindeutig und effizient berechenbar sein. Formal muss eine injektive Abbildung zwischen der Menge der möglichen Stapel und der Menge der möglichen IDs verwendet werden.
Im vorliegenden Fall wird als ID eines Stapels ein Paar aus zwei Zahlen verwendet: Die Anzahl der Pfannkuchen im Stapel und die Nummer dieses Stapels, wenn alle möglichen Stapel dieser Größe in lexikografisch aufsteigender Ordnung sortiert und 0-indiziert durchnummeriert werden. Dabei werden nur Stapel betrachtet, die aus k unterschiedlichen Pfannkuchen mit den Größen von 1 bis k bestehen (also „vollständige“ Stapel).

Beispiel 5. *Die lexikografisch aufsteigende Ordnung aller Stapel mit drei Pfannkuchen ist:*

1. $(1, 2, 3)$
2. $(1, 3, 2)$
3. $(2, 1, 3)$
4. $(2, 3, 1)$
5. $(3, 1, 2)$
6. $(3, 2, 1)$

Entsprechend ist die ID von $(1, 2, 3)$ *ein Paar aus den Zahlen 3 und 0 und die von* $(2, 3, 1)$ *ein Paar aus den Zahlen 3 und 3.*

Die ID zu einem gegebenen Stapel wird dann in der Funktion `stackIndex()`, der Stapel zu einer gegebenen ID in der Funktion `explicitStack()` berechnet. Dabei ist die Anzahl der Pfannkuchen im Stapel schon bekannt und nur die Position des Stapels in der lexikografischen Sortierung aller Stapel mit dieser Anzahl an Pfannkuchen muss bestimmt werden.
Wichtig dafür ist die Beobachtung, dass es mit k Pfannkuchen genau $(i - k)!$ Stapel gibt, die einen gegebenen Präfix der Länge i haben. Nach Vorberechnung aller Fakultäten entsprechender Größe kann also in linearer Laufzeit durch den Stapel iteriert werden. Aufaddieren von entsprechenden Fakultäten ergibt dann die ID, ganzzahliges Teilen der ID durch die Fakultäten die Elemente des Stapels.

2.2 Aufgabenteil a: Berechnung der Funktion A

Die Funktion A wird in der Funktion `evalA()` berechnet. Diese liest zunächst den zu sortierenden Stapel ein und verwaltet dann die Warteschlange Q als Container des Typs `queue<pair<int, long long>>`, also als Warteschlange der jeweiligen IDs.
Die Breitensuche wird in einer `while`-Schleife durchgeführt, bis eine Lösung ermittelt wurde. Dafür werden in jedem Schritt in einer `for`-Schleife alle vor diesem Schritt in Q vorhandenen Stapel aus Q genommen und gemäß der Lösungsidee alle durch PWUE-Operationen erreichbaren, noch nicht betrachteten Stapel in Q eingefügt.
Die `while`-Schleife wird abgebrochen, sobald ein sortierter Stapel in Q eingefügt werden müsste. Von diesem Punkt an wird die Ausgabe in der Funktion `outputPWUESequence()` ermittelt. Dazu wird die oben beschriebene Hashmap B verwendet, die hier auch die beabsichtigte Funktionalität von H abdeckt und als Container des Typs `unordered_map<long long, pair<int, long long>>` verwaltet wird. In der Funktion für die Ausgabe werden dann in einer `while`-Schleife die Vorgänger des jeweiligen Stapels durchgegangen, bis der Ausgangsstapel gefunden wurde.

2.3 Aufgabenteil b: Berechnung der Funktion P

Um die Berechnung der Funktion P umzusetzen, werden in der Funktion `evalP()` zunächst ein Container des Typs `vector<vector<vector<vector<short>>>>` und ein Container bzw. eine Hashmap des Typs `vector<unordered_map<long long, short>>` angelegt, um die zu berechnenden Mengen $M_{k,l}$ zu verwalten. Diese werden zu Beginn nur mit dem trivialen Stapel mit Größe 1 initialisiert.
Außerdem wird die Funktion `derive()` implementiert, die die Berechnung einer Menge $M_{k,l}$ aus der Menge $M_{(k-1),(l-1)}$ umsetzt. Hierfür wird in ineinander verschachtelten `for`-Schleifen über jeden Stapel in $M_{(k-1),(l-1)}$ und für jeden dieser Stapel über jede mögliche, „umgekehrte“ PWUE-Operation iteriert. In einer weiteren `for`-Schleife werden die resultierenden Stapel durch PWUE-Operationen getestet und der neue Stapel gegebenenfalls in $M_{k,l}$ eingefügt.
Die darauf aufbauende, rekursive Berechnung der Mengen $M_{k,l}$ wird dann in der Funktion `compute()` durchgeführt. Diese ruft sich nötigenfalls selbst auf, um weitere Mengen $M_{i,j}$ zu berechnen. Die Funktion `compute()` muss schlussendlich nur noch für jede zu berechnende Zahl $P(k)$ aufgerufen werden, was in einer einfachen `for`-Schleife geschieht.

2.4 Erweiterung: Andere Operationen

Für PEUW-Operationen müssen im Vergleich zur Betrachtung von PWUE-Operationen lediglich an einigen Stellen im Code das Essen und das Wende von Pfannkuchen vertauscht werden. In allen anderen Belangen bleibt die Umsetzung gleich, beschrieben wird hier also nur die Umsetzung der Berechnungen der Funktion A_E.
Für diese wird neben dem Stapel eine Variable verwaltet, die am Ende den Index des Pfannkuchens im Stapel enthalten soll, der am Anfang des längsten, sortierten untersten Teil des Stapels liegt. Am Anfang der Ausführung enthält diese Variable den Index des untersten Pfannkuchens.
In einer `while`-Schleife wird diese Variable nun so lange wie möglich dekrementiert, also bis sie entweder auf den obersten Pfannkuchen zeigt, oder der entsprechende Postfix nicht mehr sortiert ist. Danach kann schon die Ausgabe gemacht werden.

3 Beispiele

3.1 Aufgabenteil a: Berechnung der Funktion A

Ausgabe für die Eingabedatei pancake0.txt:

Fuer das Sortieren des Stapels (3 2 4 5 1) werden mindestens 2 PWUE-Operationen benoetigt.
Es ist also A(3 2 4 5 1) = 2.
Folgende Operationen sortieren den Stapel mit der minimalen Anzahl an Schritten:

Nach Anwenden der PWUE-Operation auf die ersten 5 Pfannkuchen:
(5 4 2 3)

Nach Anwenden der PWUE-Operation auf die ersten 4 Pfannkuchen:
(2 4 5)

Damit ist der Pfannkuchenstapel sortiert.

Ausgabe für die Eingabedatei pancake1.txt:

Fuer das Sortieren des Stapels (6 3 1 7 4 2 5) werden mindestens 3 PWUE-Operationen benoetigt.
Es ist also A(6 3 1 7 4 2 5) = 3.
Folgende Operationen sortieren den Stapel mit der minimalen Anzahl an Schritten:

Nach Anwenden der PWUE-Operation auf die ersten 3 Pfannkuchen:
(3 6 7 4 2 5)

Nach Anwenden der PWUE-Operation auf die ersten 4 Pfannkuchen:
(7 6 3 2 5)

Nach Anwenden der PWUE-Operation auf die ersten 5 Pfannkuchen:
(2 3 6 7)

Damit ist der Pfannkuchenstapel sortiert.

Ausgabe für die Eingabedatei pancake2.txt:

Fuer das Sortieren des Stapels (8 1 7 5 3 6 4 2) werden mindestens 4 PWUE-Operationen benoetigt.
Es ist also A(8 1 7 5 3 6 4 2) = 4.
Folgende Operationen sortieren den Stapel mit der minimalen Anzahl an Schritten:

Nach Anwenden der PWUE-Operation auf die ersten 2 Pfannkuchen:
(8 7 5 3 6 4 2)

Nach Anwenden der PWUE-Operation auf die ersten 3 Pfannkuchen:
(7 8 3 6 4 2)

Nach Anwenden der PWUE-Operation auf die ersten 3 Pfannkuchen:
(8 7 6 4 2)

Nach Anwenden der PWUE-Operation auf die ersten 5 Pfannkuchen:
(4 6 7 8)

Damit ist der Pfannkuchenstapel sortiert.

Ausgabe für die Eingabedatei pancake3.txt:

Fuer das Sortieren des Stapels (5 10 1 11 4 8 2 9 7 3 6) werden mindestens 6 PWUE-Operationen benoetigt.
Es ist also A(5 10 1 11 4 8 2 9 7 3 6) = 6.
Folgende Operationen sortieren den Stapel mit der minimalen Anzahl an Schritten:

Nach Anwenden der PWUE-Operation auf die ersten 1 Pfannkuchen:
(10 1 11 4 8 2 9 7 3 6)

Nach Anwenden der PWUE-Operation auf die ersten 1 Pfannkuchen:
(1 11 4 8 2 9 7 3 6)

Nach Anwenden der PWUE-Operation auf die ersten 7 Pfannkuchen:
(9 2 8 4 11 1 3 6)

Nach Anwenden der PWUE-Operation auf die ersten 2 Pfannkuchen:
(9 8 4 11 1 3 6)

Nach Anwenden der PWUE-Operation auf die ersten 7 Pfannkuchen:
(3 1 11 4 8 9)

Nach Anwenden der PWUE-Operation auf die ersten 3 Pfannkuchen:
(1 3 4 8 9)

Damit ist der Pfannkuchenstapel sortiert.

Ausgabe für die Eingabedatei pancake4.txt:

Fuer das Sortieren des Stapels (7 4 11 5 10 6 1 13 12 9 3 8 2) werden mindestens 7 PWUE-Operationen benoetigt.
Es ist also A(7 4 11 5 10 6 1 13 12 9 3 8 2) = 7.

Folgende Operationen sortieren den Stapel mit der minimalen Anzahl an Schritten:

Nach Anwenden der PWUE-Operation auf die ersten 1 Pfannkuchen:
(4 11 5 10 6 1 13 12 9 3 8 2)

Nach Anwenden der PWUE-Operation auf die ersten 1 Pfannkuchen:
(11 5 10 6 1 13 12 9 3 8 2)

Nach Anwenden der PWUE-Operation auf die ersten 2 Pfannkuchen:
(11 10 6 1 13 12 9 3 8 2)

Nach Anwenden der PWUE-Operation auf die ersten 5 Pfannkuchen:
(1 6 10 11 12 9 3 8 2)

Nach Anwenden der PWUE-Operation auf die ersten 6 Pfannkuchen:
(12 11 10 6 1 3 8 2)

Nach Anwenden der PWUE-Operation auf die ersten 8 Pfannkuchen:
(8 3 1 6 10 11 12)

Nach Anwenden der PWUE-Operation auf die ersten 4 Pfannkuchen:
(1 3 8 10 11 12)

Damit ist der Pfannkuchenstapel sortiert.

Ausgabe für die Eingabedatei pancake5.txt:

Fuer das Sortieren des Stapels (4 13 10 8 2 3 7 9 14 1 12 6 5 11) werden mindestens 6 PWUE-Operationen benoetigt.
Es ist also A(4 13 10 8 2 3 7 9 14 1 12 6 5 11) = 6.
Folgende Operationen sortieren den Stapel mit der minimalen Anzahl an Schritten:

Nach Anwenden der PWUE-Operation auf die ersten 1 Pfannkuchen:
(13 10 8 2 3 7 9 14 1 12 6 5 11)

Nach Anwenden der PWUE-Operation auf die ersten 13 Pfannkuchen:
(5 6 12 1 14 9 7 3 2 8 10 13)

Nach Anwenden der PWUE-Operation auf die ersten 7 Pfannkuchen:
(9 14 1 12 6 5 3 2 8 10 13)

Nach Anwenden der PWUE-Operation auf die ersten 9 Pfannkuchen:
(2 3 5 6 12 1 14 9 10 13)

Nach Anwenden der PWUE-Operation auf die ersten 5 Pfannkuchen:
(6 5 3 2 1 14 9 10 13)

Nach Anwenden der PWUE-Operation auf die ersten 6 Pfannkuchen:
(1 2 3 5 6 9 10 13)

Damit ist der Pfannkuchenstapel sortiert.

Ausgabe für die Eingabedatei pancake6.txt:

Fuer das Sortieren des Stapels (14 8 4 12 13 2 1 15 7 11 3 9 5 10 6) werden mindestens 8 PWUE-Operationen benoetigt.
Es ist also A(14 8 4 12 13 2 1 15 7 11 3 9 5 10 6) = 8.
Folgende Operationen sortieren den Stapel mit der minimalen Anzahl an Schritten:

Nach Anwenden der PWUE-Operation auf die ersten 1 Pfannkuchen:
(8 4 12 13 2 1 15 7 11 3 9 5 10 6)

Nach Anwenden der PWUE-Operation auf die ersten 2 Pfannkuchen:
(8 12 13 2 1 15 7 11 3 9 5 10 6)

Nach Anwenden der PWUE-Operation auf die ersten 6 Pfannkuchen:
(1 2 13 12 8 7 11 3 9 5 10 6)

Nach Anwenden der PWUE-Operation auf die ersten 11 Pfannkuchen:
(5 9 3 11 7 8 12 13 2 1 6)

Nach Anwenden der PWUE-Operation auf die ersten 2 Pfannkuchen:
(5 3 11 7 8 12 13 2 1 6)

Nach Anwenden der PWUE-Operation auf die ersten 3 Pfannkuchen:
(3 5 7 8 12 13 2 1 6)

Nach Anwenden der PWUE-Operation auf die ersten 6 Pfannkuchen:
(12 8 7 5 3 2 1 6)

Nach Anwenden der PWUE-Operation auf die ersten 8 Pfannkuchen:
(1 2 3 5 7 8 12)

Damit ist der Pfannkuchenstapel sortiert.

Ausgabe für die Eingabedatei pancake7.txt:

Fuer das Sortieren des Stapels (8 5 10 15 3 7 13 6 2 4 12 9 1 14 16 11) werden mindestens 8 PWUE-Operationen benoetigt.
Es ist also A(8 5 10 15 3 7 13 6 2 4 12 9 1 14 16 11) = 8.
Folgende Operationen sortieren den Stapel mit der minimalen Anzahl an Schritten:

Nach Anwenden der PWUE-Operation auf die ersten 1 Pfannkuchen:
(5 10 15 3 7 13 6 2 4 12 9 1 14 16 11)

Nach Anwenden der PWUE-Operation auf die ersten 3 Pfannkuchen:
(10 5 3 7 13 6 2 4 12 9 1 14 16 11)

Nach Anwenden der PWUE-Operation auf die ersten 8 Pfannkuchen:
(2 6 13 7 3 5 10 12 9 1 14 16 11)

Nach Anwenden der PWUE-Operation auf die ersten 4 Pfannkuchen:
(13 6 2 3 5 10 12 9 1 14 16 11)

Nach Anwenden der PWUE-Operation auf die ersten 9 Pfannkuchen:
(9 12 10 5 3 2 6 13 14 16 11)

Nach Anwenden der PWUE-Operation auf die ersten 11 Pfannkuchen:
(16 14 13 6 2 3 5 10 12 9)

Nach Anwenden der PWUE-Operation auf die ersten 10 Pfannkuchen:
(12 10 5 3 2 6 13 14 16)

Nach Anwenden der PWUE-Operation auf die ersten 6 Pfannkuchen:
(2 3 5 10 12 13 14 16)

Damit ist der Pfannkuchenstapel sortiert.

Laufzeit:

Die oben gezeigten Ergebnisse wurden alle innerhalb von weniger als 60 Sekunden auf einem durchschnittlichen Computer erzielt.

3.2 Aufgabenteil b: Berechnung der Funktion P

Ausgabe für $n = 12$:

Es gilt P(1) = 0
Ein beispielhafter Pfannkuchenstapel S mit A(S) = P(1) = 0 ist:
(1)

Es gilt P(2) = 1
Ein beispielhafter Pfannkuchenstapel S mit A(S) = P(2) = 1 ist:
(2 1)

Es gilt P(3) = 2
Ein beispielhafter Pfannkuchenstapel S mit A(S) = P(3) = 2 ist:
(2 3 1)

Es gilt P(4) = 2
Ein beispielhafter Pfannkuchenstapel S mit A(S) = P(4) = 2 ist:
(2 3 1 4)

Es gilt P(5) = 3
Ein beispielhafter Pfannkuchenstapel S mit A(S) = P(5) = 3 ist:
(1 5 2 4 3)

Es gilt P(6) = 3
Ein beispielhafter Pfannkuchenstapel S mit A(S) = P(6) = 3 ist:
(2 6 3 1 4 5)

Es gilt P(7) = 4
Ein beispielhafter Pfannkuchenstapel S mit A(S) = P(7) = 4 ist:
(7 1 6 3 5 4 2)

Es gilt P(8) = 5
Ein beispielhafter Pfannkuchenstapel S mit A(S) = P(8) = 5 ist:
(4 6 3 7 1 8 5 2)

Es gilt P(9) = 5
Ein beispielhafter Pfannkuchenstapel S mit A(S) = P(9) = 5 ist:
(7 9 1 6 3 5 8 4 2)

Es gilt P(10) = 6
Ein beispielhafter Pfannkuchenstapel S mit A(S) = P(10) = 6 ist:
(4 9 5 1 6 2 7 10 3 8)

Es gilt P(11) = 6
Ein beispielhafter Pfannkuchenstapel S mit A(S) = P(11) = 6 ist:
(4 8 5 3 6 1 9 7 11 2 10)

Es gilt P(12) = 7
Ein beispielhafter Pfannkuchenstapel S mit A(S) = P(12) = 7 ist:
(6 10 7 3 8 1 12 4 9 11 5 2)

Laufzeit:

Für $n = 11$ gibt das Programm das Ergebnis auf einem durchschnittlichen Computer nach etwa 40 Sekunden aus. Für $n = 12$ beträgt diese Laufzeit schon etwa 45 Minuten.

3.3 Erweiterung: Berechnung der Funktion A_R

Ausgabe für die Eingabedatei pancake0.txt:

Fuer das Sortieren des Stapels (3 2 4 5 1) werden mindestens 2 PEUW-Operationen benoetigt.
Es ist also A(3 2 4 5 1) = 2.
Folgende Operationen sortieren den Stapel mit der minimalen Anzahl an Schritten:

Nach Anwenden der PEUW-Operation auf die ersten 4 Pfannkuchen:
(5 4 2 1)

Nach Anwenden der PEUW-Operation auf die ersten 4 Pfannkuchen:
(1 2 4)

Damit ist der Pfannkuchenstapel sortiert.

Ausgabe für die Eingabedatei pancake1.txt:

Fuer das Sortieren des Stapels (6 3 1 7 4 2 5) werden mindestens 3 PEUW-Operationen benoetigt.
Es ist also A(6 3 1 7 4 2 5) = 3.
Folgende Operationen sortieren den Stapel mit der minimalen Anzahl an Schritten:

Nach Anwenden der PEUW-Operation auf die ersten 4 Pfannkuchen:
(7 1 3 4 2 5)

Nach Anwenden der PEUW-Operation auf die ersten 5 Pfannkuchen:
(2 4 3 1 5)

Nach Anwenden der PEUW-Operation auf die ersten 4 Pfannkuchen:
(1 3 4 5)

Damit ist der Pfannkuchenstapel sortiert.

Ausgabe für die Eingabedatei pancake2.txt:

Fuer das Sortieren des Stapels (8 1 7 5 3 6 4 2) werden mindestens 4 PEUW-Operationen benoetigt.
Es ist also A(8 1 7 5 3 6 4 2) = 4.
Folgende Operationen sortieren den Stapel mit der minimalen Anzahl an Schritten:

Nach Anwenden der PEUW-Operation auf die ersten 1 Pfannkuchen:
(1 7 5 3 6 4 2)

Nach Anwenden der PEUW-Operation auf die ersten 3 Pfannkuchen:
(5 7 3 6 4 2)

Nach Anwenden der PEUW-Operation auf die ersten 3 Pfannkuchen:
(3 7 6 4 2)

Nach Anwenden der PEUW-Operation auf die ersten 5 Pfannkuchen:
(2 4 6 7)

Damit ist der Pfannkuchenstapel sortiert.

Ausgabe für die Eingabedatei pancake3.txt:

Fuer das Sortieren des Stapels (5 10 1 11 4 8 2 9 7 3 6) werden mindestens 5 PEUW-Operationen benoetigt.
Es ist also A(5 10 1 11 4 8 2 9 7 3 6) = 5.
Folgende Operationen sortieren den Stapel mit der minimalen Anzahl an Schritten:

Nach Anwenden der PEUW-Operation auf die ersten 3 Pfannkuchen:
(1 10 11 4 8 2 9 7 3 6)

Nach Anwenden der PEUW-Operation auf die ersten 4 Pfannkuchen:
(4 11 10 8 2 9 7 3 6)

Nach Anwenden der PEUW-Operation auf die ersten 9 Pfannkuchen:
(6 3 7 9 2 8 10 11)

Nach Anwenden der PEUW-Operation auf die ersten 4 Pfannkuchen:
(9 7 3 2 8 10 11)

Nach Anwenden der PEUW-Operation auf die ersten 4 Pfannkuchen:
(2 3 7 8 10 11)

Damit ist der Pfannkuchenstapel sortiert.

Ausgabe für die Eingabedatei pancake4.txt:

Fuer das Sortieren des Stapels (7 4 11 5 10 6 1 13 12 9 3 8 2) werden mindestens 6 PEUW-Operationen benoetigt.
Es ist also A(7 4 11 5 10 6 1 13 12 9 3 8 2) = 6.
Folgende Operationen sortieren den Stapel mit der minimalen Anzahl an Schritten:

Nach Anwenden der PEUW-Operation auf die ersten 3 Pfannkuchen:
(11 4 5 10 6 1 13 12 9 3 8 2)

Nach Anwenden der PEUW-Operation auf die ersten 9 Pfannkuchen:
(9 12 13 1 6 10 5 4 3 8 2)

Nach Anwenden der PEUW-Operation auf die ersten 5 Pfannkuchen:
(6 1 13 12 10 5 4 3 8 2)

Nach Anwenden der PEUW-Operation auf die ersten 9 Pfannkuchen:
(8 3 4 5 10 12 13 1 2)

Nach Anwenden der PEUW-Operation auf die ersten 8 Pfannkuchen:
(1 13 12 10 5 4 3 2)

Nach Anwenden der PEUW-Operation auf die ersten 8 Pfannkuchen:
(2 3 4 5 10 12 13)

Damit ist der Pfannkuchenstapel sortiert.

Ausgabe für die Eingabedatei pancake5.txt:

Fuer das Sortieren des Stapels (4 13 10 8 2 3 7 9 14 1 12 6 5 11) werden mindestens 6 PEUW-Operationen benoetigt.
Es ist also A(4 13 10 8 2 3 7 9 14 1 12 6 5 11) = 6.

Folgende Operationen sortieren den Stapel mit der minimalen Anzahl an Schritten:

Nach Anwenden der PEUW-Operation auf die ersten 7 Pfannkuchen:
(7 3 2 8 10 13 9 14 1 12 6 5 11)

Nach Anwenden der PEUW-Operation auf die ersten 9 Pfannkuchen:
(1 14 9 13 10 8 2 3 12 6 5 11)

Nach Anwenden der PEUW-Operation auf die ersten 3 Pfannkuchen:
(9 14 13 10 8 2 3 12 6 5 11)

Nach Anwenden der PEUW-Operation auf die ersten 11 Pfannkuchen:
(11 5 6 12 3 2 8 10 13 14)

Nach Anwenden der PEUW-Operation auf die ersten 4 Pfannkuchen:
(12 6 5 3 2 8 10 13 14)

Nach Anwenden der PEUW-Operation auf die ersten 5 Pfannkuchen:
(2 3 5 6 8 10 13 14)

Damit ist der Pfannkuchenstapel sortiert.

Ausgabe für die Eingabedatei pancake6.txt:

Fuer das Sortieren des Stapels (14 8 4 12 13 2 1 15 7 11 3 9 5 10 6) werden mindestens 7 PEUW-Operationen benoetigt.
Es ist also A(14 8 4 12 13 2 1 15 7 11 3 9 5 10 6) = 7.
Folgende Operationen sortieren den Stapel mit der minimalen Anzahl an Schritten:

Nach Anwenden der PEUW-Operation auf die ersten 5 Pfannkuchen:
(13 12 4 8 2 1 15 7 11 3 9 5 10 6)

Nach Anwenden der PEUW-Operation auf die ersten 8 Pfannkuchen:
(7 15 1 2 8 4 12 11 3 9 5 10 6)

Nach Anwenden der PEUW-Operation auf die ersten 13 Pfannkuchen:
(6 10 5 9 3 11 12 4 8 2 1 15)

Nach Anwenden der PEUW-Operation auf die ersten 3 Pfannkuchen:
(5 10 9 3 11 12 4 8 2 1 15)

Nach Anwenden der PEUW-Operation auf die ersten 4 Pfannkuchen:
(3 9 10 11 12 4 8 2 1 15)

Nach Anwenden der PEUW-Operation auf die ersten 6 Pfannkuchen:
(4 12 11 10 9 8 2 1 15)

Nach Anwenden der PEUW-Operation auf die ersten 8 Pfannkuchen:
(1 2 8 9 10 11 12 15)

Damit ist der Pfannkuchenstapel sortiert.

Ausgabe für die Eingabedatei pancake7.txt:

Fuer das Sortieren des Stapels (8 5 10 15 3 7 13 6 2 4 12 9 1 14 16 11) werden mindestens 8 PEUW-Operationen benoetigt.
Es ist also A(8 5 10 15 3 7 13 6 2 4 12 9 1 14 16 11) = 8.
Folgende Operationen sortieren den Stapel mit der minimalen Anzahl an Schritten:

Nach Anwenden der PEUW-Operation auf die ersten 1 Pfannkuchen:
(5 10 15 3 7 13 6 2 4 12 9 1 14 16 11)

Nach Anwenden der PEUW-Operation auf die ersten 12 Pfannkuchen:
(1 9 12 4 2 6 13 7 3 15 10 14 16 11)

Nach Anwenden der PEUW-Operation auf die ersten 10 Pfannkuchen:
(15 3 7 13 6 2 4 12 9 10 14 16 11)

Nach Anwenden der PEUW-Operation auf die ersten 13 Pfannkuchen:
(11 16 14 10 9 12 4 2 6 13 7 3)

Nach Anwenden der PEUW-Operation auf die ersten 12 Pfannkuchen:
(3 7 13 6 2 4 12 9 10 14 16)

Nach Anwenden der PEUW-Operation auf die ersten 3 Pfannkuchen:
(13 7 6 2 4 12 9 10 14 16)

Nach Anwenden der PEUW-Operation auf die ersten 6 Pfannkuchen:
(12 4 2 6 7 9 10 14 16)

Nach Anwenden der PEUW-Operation auf die ersten 3 Pfannkuchen:
(2 4 6 7 9 10 14 16)

Damit ist der Pfannkuchenstapel sortiert.

3.4 Erweiterung: Berechnung der Funktion P_R

Es wird nur die Ausgabe für $n = 11$ gezeigt. Größere Werte von n wären von der Laufzeit her auch möglich, aber für $n = 12$ belegt das Programm auf dem ausführenden Computer (in einer zugegebenermaßen sehr kleinen Partition) schon vor Beendigung den gesamten Arbeitsspeicher.

Es gilt PR(1) = 0
Ein beispielhafter Pfannkuchenstapel S mit A(S) = PR(1) = 0 ist:
(1)

Es gilt PR(2) = 1
Ein beispielhafter Pfannkuchenstapel S mit A(S) = PR(2) = 1 ist:
(2 1)

Es gilt PR(3) = 1
Ein beispielhafter Pfannkuchenstapel S mit A(S) = PR(3) = 1 ist:
(2 1 3)

Es gilt PR(4) = 2
Ein beispielhafter Pfannkuchenstapel S mit A(S) = PR(4) = 2 ist:
(1 4 2 3)

Es gilt PR(5) = 3
Ein beispielhafter Pfannkuchenstapel S mit A(S) = PR(5) = 3 ist:
(1 4 3 5 2)

Es gilt PR(6) = 3
Ein beispielhafter Pfannkuchenstapel S mit A(S) = PR(6) = 3 ist:
(1 4 3 5 2 6)

Es gilt PR(7) = 4

Ein beispielhafter Pfannkuchenstapel S mit A(S) = PR(7) = 4 ist:
(1 4 7 3 5 2 6)

Es gilt PR(8) = 4
Ein beispielhafter Pfannkuchenstapel S mit A(S) = PR(8) = 4 ist:
(1 8 7 3 6 4 5 2)

Es gilt PR(9) = 5
Ein beispielhafter Pfannkuchenstapel S mit A(S) = PR(9) = 5 ist:
(1 9 2 8 4 7 5 6 3)

Es gilt PR(10) = 6
Ein beispielhafter Pfannkuchenstapel S mit A(S) = PR(10) = 6 ist:
(1 2 7 6 8 5 9 4 10 3)

Es gilt PR(11) = 6
Ein beispielhafter Pfannkuchenstapel S mit A(S) = PR(11) = 6 ist:
(1 2 7 6 8 5 9 4 10 3 11)

3.5 Erweiterung: Berechnung der Funktion A_E:

Ausgabe für die Eingabedatei pancake0.txt:

Der Pfannkuchenstapel kann durch 4 PE-Operationen sortiert werden.

Ausgabe für die Eingabedatei pancake1.txt:

Der Pfannkuchenstapel kann durch 5 PE-Operationen sortiert werden.

Ausgabe für die Eingabedatei pancake2.txt:

Der Pfannkuchenstapel kann durch 7 PE-Operationen sortiert werden.

Ausgabe für die Eingabedatei pancake3.txt:

Der Pfannkuchenstapel kann durch 9 PE-Operationen sortiert werden.

Ausgabe für die Eingabedatei pancake4.txt:

Der Pfannkuchenstapel kann durch 12 PE-Operationen sortiert werden.

Ausgabe für die Eingabedatei pancake5.txt:

Der Pfannkuchenstapel kann durch 12 PE-Operationen sortiert werden.

Ausgabe für die Eingabedatei pancake6.txt:

Der Pfannkuchenstapel kann durch 14 PE-Operationen sortiert werden.

Ausgabe für die Eingabedatei pancake7.txt:

Der Pfannkuchenstapel kann durch 15 PE-Operationen sortiert werden.

Laufzeit: Die vorgegebenen Eingabedateien werden auf einem durchschnittlichen Computer in weniger als einer Sekunde bearbeitet. Aufgrund seiner linearen Laufzeit könnte das Programm problemlos auch Pfannkuchenstapel mit $\sim 10^6$ Pfannkuchen mit einer Laufzeit im Bereich einer Sekunde verarbeiten.

4 Quellcode

4.1 Berechnung der Funktionen A und P

Die verwendeten Definitionen:

```
#define vs vector<short>
#define ll long long
#define pil pair<int, ll>
#include <bits/stdc++.h>
using namespace std;
```

Die beschriebenen Funktionen für das Behandeln von Pfannkuchenstapeln:

```
//Vereinfacht einen Pfannkuchenstapel, in dem genau ein Pfannkuchen fehlt:
void simplify(vs& pancakes)
{
    int pcounter = pancakes.size(); //Anzahl der verbleibenden Pfannkuchen

    //Berechnen des fehlenden Pfannkuchens:
    int sum = 0; //Summe aller enthaltenen Pfannkuchen
    for(short i : pancakes)
    {
        sum += i;
    }
    int missing = ((pcounter+1)*(pcounter+2))/2-sum; //Berechnung des fehlenden Pfannkuchens

    //Vereinfachung durch Verkleinerung von Pfannkuchenzahlen:
    for(int i = 0; i < pcounter; i++) //Durchgehen aller Pfannkuchen
    {
        if(pancakes[i] > missing) //Der Pfannkuchen ist groesser als der fehlende
        {
            pancakes[i]--; //Verkleinern des Pfannkuchens
        }
    }

    //Vereinfachung durch Entfernen von Pfannkuchen an der Unterseite des Stapels
    while(pcounter > 0 && pcounter == pancakes[pcounter-1])
    {
        //Der groesste Pfannkuchen liegt ganz unten
        pancakes.pop_back(); //Entfernen des untersten Pfannkuchens
        pcounter--;
    }
}

//Vereinfacht einen Pfannkuchenstapel, in dem eine unbekannte Anzahl Pfannkuchen fehlt:
void outputSimplify(vs& pancakes)
{
    int pcounter = pancakes.size();
    int m = *max_element(pancakes.begin(), pancakes.end()); //Groesster Pfannkuchen

    vs pancopy = pancakes; //Pfannkuchen nach Groesse sortiert
    sort(pancopy.begin(), pancopy.end());
    vs dict(m); //dict[i] = Wert, den Pfannkuchen i annehmen muss

    int pre = 0;
    for(int i : pancopy)
    {
        int diff = i-pre; //Verringern des Wertes je nach Luecke zum vorherigen
        dict[i-1] = 1+i-diff;
        pre = dict[i-1];
    }

    //Aktualisieren der Pfannkuchen:
    for(int i = 0; i < pcounter; i++)
    {
        pancakes[i] = dict[pancakes[i]-1];
    }

    //Vereinfachung durch Entfernen von Pfannkuchen an der Unterseite des Stapels
    while(!pancakes.empty() && pancakes[pancakes.size()-1] == (int) pancakes.size())
    {
```

```
59          pancakes.pop_back();
        }
61  }

63  vs flipOperation(vs pancakes, int range) //Umsetzung einer Wende-Und-Ess-Operation
    {
65      //range gibt die Anzahl der betroffenen Pfannkuchen an
        reverse(pancakes.begin(), pancakes.begin()+range+1); //Umdrehen der Pfannkuchen
67      pancakes.erase(pancakes.begin()); //Aufessen des nun obersten Pfannkuchens
        simplify(pancakes);
69      return pancakes;
    }
71
    //Gibt Nummer dieses Stapels in lexikographischer Sortierung aller Stapel (dieser Groesse!) zurueck
73  ll stackIndex(vs pancakes, vector<ll>& faculty)
    {
75      int pcounter = pancakes.size();

77      //Berechnen des Indizes:
        ll index = 0;
79      for(int i = 0; i < pcounter; i++) //Durchgehen aller Positionen im Stapel
        {
81          index += faculty[pcounter-i-1]*((ll) pancakes[i]-1);
            //Erhohen des Indizes nach Einreihung des Elements unter den noch verfuegbaren
83          for(int j = i+1; j < pcounter; j++) //Anpassen der folgenden Zahlen
            {
85              if(pancakes[j] > pancakes[i])
                {
87                  pancakes[j]--;
                }
89          }
        }
91      return index;
    }
93
    //Gibt Stapel mit entsprechender Anzahl an Elementen und diesem Index zurueck
95  vs explicitStack(int pcounter, ll index, vector<ll>& faculty)
    {
97      vs output(pcounter); //Der zu berechnende Stack
        vs remaining(pcounter); //Noch nicht auf den Stapel gelegte Pfannkuchen
99      iota(remaining.begin(), remaining.end(), 1);

101     for(int i = 0; i < pcounter; i++)
        {
103         //Hinzufuegen des naechsten Pfannkuchens
            int position = index/faculty[pcounter-i-1];
105         output[i] = remaining[position];
            remaining.erase(remaining.begin()+position);
107         index -= faculty[pcounter-i-1]*position;
        }
109
        return output;
111 }

113 vs insertCake(int pos, short val, vs pancakes)
    {
115     /*Erstellt durch das Einfuegen des uebergebenen Werts
        an der gegebenen Stelle einen Pfannkuchenstapel S mit
117     A(S) = A(pancakes)+1, der sich durch eine PWUE-Operation
        in pancakes ueberfuehren laesst.*/
119     for(int i = 0; i < (int) pancakes.size(); i++)
        {
121         if(pancakes[i] >= val)
            {
123             pancakes[i]++; //Erhoehen von nun zu kleinen Werten
            }
125     }
        //Umkehrung einer PWUE-Operation:
127     reverse(pancakes.begin(), pancakes.begin()+pos);
        pancakes.insert(pancakes.begin()+pos, val);
129     return pancakes;
    }
```

Die Funktion `evalA()`, die die Funktion A berechnet:

```
void evalA(vector<ll>& faculty) //Die Funktion fuer das Berechnend der Funktion A(S)
{
    int pcounter; //Anzahl der zu sortierenden Pfannkuchen
    vs pancakes; //Der zu sortierende Pfannkuchenstapel
    ofstream fout; //Ausgabedatei

    readInput(pcounter, pancakes, fout); //Einlesen der Eingabe

    if(sorted(pancakes))
    {
        //Es muessen keine Operationen mehr durchgefuehrt werden!
        fout<<"Der␣Pfannkuchenstapel␣"<<stackOutput(pancakes)<<"␣ist␣schon␣sortiert.\n";
        fout<<"Fuer␣diesen␣Stapel␣S␣ist␣entsprechend␣A(S)␣=␣0\n";
        fout.close();
        exit(0);
    }

    unordered_map<ll, vector<pil>> backtracking;
    //Der Stapel mit (j+1) Pfannkuchen und Index i kann vom Stapel in backtracking[i][j]
    //(Anzahl, Index) durch eine PWUE-Operation erreicht werden
    queue<pil> q;
    //Warteschlange fuer Pfannkuchenstapel. (i, j) := Stapel mit i Pfannkuchen und Index j
    q.push({pcounter, stackIndex(pancakes, faculty)});

    while(true) //Berechnung durchfuehren bis zum Erreichen einer Loesung
    {
        //Aktuell zu untersuchender Stapel
        vs current = explicitStack(q.front().first, q.front().second, faculty);
        q.pop();
        ll currentIndex = stackIndex(current, faculty);
        //Durchgehen aller moeglichen Positionen fuer eine PWUE-Operation:
        for(int i = 0; i < (int) current.size(); i++)
        {
            vs test = flipOperation(current, i); //Stapel nach einer solchen PWUE-Operation
            if(test.empty())
            {
                //Der Pfannkuchenstapel ist nun sortiert!
                //Eine Loesung wird ausgegeben
                outputPWUESequence(fout, backtracking, current, pancakes, faculty);
                exit(0);
            }
            //Der Pfannkuchenstapel ist noch nicht sortiert
            ll testIndex = stackIndex(test, faculty);
            //Aktualisieren von backtracking:
            if(backtracking.count(testIndex) == 0)
            {
                backtracking[testIndex].assign(pcounter, {-1, -1});
            }
            if(backtracking[testIndex][test.size()-1] == make_pair(-1, (ll) -1))
            {
                //Der Stapel wurde noch nicht untersucht
                backtracking[testIndex][test.size()-1] = {current.size(), currentIndex};
                q.push({test.size(), testIndex});
            }
        }
    }
}
```

Die Funktion `derive()`, die eine Menge $M_{k,l}$ aus $M_{(k-1),(l-1)}$ berechnet:

```
void derive(vector<vs>& target, vector<vs>& source, vector<unordered_map<ll, short>>& moves,
 vector<ll>& faculty)
{
    /*Diese Funktion berechnet alle Pfannkuchenstapel, die durch eine PWUE-Operation
    nur in Stapel ueberfuehrt werden koennen,
    die entweder in source liegen oder die fuer das Sortieren mehr Operationen
    brauchen als die Stapel in der Liste source. Diese werden dann in target geschoben
    */
    if(source.empty())
    {
        return;
```

```
    }
    int pcounter = source[0].size(); //Anzahl der Pfannkuchen in Stapeln in source
    unordered_set<ll> tested; //Schon fuer target untersucht

    for(vs v : source) //Durchgehen aller Pfannkuchenstapel in source
    {
        short vMoves = moves[v.size()-1][stackIndex(v, faculty)]; //A(v)

        //Durchgehen aller Stapel, die durch eine PWUE-Operation in v ueberfuehrt werden koennen
        for(int pos = 0; pos <= pcounter; pos++)
        {
            //Durchgehen der Position, an der ein neuer Pfannkuchen eingefuegt wird
            for(short val = 1; val <= pcounter+1; val++) //Wert des neuen Pfannkuchens
            {
                vs test = insertCake(pos, val, v); //Zu untersuchender Pfannkuchenstapel
                ll testInd = stackIndex(test, faculty);
                if(tested.count(testInd) != 0) //Der Stapel wurde schon untersucht
                {
                    continue;
                }

                if(moves[test.size()-1].count(testInd) == 0 && !sorted(test))
                {
                    //Der Pfannkuchenstapel ist nicht sortiert und
                    //kann daher weiter untersucht werden
                    bool vSort = true; //Fuehrt das schnellstmoegliche Sortieren ueber v?
                    //Durchgehen aller Positionen fuer die PWUE-Operation
                    for(int k = 0; k <= pcounter; k++)
                    {
                        vs flipped = flipOperation(test, k); //Durchfuehren einer PWUE-Operation
                        ll fInd = stackIndex(flipped, faculty);

                        if((sorted(flipped) && !sorted(v)) || (!sorted(flipped) &&
                         (moves[flipped.size()-1].count(fInd) == 0 ||
                         moves[flipped.size()-1][fInd] < vMoves)))
                        {
                            /*Der Stapel laesst sich durch eine PWUE-Operation in einen
                            Stapel ueberfuehren,  der schneller sortierbar ist als v*/
                            vSort = false;
                            break;
                        }
                    }
                    if(vSort)
                    {
                        //Der Stapel muss in target eingefuegt werden
                        moves[test.size()-1][testInd] = vMoves+1;
                        target.push_back(test);
                    }
                }
                tested.insert(testInd);
            }
        }
    }
}
```

Die Funktion `compute()`, die die Aufrufe der Funktion `derive()` koordiniert:

```
void compute(int i, int j, vector<vector<vector<vs>>>& stacklist,
vector<unordered_map<ll, short>>& moves, vector<short>& output, int goal, vector<ll>& faculty)
{
    //Berechnet stacklist[i][j] und output[i] und alle weiteren
    //dafuer noetigen Eintraege von stacklist
    if(j == output[i])
    {
        //In stacklist[i][j] ist nur der sortierte Stapel S mit A(S) = 0
        vs pancakesSorted(i+1);
        iota(pancakesSorted.begin(), pancakesSorted.end(), 1); //Erstellen des sortieren Stapels
        stacklist[i].push_back({{pancakesSorted}}); //Einfuegen dieses Stapels
        moves[pancakesSorted.size()-1][stackIndex(pancakesSorted, faculty)] = 0;
        return;
    }

```

```
    //Erzeugen eines neuen Eintrags von stacklist:
    stacklist[i].push_back({});

    int pre = j; //stacklist[i][j] wird aus stacklist[i-1][pre] berechnet
    if(output[i] == output[i-1]) //P(i) = P(i-1), daher muss sich pre veraendern
    {
        pre++;
    }
    while(pre >= (int) stacklist[i-1].size() && (int) stacklist[i-1].size() <= output[i-1])
    {
        //Es fehlen noch notwendige Eintraege von stacklist
        compute(i-1, stacklist[i-1].size(), stacklist, moves, output, goal, faculty);
    }

    derive(stacklist[i][j], stacklist[i-1][pre], moves, faculty);
    if(j == 0) //Es wurden die Stapel S mit A(S) = P(i) untersucht
    {
        output[i] = output[i-1]+1;
    }

    if(stacklist[i][j].empty()) //Es gibt keinen Stapel S mit i Pfannkuchen und A(s) > P(i-1)
    {
        output[i]--;
        if(i == goal)
        {
            //Es muessen keine weiteren Funktionswerte von P(i) berechnet werden
            //Daher muss stacklist[i][0] nicht vollstaendig bekannt sein.
            vs simple = stacklist[i-1][0][0];
            simple.push_back(simple.size()+1); //Einfacher Stapel S mit A(S) = P(i)
            stacklist[i][0].push_back(simple);
            return;
        }
        else
        {
            pre++; //Ein weiterer Eintrag von stacklist muss berechnet werden
            while(pre >= (int) stacklist[i-1].size())
            {
                //Es fehlen noch Eintraege von stacklist, auf denen die Berechnung aufbaut
                compute(i-1, stacklist[i-1].size(), stacklist, moves, output, goal, faculty);
            }
            derive(stacklist[i][j], stacklist[i-1][pre], moves, faculty);
        }
    }
}
```

Die Funktion `evalP()`, in der P berechnet wird:

```
void evalP(vector<ll>& faculty) //Berechnung der Funktion P
{
    //Variablen fuer das Speichern der Eingabe:
    int n; //Berechne P(1) bis P(n)
    cout<<"Berechnung von P(k) fuer k = 1, 2, ..., n\n";
    cout<<"Wert von n: ";
    cin>>n;

    cout<<"Name der Ausgabedatei: ";
    string foutPath;
    cin>>foutPath;
    ofstream fout; //Ausgabedatei
    fout.open(foutPath, ios_base::out);

    //Fuer die Berechnung notwendige Datenstrukturen:
    vector<vector<vector<vs>>> stacklist (n, vector<vector<vs>> (0));
    //stacklist[i][j] = Liste von Stapeln S mit (i+1) Pfannkuchen und A(S) = P(i+1)-j
    vector<unordered_map<ll, short>> moves(n);
    //moves[i][j] = A(S) fuer einen Stapel S mit (i+1) Pfannkuchen und Index j
    vector<short> output(n, numeric_limits<short>::max()/2); //output[i] = P(i)

    //Vorgeben der Werte fuer P(1):
    stacklist[0].push_back({{{1}}});
    moves[0][stackIndex({1}, faculty)] = 0;
    output[0] = 0;
```

```
27      //Berechnen der Funktionswerte von P:
        for(int i = 1; i < n; i++)
29      {
            compute(i, 0, stacklist, moves, output, n-1, faculty);
31          cout<<"P("<<i+1<<")␣=␣"<<output[i]<<endl; //Kontrollausgabe an die Konsole
        }
33
        //Ausgabe an die Ausgabedatei:
35      for(int i = 0; i < n; i++)
        {
37          //Ausgabe von P(i)
        }
39      fout.close();
}
```

4.2 Erweiterung: Andere Operationen

Wie schon in Kapitel 2.4 begründet, gibt es im Quelltext der Berechnung der Funktionen A_R und P_R im Vergleich zum im vorherigen Abschnitt dargestellten Quelltext nur minimale Änderungen.
Deswegen wird hier nur der Quelltext für die Berechnung der Funktion A_E dokumentiert. Für diese wird die `main()`-Funktion gezeigt:

```
int main()
2 {
      int pcounter; //Anzahl der zu sortierenden Pfannkuchen
4     vs pancakes; //Der zu sortierende Pfannkuchenstapel

6     readInput(pcounter, pancakes); //Einlesen der Eingabe

8     int postfix = pcounter-1; //Start des laengsten sortierten Postfix

10  //postfix so weit wie moeglich verringern:
      while(postfix > 0 && pancakes[postfix] > pancakes[postfix-1])
12    {
          postfix--;
14    }

16    int operations = postfix; //Anzahl der benoetigten Operationen
    //Ausgabe
18
      return 0;
20  }
```