Unterschrift des Betreuers

TECHNISCHE UNIVERSITÄT WIEN

VIENNA UNIVERSITY OF TECHNOLOGY

D I P L O M A R B E I T

Analyse und Varianten von In situ-Permutationsalgorithmen

Ausgeführt am Institut für
Diskrete Mathematik und Geometrie
der Technischen Universität Wien

unter der Anleitung von
Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Alois Panholzer

durch
Silvio Dorrighi
Gumpendorferstraße 8/18
1060 Wien

Datum

Unterschrift

Worte des Dankes

Zu aller erst möchte ich mich bei Herrn Prof. Alois Panholzer für die Möglichkeit bedanken, bei ihm eine Diplomarbeit aus seinem Forschungsgebiet, der diskreten Mathematik, schreiben zu dürfen. Ich habe die persönliche Zusammenarbeit und die nette Umgangsform sehr geschätzt. Auch der gemeinsam festgelegte didaktische Schwerpunkt dieser Arbeit hat mir sehr zugesagt und ich bin froh, dass mich Prof. Panholzer, dessen didaktische Fähigkeiten ich in mehreren Vorlesungen genießen durfte, auch dahingehend gefördert hat.

Ich möchte mich als nächstes bei meinen Studienkollegen bedanken, die durch unsere gelebte Teamarbeit maßgeblich am schnellen Studienerfolg und am Spaß an der Sache teilhaben. Gerne erinnere ich mich an so manches Projekt zurück, das wir gemeinsam erfolgreich durchgeführt haben.

Ein Danke möchte ich auch gegenüber meinem Freundeskreis formulieren, der immer da ist, wenn ich etwas brauchen sollte und mit dem ich viele gemeinsame abwechslungsreiche Stunden meiner Studentenzeit verbringen durfte.

Mein besonderer Dank gilt meiner Familie, ganz besonders meinen Eltern Gerlinde Dorrighi und Günter Dorrighi, die es mir ermöglicht haben, ein Studium zu absolvieren und dabei immer einen starken Rückhalt darstellten, auf den man sich jederzeit verlassen konnte. Mathematisch gesehen waren und sind sie eine große Konstante in meinem Leben und das im besten Sinn.

Abschließend möchte ich mich noch bei Petra Goldenits bedanken, die mich auf allen Ebenen unterstützt.

Euch allen ein herzliches Dankeschön!

Kurzfassung

Diese Arbeit beschäftigt sich mit der Laufzeitanalyse von In situ-Permutationsalgorithmen, deren Ziel es ist, ein Datenfeld anhand einer Permutation umzuspeichern und dabei mit weniger als linearen Hilfsspeicher auszukommen. Dabei wird stets die Zyklenstruktur einer Permutation ausgenützt, um ein spezielles Element in jedem Zyklus, den Zyklenführer, auszuzeichnen und dort beginnend mit einer Umspeicherroutine den gesamten Zyklus im Speicher zu rotieren. Daher bezeichnet die Zyklenführersuche den Kern der Algorithmen.

D. E. Knuth (siehe [2]) war der erste, der dieser Problematik mathematisch nachgegangen ist und dessen Ergebnisse die Basis für die nachfolgende Forschung und auch für diese Arbeit darstellen. P. Kirschenhofer, H. Prodiner und R. F. Tichy nahmen diese Forschungsarbeit auf und entwickelten einen alternativen Zugang zur Berechnung höherer Momente eines charakteristischen Parameters, der ein wesentliches Gütekriterium für die gefundenen Algorithmen darstellt (siehe [3]). Auf diesen beiden Arbeiten ist im Wesentlichen das dritte Kapitel "Der klassische Algorithmus" aufgebaut. F. E. Fich, J. I. Munro und P. V. Poblete lieferten in ihrer Arbeit (siehe [1]) interessante Algorithmus-Konstruktionen, um das worst-case Szenario der Zyklenführeralgorithmen zu verbessern. Diese Modifikationen und alternativen Zugänge werden im vierten Kapitel "Varianten und Modifikationen" untersucht. J. Keller gelang es schließlich zwei Abbruchkriterien "Fastbreak 1" und "Fastbreak 2" zu finden, die in sämtlichen Algorithmen implementiert werden können und die maßgeblich zur Laufzeitverbesserung beitragen (siehe [11]). Auf dieser Arbeit baut jene von A. Panholzer, H. Prodinger und M. Riedel (siehe [12]) auf und analysiert mathematisch die Einsparungen, die Keller vorwiegend experimentell festgestellt hat. Aufgrund der großen Bedeutung dieser beiden Abbruchkriterien werden sie in je einem eigenen Kaptitel behandelt.

In diesem Spannungsfeld bewegt sich also die vorliegende Arbeit. Es werden zuerst rein theoretisch Aussagen zum Laufzeitverhalten entwickelt und anschließend in mehreren Implementierungen einige Berechnungen und aufgestellte Thesen experimentell überprüft. Diese Experimente werden im siebenten Kapitel "Experimentelles Testen" erklärt. Der zugehörige Quellcode ist dann im Anhang angefügt.

Inhaltsverzeichnis

Kapitel 1

Grundlagen

Hier werden jene mathematischen Grundlagen erklärt, die zum Verständnis der Arbeit notwendig sind. Außerdem wird auf die verwendete Notation eingegangen. Vorausgesetzt wird lediglich grundlegendes mathematisches Verständnis sowie Grundbegriffe der Mengenlehre, speziell der Mengenbegriff, wobei für vorliegende Arbeit die intuitive Auffassung als ein Zusammenschluss von beliebig vielen (möglicherweise keinen) unterscheidbaren Elementen bei irrelevanter Reihenfolge genügt.

1.1 Notation

Die geschwungenen Klammern "{" und "}" kennzeichnen Mengen, deren Elemente – sofern vorhanden – mit Beistrichen getrennt sind. Die Zeichen "$\subseteq$", "$\setminus$", "$\cap$" und "$\cup$" stehen für die Enthaltensrelation, Mengendifferenz, Mengendurchschnitt und Mengenvereinigung. Die Enthaltensrelationen $\in$ bzw. $\subseteq$ und deren Negationen $\notin$ bzw. $\not\subseteq$ geben an, dass sich ein Element bzw. eine Untermenge in einer Menge befindet oder nicht in der Menge enthalten ist. Ein Doppelpunkt oder ein gerader Strich in der Menge leiten Bedingungen ein, die an Elemente der Menge geknüpft sind und durch Beistriche bzw. Junktoren getrennt sind.

Die natürlichen Zahlen werden mit $\mathbb{N}$ bezeichnet und beinhalten hier auch die Null; sie sind also definiert als $\mathbb{N} = \{0, 1, 2, ...\}$. Mit $\mathbb{Z}$ sind die ganzen Zahlen gemeint, die die natürlichen Zahlen um das additiv inverse Element erweitern, also $\mathbb{Z} = \{..., -2, -1, 0, 1, 2, ...\}$. $\mathbb{R}$ steht für die reellen Zahlen, während $\mathbb{C}$ die komplexen Zahlen bezeichnet. In $\mathbb{R}$ können Intervalle durch eine untere Grenze $a \in \mathbb{R}$ und eine obere Grenze $b \in \mathbb{R}$ angegeben werden; runde Klammern bezeichnen Intervalle mit exkludierten Grenzen, während eckige Klammern die Grenzen zum Intervallbereich hinzufügen. Das Intervall $(a, b]$ beinhaltet also alle reellen Zahlen zwischen a und b, exklusive der Zahl a und inklusive der Zahl b.

Vektoren können hier als Zahlentupel aufgefasst werden, also als Mengen, bei denen Elemente auch mehrmals vorkommen können und die Reihenfolge wesentlich ist. Sie werden mit runden Klammern eingeleitet und abgeschlossen. Die Dimension eines Vektors ist seine Länge, also die Anzahl an mit Beistrichen getrennten Elementen. Ein Array als Datenfeld kann als endlichdimensionaler Vektor aufgefasst werden.

Der Begriff "Funktion" kommt in vorliegender Arbeit sowohl als die elementare mathematische Konstruktion einer Relation zwischen zwei Mengen vor (allen Elementen des Definitionsbereichs wird je genau ein Element der Zielmenge zugeordnet), als auch als Programmabschnitt, der für beliebig viele übergebene Zahlenwerte, genannt Eingangsparameter, genau einen Rückgabewert ermittelt und dem aufrufenden Programmabschnitt zurück gibt. Das Konzept ist also dasselbe:

Eine Programmfunktion kann als eine eventuell mehrdimensionale mathematische Funktion gesehen werden. Unterprogramme ohne Rückgabewert werden Prozeduren genannt.

Definitionen werden entweder durch das Definitionszeichen ":=" bzw. "=:" (auf der Seite mit dem Doppelpunkt steht das neue Zeichen für den Ausdruck auf der anderen Seite) im Text bzw. in der Rechnung gesetzt, oder explizit mit dem Einleitewort "Definition" und anschließender Nummer eingeleitet. Analog werden auch Lemmata, Sätze, Korollare, Beispiele, Bemerkungen und Algorithmen eingeleitet – sie alle folgen einer einheitlichen Nummerierung, die aus Kapitelnummer, Unterkapitelnummer und pro Unterkapitel fortlaufender Nummer besteht. Zur Illustration sollen hier die ersten Definitionen und der erste Satz angeführt werden.

Definition 1.1.1 (Fakultät) *Sei $n \in \mathbb{N}$. Dann wird mit $n!$ die Fakultät (oft auch Faktorielle) bezeichnet, für die*

$$n! = \begin{cases} 1, & n = 0, \\ n \cdot (n-1)!, & n \geq 1, \end{cases}$$

gilt.

Dieselbe Definiton kann auch einfach im Text gesetzt werden: $n! := n \cdot (n-1) \cdot \cdots 1$ für $n \in \mathbb{N}$.

Satz 1.1.2 (Beispielsatz) *Alle Kapitel-, Subkapitel- und Subsubkapitelüberschriften sind hierarchisch nummeriert, das heißt, die Kapitel erhalten eine einfache fortlaufende Nummerierung, die Subkapitel übernehmen die Kapitelnummer und fügen nach einem Punkt eine eigene fortlaufende Nummerierung an, während die Subsubkapitelnummer sowohl aus Kapitel-, Subkapitel- und pro Subkapitel eigener fortlaufender Nummer besteht.*
Die Abbildungsnummern beinhalten die aktuelle Kapitelnummer und pro Kapitel eine eigene fortlaufende Nummer.

Beweis. Da diese Arbeit eine endliche Anzahl von Seiten hat, kann die Aussage in endlicher Zeit überprüft werden. [Das Beweisende wird durch ein kleines Quadrat rechts gekennzeichnet.] □

$\min\{x_1, ..., x_n\} = \min_{1 \leq i \leq n}\{x_i\}$ steht für die kleinste Zahl der angeführten Menge ($\max\{...\}$ analog). Damit kann man $\lfloor x \rfloor := \max\{n \in \mathbb{Z} : n \leq x\}$ und $\lceil x \rceil := \min\{n \in \mathbb{Z} : n \geq x\}$ definieren. Senkrechte Striche bezeichnen die Mächtigkeit: $|\{x_1, ..., x_n\}| = n$. Das formale Zeichen für Unendlichkeit ist ∞. Mit $f'(x)$ bzw. mit $f^{(k)}(x)$ wird die erste Ableitung bzw. die k-te Ableitung bezeichnet, wenn klar ist, nach welcher Variable abgeleitet wird. Bei mehreren auftretenden Variablen wird der Differentialoperator verwendet; also beispielsweise $\frac{\partial}{\partial u}$, um partiell nach u abzuleiten. Das Riemann'sche Integral wird wie üblich mit $\int_a^b f(x)dx$ bezeichnet, wobei die beiden Grenzen optional sind. Mit log wird in vorliegender Arbeit der natürliche Logarithmus, also der Logarithmus zur Basis e, bezeichnet; ist die Basis $a \neq e$ so wird dafür $\log_a$ geschrieben.

Definition 1.1.3 (Halbordnung, Totalordnung, strenge Totalordnung) *Eine binäre Relation "$\leq$" auf der Menge M wird als Halbordnung bezeichnet, wenn für alle $a, b, c \in M$ gilt:*

- $a \leq a$ *(Reflexivität),*
- *wenn* $a \leq b \leq c$ *dann* $a \leq c$ *(Transitivität),*
- *wenn* $a \leq b$ *und* $b \leq a$ *dann* $a = b$ *(Antisymmetrie).*

Als Totalordnung auf einer Menge M wird eine Halbordung "$\leq$" verstanden, sodass für alle $a, b \in M$ gilt: $a \leq b$ oder $b \leq a$ (alle Elemente in M sind also vergleichbar).
Eine strenge Totalordnung auf der Menge M ist eine transitive binäre Relation "$<$", sodass für alle $a, b \in M$ gilt: $a < b$ oder $a = b$ oder $b < a$ (Trichotonie-Eigenschaft).

1.1.1 Pseudo-Code

Der Code der einzelnen Algorithmen beinhaltet englische Befehle, die an gängige Programmiersprachen angelehnt und größtenteils selbsterklärend sind. Als Zuweisungsoperator wurde, wie im Formelmanipulationsprogramm Maple, das Zeichen ":=" gewählt, da es sich besser in den mathematischen Kontext dieser Arbeit einfügt. Somit steht das Gleichheitszeichen "=" für tatsächliche Gleichheit, die bei diversen Abfragen überprüft werden kann. Die Konstruktion des Codes beruht im Wesentlichen auf der Verwendung von if-else-Verzweigungen, select-case-Verzweigungen, for-Schleifen, while-Schleifen und repeat-until-Schleifen. Auf eine explizite Variablendeklaration wurde verzichtet; ebenso gibt es kein Semilokon als Befehlsende. Die Zeilen sind nummeriert und Bemerkungen bzw. Laufzeit-Zählvariablen, die nicht zum Code gehören, werden mit "▷" eingeleitet.

1.1.2 O-Notation

Bei jedem Polynom mit der Unbekannten x gibt die Ordnung die höchste Potenz von x an, die auch für wachsendes x den ausschlaggebenden Beitrag liefert. Die Funktion $f(n) = n^3+5n^2+15$ nähert sich also mit wachsendem n der Funktion n^3, da

$$\lim_{n\to\infty} \frac{n^3+5n^2+15}{n^3} = \lim_{n\to\infty} \left(1+\frac{5}{n}+\frac{15}{n^3}\right) = \lim_{n\to\infty} 1 + \underbrace{\lim_{n\to\infty} \frac{5}{n}}_{=0} + \underbrace{\lim_{n\to\infty} \frac{15}{n^3}}_{=0} = 1.$$

Solche einfachen Vergleichsfunktionen sind äußerst hilfreich beim Überprüfen der Laufzeiteigenschaft. Daher führt man auch eine eigene Notation dafür ein. Die folgende Definition wurde von [14] übernommen.

Definition 1.1.4 (Θ, O, Ω) *Sei $g : \mathbb{N}_0 \to \mathbb{N}_0$ eine Funktion. Dann sollen die folgenden Funktionenmengen definiert werden, wobei die vorkommenden Konstanten c, c_1, c_2 beliebige ganze Zahlen sein dürfen und $n_0 \in \mathbb{N}$ gelten soll:*

$$\Theta(g(n)) = \{f(n) \mid (\exists c_1, c_2, n_0 > 0), (\forall n \geq n_0) : 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)\},$$

$$O(g(n)) = \{f(n) \mid (\exists c, n_0 > 0), (\forall n \geq n_0) : 0 \leq f(n) \leq cg(n)\},$$

$$\Omega(g(n)) = \{f(n) \mid (\exists c, n_0 > 0), (\forall n \geq n_0) : 0 \leq cg(n) \leq f(n)\}.$$

Anstatt $f(n) \in \Theta(g(n))$ wird $f(n) = \Theta(g(n))$ geschrieben; analog auch für O und Ω.

Dasselbe kann man, wie im vorangestellten Beispiel, auch mit Grenzwertbedingungen erreichen (siehe [13]), sofern diese existieren: Falls $a = \lim_{n\to\infty} \left|\frac{f(n)}{g(n)}\right|$ existiert und $0 < a < \infty$ gilt, folgt daraus $f(n) = \Theta(g(n))$. Analog gilt bei existierendem Grenzwert a und $0 < a$ bzw. $a < \infty$: $f(n) = \Omega(g(n))$ bzw. $f(n) = O(g(n))$.

Bei $f(n) = O(g(n))$ bzw. $f(n) = \Omega(g(n))$ hat man also mit Hilfe von $g(n)$ eine obere bzw. untere Schranke, bis auf einen konstanten Faktor, für die asymptotische Entwicklung der (Laufzeit-)-Funktion $f(n)$ gefunden.

1.2 Grundbegriffe der Kombinatorik

Da es in der Kombinatorik darum geht, die Anzahl von gewissen Anordnungs- und Auswahlmöglichkeiten zu ermitteln, ist folgendes Grundproblem essentiell: Wieviele Möglichkeiten gibt es, aus n Elementen k Elemente mit/ohne Zurücklegen auszuwählen, wobei die Reihenfolge eine/keine Rolle spielt? Es soll für das Folgende stets $n, k \in \mathbb{N}$ gelten.

Variation mit Wiederholung

Spielt die Reihenfolge bei der Auswahl eine Rolle, so spricht man von einer "Variation". Werden alle Elemente unmittelbar nach der Wahl wieder zurückgelegt, dann gibt es für jedes der k Elemente n mögliche Auswahlkandidaten. Daraus folgt die Gesamtanzahl der Möglichkeiten:

$$V_{n,k}^{W} = n^k.$$

Variation ohne Wiederholung

Kommen die ausgewählten Elemente nicht mehr in den Auswahl-Pool zurück, verringern sich die Möglichkeiten nach jeder Festlegung von einem der k Elemente um 1. Daher ist hier die Anzahl der Möglichkeiten:

$$V_{n,k} = n(n-1)(n-2)...(n-k+1) = \frac{n!}{(n-k)!}.$$

Definition 1.2.1 (Permutation) *Eine Permutation π der Länge n ist eine bijektive Funktion $\pi : \{1, ..., n\} \to \{1, ..., n\}$, also eine Funktion, die jedem Element der Ursprungsmenge ein anderes Element der Zielmenge zuordnet (wegen der gleichen, endlichen Mächtigkeit der beiden Mengen folgt aus der Injektivität unmittelbar die Bijektivität).*

Die Anzahl der verschiedenen Permutationen entspricht einer Variation von n Elementen aus n Elementen und beträgt somit $V_{n,n} = n!$.

Kombination ohne Wiederholung

Ist die Reihenfolge der ermittelten k Elemente unerheblich, spricht man von einer "Kombination". Eine Kombination ohne Zurücklegen erhält man von einer Variation ohne Wiederholung, indem man diese durch die Anzahl der Anordnungsmöglichkeiten der ausgewählten k Elemente, also durch die Anzahl der Permutationen der Länge k, durchdividiert:

$$K_{n,k} = \frac{n(n-1)(n-2)...(n-k+1)}{k!} = \frac{n!}{k!(n-k)!}.$$

Definition 1.2.2 (Binomialkoeffizient) *Für $n \in \mathbb{C}$ und $k \in \mathbb{N}$ ist der Binomialkoeffizient $\binom{n}{k}$ so definiert:*

$$\binom{n}{k} = \frac{n(n-1)(n-2)...(n-k+1)}{k!}.$$

Somit gilt: $K_{n,k} = \binom{n}{k}$.

Lemma 1.2.3 *Für $n \in \mathbb{C}$ und $k \in \mathbb{N}$ gilt:*

$$\binom{n+1}{k+1} = \binom{n}{k} + \binom{n}{k+1}, \quad \binom{n}{n} = \binom{n}{0} = 1.$$

Beweis. Bei $\binom{n}{0}$ ist der Dividend das leere Produkt und somit 1, also $\binom{n}{0} = \frac{1}{0!} = 1$. Für $\binom{n}{n}$ gilt $\binom{n}{n} = \frac{n!}{n!} = 1$. Die erste Gleichung kann so gezeigt werden:

$$\begin{aligned}\binom{n}{k} + \binom{n}{k+1} &= (k+1)\frac{n(n-1)\cdots(n-k+1)}{(k+1)!} + \frac{n(n-1)\cdots(n-k+1)(n-k)}{(k+1)!} \\ &= \frac{n(n-1)\cdots(n-k+1)}{(k+1)!}(k+1+n-k) = \binom{n+1}{k+1}.\end{aligned}$$ □

Kombination mit Wiederholung

Diese Problemstellung soll mit einem Beispiel erklärt werden. Für $n = 4$, $k = 6$ ist 1 – 1 – 2 – 4 – 4 – 4 eine gültige Auswahl. Diese Möglichkeit kann man auch so anschreiben: $**|*||***$. Dabei trennen die senkrechten Striche die n Kategorien voneinander ab (die Kategorien entsprechen den auszuwählenden Elementen) und die Sterne zwischen zwei dieser Striche kennzeichnen, wieviele der k Elemente aus genau dieser Kategorie ausgewählt wurden. Diese Problemstellung entspricht also einer Variation ohne Wiederholung von k Sternen und $n-1$ Strichen aus $n+k-1$ Elementen oder mit anders formuliert: Es müssen k Sterne auf die $n+k-1$ Plätze verteilt werden. Daher ergibt sich für $n, k \geq 1$:

$$K^W_{n,k} = \binom{n+k-1}{k}.$$

1.3 Grundbegriffe der Wahrscheinlichkeitsrechnung

Es werden hier nur die für die Arbeit relevanten Begriffe nach einem Skriptum von K. Felsenstein (siehe [15]) so vereinfacht wie möglich eingeführt.
Alle möglichen Resultate eines Experiments liegen in einer Menge Ω, die als Merkmalsraum bezeichnet wird. $\omega \in \Omega$ sind Elementarereignisse, während $E \subset \Omega$ ein Ereignis ist. Die Wahrscheinlichkeit $\mathbb{P}(E)$ (englisch: "probability"), dass ein Ereignis eintritt, ist definiert als die Anzahl der günstigen Ausgänge durch die Anzahl der möglichen Ausgänge, und somit eine Funktion $\mathbb{P} : \Omega \to [0, 1]$. Da vorliegender Arbeit das Model der Gleichverteilung zugrunde liegt und stets endliche Merkmalsräume betrachtet werden, gilt hier für $\Omega = \{\omega_1, ..., \omega_n\}$:

$$\mathbb{P}(E) = \frac{|E|}{|\Omega|} = \frac{|E|}{n}.$$

Führt man die Zufallsvariable (oft auch Zufallszahl oder stochastische Größe genannt) X als eine Funktion $X : \Omega \to \mathbb{R}$ ein, dann ist die Punktwahrscheinlichkeit durch

$$p_i^X = \mathbb{P}(\{\omega | X(\omega) = i\}) = \mathbb{P}(X^{-1}(i))$$

definiert. Gilt also etwa in unserem diskreten Fall $X(\omega_i) = i$, dann ist die Punktwahrscheinlichkeit durch $p_i = \mathbb{P}(X = i)$ festgelegt. Die Verteilungsfunktion $F(x)$ wird für jedes x als die Wahrscheinlichkeit des Ereignisses $E = [X \leq x]$ definiert, also:

$$F(x) = \mathbb{P}(X \leq x) = \sum_{i \leq x} p_i.$$

Die Verteilung wird wegen $p_i = F(i) - F(i-1)$ eindeutig durch die Verteilunsfunktion festgelegt. $X \sim F$ kennzeichnet, dass die Zufallszahl X nach der Verteilungsfunktion F verteilt ist. Ist X also auf $\{1, 2, ..., m\}$ gleichverteilt ($X \sim D_m$), ist wegen der Punktwahrscheinlichkeit $p_i = \frac{1}{m}$ für $i = 1, 2, ..., m$ die Verteilungsfunktion durch $F(x) = \frac{\lfloor x \rfloor}{m}$ gegeben.
Mit dem Zeichen "$\stackrel{\mathrm{d}}{=}$" wird die Gleichheit bezüglich einer Verteilung gekennzeichnet: Man schreibt also $X \stackrel{\mathrm{d}}{=} Y$, wenn die Zufallsvariablen X und Y dieselbe Verteilung besitzen. Mit $X_n \stackrel{\mathrm{d}}{\to} X$ ist gemeint, dass die Folge von Zufallsvariablen X_n in der Verteilung gegen die Zufallsvariable X konvergiert.

Definition 1.3.1 (Erwartungswert) *Der Erwartungswert (die Erwartung) einer diskreten Verteilung mit $\Omega = \{\omega_1, ..., \omega_n\}$ und den Punktwahrscheinlichkeiten p_i auf den Punkten x_i ist*

$$\mathbb{E}(X) = \sum_{\omega \in \Omega} X(\omega)\ \mathbb{P}(\omega) = \sum_{i=1}^{n} x_i\ p_i.$$

Der Erwartungswert wird auch als das erste Moment bezeichnet. Höhere Momente sind Erwarungswerte von X^k; das k-te Moment ist daher: $\mathbb{E}(X^k)$.

Definition 1.3.2 (Varianz) *Die Varianz einer Zufallszahl X mit endlichem zweiten Moment, also $\mathbb{E}(X^2) < \infty$, ist*

$$\mathbb{V}(X) = \mathbb{E}\left(X - \mathbb{E}(X)\right)^2.$$

Satz 1.3.3 (Steiner'scher Verschiebungssatz) *Für $\mathbb{E}(X^2) < \infty$ gilt:*

$$\mathbb{V}(X) = \mathbb{E}(X^2) - \left(\mathbb{E}(X)\right)^2.$$

Beweis. Aufgrund der Linearität (eine Funktion $f(x)$ ist genau dann linear, wenn für α konstant $f(\alpha x + y) = \alpha f(x) + f(y)$ gilt) des Erwartungswertes ergibt sich:

$$\mathbb{V}(X) = \mathbb{E}(X - \underbrace{\mathbb{E}(X)}_{=\mu})^2 = \mathbb{E}(X^2 - 2X\mu + \mu^2) = \mathbb{E}(X^2) - 2\mu \underbrace{\mathbb{E}(X)}_{=\mu} + \mu^2 = \mathbb{E}(X^2) - \overbrace{\left(\mathbb{E}(X)\right)^2}^{=\mu^2}.$$

□

1.4 Summationsformeln

Hier sollen geschlossene Formeln für einige bekannte Summen gezeigt werden, die im Laufe der Arbeit benötigt werden.

1.4.1 Reihen mit Binomialkoeffizienten

Lemma 1.4.1 *Für $n, m \in \mathbb{N}$ gilt:*

$$\sum_{k=0}^{n} \binom{n-k}{m} = \binom{n+1}{m+1}.$$

Beweis. Für $m = 0$ gilt die Formel wegen $\binom{n+1}{1} = n+1$ und Lemma 1.2.3, woraus für $0 \leq k \leq n$ die Gültigkeit von $\binom{n-k}{0} = 1$ ersichtlich ist. Im Folgenden kann also $m \geq 1$ angenommen werden. Prinzipiell können alle diese hier angeführten Summenformeln mit vollständiger Induktion gelöst werden. Wählt man als Induktionsanfang $n = 0$ so erhält man wegen $\binom{a}{b} = 0$ für $a < b \in \mathbb{N}$ (siehe Definition 1.2.2):

$$\binom{0}{m} = \binom{1}{m+1} = \begin{cases} 1, & m = 0, \\ 0, & \text{sonst}, \end{cases}$$

die geforderte Gleichheit. Anschließend kann im Induktionsschritt die Gültigkeit der Aussage für n als Induktionsvoraussetzung (Ind.Vor.) angenommen werden. Daraus soll jetzt auch die Richtigkeit von $n+1$ gezeigt werden:

$$\begin{aligned} \sum_{k=0}^{n+1} \binom{(n+1)-k}{m} &= \sum_{k=0}^{n} \binom{n-k+1}{m} + \overbrace{\binom{0}{m}}^{=0 \text{ für } m \geq 1} \overset{m \geq 1}{=} \sum_{k=0}^{n} \left[\binom{n-k}{m} + \binom{n-k}{m-1} \right] \\ &\overset{\text{Ind.Vor.}}{=} \binom{n+1}{m+1} + \binom{n+1}{m} = \binom{(n+1)+1}{m+1}. \end{aligned}$$

□

Lemma 1.4.2 *Für $m \in \mathbb{N}$ gilt:*

$$\sum_{k=0}^{n} \binom{k+m}{k} = \binom{n+m+1}{n}.$$

Beweis. Für $n = 0$ gilt wegen Lemma 1.2.3 die Gleichung $\binom{0+m}{0} = \binom{0+m+1}{0} = 1$. Gilt nun die Gleichung für n, so folgt wieder mit Lemma 1.2.3 für $n+1$:

$$\begin{aligned} \sum_{k=0}^{n+1} \binom{k+m}{k} &= \sum_{k=0}^{n} \binom{k+m}{k} + \binom{n+1+m}{n+1} \overset{\text{Ind.Vor.}}{=} \binom{n+m+1}{n} + \binom{n+1+m}{n+1} \\ &= \binom{(n+1)+m+1}{n+1}. \end{aligned}$$

□

1.4.2 Arithmetische Reihen

Als "arithmetische Reihe" wird eine Reihe bezeichnet, die für je zwei aufeinanderfolgende Summanden eine konstante Differenz d hat. Für $n \in \mathbb{N} \cup \{\infty\}$ ist also $\sum_{j=1}^{n} a_j$ eine arithmetische Reihe wenn $a_{j+1} - a_j = d$ für alle $j \in \mathbb{N}$ und d konstant gilt.

Lemma 1.4.3 (Gauß'sche Summenformel) *Für $n \in \mathbb{N}$ gilt:*

$$\sum_{j=1}^{n} j = \frac{n(n+1)}{2} = \binom{n+1}{2}.$$

Beweis. Folgende Beweisidee geht auf Gauß zurück und bedient sich der Vorwärts- und Rückwärtssummation:

$$2\sum_{j=1}^{n} j = \overbrace{1+2+...+n}^{\text{erste Summe } S_1} + \overbrace{n+(n-1)+...+1}^{\text{zweite Summe } S_2} = \underbrace{\overbrace{1}^{S_1}+\overbrace{n}^{S_2}}_{=(n+1)} + \underbrace{\overbrace{2}^{S_1}+\overbrace{(n-1)}^{S_2}}_{=(n+1)} + ... + \underbrace{\overbrace{n}^{S_1}+\overbrace{1}^{S_2}}_{=(n+1)}$$
$$= n(n+1).$$

Man kann sich leicht überlegen, dass diese Vorgehensweise sowohl bei n gerade, als auch bei n ungerade funktioniert. □

1.4.3 Potenzreihen

Wie der Name schon vermuten lässt, sind Potenzreihen Reihen, deren Summanden die Form $c_j(x-x_0)^j$ haben, wobei alle c_j und die Entwicklungsstelle x_0 konstant sind.

Lemma 1.4.4 *Für $n \in \mathbb{N} \setminus \{0\}$ gilt:*

$$\sum_{j=1}^{n} j^2 = \frac{n(n+1)(2n+1)}{6}.$$

Beweis. Als Induktionsanfang wird die Aussage also für $n = 1$ verifiziert: $\frac{1(1+1)(2+1)}{6} = 1$. Im Induktionsschritt wird die Gültigkeit der Aussage für n angenommen und soll nun für $n+1$ gezeigt werden:

$$\sum_{j=1}^{n+1} j^2 = \sum_{j=1}^{n} j^2 + (n+1)^2 = \frac{n(n+1)(2n+1)}{6} + \frac{6(n+1)^2}{6} = \frac{(n+1)(2n^2+n+6n+6)}{6}$$
$$= \frac{(n+1)(n+2)(2n+3)}{6} = \frac{(n+1)((n+1)+1)(2(n+1)+1)}{6}.$$ □

1.4.4 Geometrische Reihen

Als "geometrische Reihe" wird eine Reihe bezeichnet, die für je zwei aufeinanderfolgende Summanden einen konstant gleichbleibenden Quotienten q hat. Für $n \in \mathbb{N} \cup \{\infty\}$ ist also $\sum_{j=0}^{n} a_j$ eine arithmetische Reihe wenn $\frac{a_{j+1}}{a_j} = q$ für alle $j \in \mathbb{N}$ gilt.

Lemma 1.4.5 (geometrische Reihe) *Für $x \neq 1$ und $n \in \mathbb{N}$ gilt:*

$$\sum_{j=0}^{n} x^j = \frac{1-x^{n+1}}{1-x}.$$

Beweis. Bezeichnet man mit $S_n = \sum_{j=0}^{n} x^j$ die n-te Partialsumme, dann gilt:

$$(1-x)S_n = S_n - xS_n = (1+x+x^2+...+x^n) - (x+x^2+x^3+...+x^{n+1}) = 1-x^{n+1},$$
$$S_n = \frac{1-x^{n+1}}{1-x}.$$ □

Bemerkung 1.4.6 *Bildet man für $|x| < 1$ den Grenzwert für $n \to \infty$, so erhält man wegen $\lim_{n\to\infty} x^{n+1} = 0$:*

$$\sum_{j=0}^{\infty} x^j = \frac{1}{1-x}.$$

Für $|x| > 1$ divergiert die Summe und ist $x = 1$ so hat die n-te Partialsumme den Wert $n + 1$.

Lemma 1.4.7 (potenzierte geometrische Reihe, Binomische Reihe) *Für $|x| < 1$ gilt:*

$$\frac{1}{(1-x)^{m+1}} = \sum_{n\geq 0} \binom{n+m}{n} x^n,$$

Beweis (entnommen von [5]). Hier wird wieder ein Induktionsbeweis geführt. Der Spezialfall $m = 0$ stellt wegen $\binom{n}{n} = 1$ die Gleichung für die geometrische Reihe aus Lemma 1.4.5 dar und ist somit gültig. Um von m auf $m + 1$ schließen zu können, wird mit Hilfe der Cauchy-Produktformel und des Lemmas 1.4.2 für die letzte Umformung so vorgegangen:

$$\begin{aligned}\frac{1}{(1-x)^{(m+1)+1}} &\overset{\text{Ind.Vor.}}{=} \sum_{n\geq 0} \binom{n+m}{n} x^n \cdot \frac{1}{1-x} = \sum_{n\geq 0} \binom{n+m}{n} x^n \cdot \sum_{n\geq 0} x^n \\ &= \sum_{n\geq 0} \sum_{k=0}^{n} \binom{k+m}{k} x^k x^{n-k} = \sum_{n\geq 0} \binom{n+m+1}{n} x^n. \end{aligned}$$ □

1.5 Harmonische Zahlen

In vorliegender Arbeit sind die harmonischen Zahlen ein wichtiger Ausdruck, der immer wieder Verwendung findet.

Definition 1.5.1 (Harmonische Zahlen) *Die endliche Summe ($n \in \mathbb{N} \setminus \{0\}$)*

$$H_n = \sum_{j=1}^{n} \frac{1}{j} \quad \textit{bzw.} \quad H_n^{(k)} = \sum_{j=1}^{n} \frac{1}{j^k} \quad \textit{für } k \in \mathbb{N}, k \geq 2$$

wird als die n-te harmonische Zahl bzw. die n-te harmonische Zahl zur Ordnung k bezeichnet.

Da harmonische Zahlen oft in Laufzeitanalysen vorkommen, ist es wichtig zu wissen, wie sich H_n asymptotisch verhält. Das Folgende wurde aus [13] zusammengefasst.

Satz 1.5.2 $H_n = O(\log n)$.

Beweis. Die Idee ist, dass man die endliche Summe $\sum_{a\leq k<b} f(k)$ für $a, b \in \mathbb{N}$ mit $\int_a^b f(x)dx$ annähern kann. Dabei muss der Approximationsfehler abgeschätzt werden. Wird das Intervall [a,b) in lauter Einheitsintervalle (Intevalle der Länge 1) unterteilt, so kann der maximale Fehler für das k-te dieser Intervalle mit

$$\delta_k = \max_{k\leq x<k+1} |f(x) - f(k)|$$

nach oben abgeschätzt werden. Da es sich bei den Harmonischen Zahlen um eine monoton fallende Funktion handelt, erhält man beim Aufsummieren dieser Fehler eine Teleskopsumme und für den Gesamtfehler bleibt nur mehr der erste und letzte Summand übrig:

$$H_n = \sum_{k=1}^{n} \frac{1}{k} \leq \int_1^n \frac{1}{x} dx + \sum_{1 \leq k < n} \delta_k = \log n + 1 - \frac{1}{n} = O(\log n). \qquad \square$$

Bemerkung 1.5.3 *Die genaue Abschätzung für die harmonischen Zahlen bekommt man mit Hilfe der diskreten Form der Euler-Maclaurin Summation (siehe [13, Kapitel 4]). Damit erhält man* $H_n = \log n + \gamma + \frac{1}{2n} + O(\frac{1}{n^2})$*, wobei* $\gamma = 0{,}57721...$ *die Euler-Mascheroni-Konstante ist.*

Aus der Analysis ist auch bekannt, dass die allgemeine harmonische Reihe $\sum_{k \geq 1} \frac{1}{k^\alpha}$ mit $\alpha \in \mathbb{R}$ für alle $\alpha > 1$ konvergiert (etwa für $\alpha = 2$: $\sum_{k \geq 1} \frac{1}{k^2} = \frac{\pi^2}{6}$), während bei $\alpha \leq 1$ Divergenz herrscht.

H_n kann auch als eine Funktion mit Argument n aufgefasst werden. Laut Definition sind bisher nur $n \in \mathbb{N} \setminus \{0\}$ zugelassen. Nun kann man aber einen Schritt weiter gehen und auch komplexe Werte zulassen ([17]).

Definition 1.5.4 (Verallgemeinerung der harmonischen Zahlen)

$$H(x) = \sum_{k \geq 1} \left(\frac{1}{k} - \frac{1}{x+k} \right) \qquad \textit{für} \quad x \in \mathbb{C} \setminus \{0, -1, -2, ...\}.$$

Für $n \in \mathbb{N} \setminus \{0\}$ gilt $H(n) = H_n$, da sich in der Summe alle Terme $\frac{1}{k}$ mit $k > n$ wegheben.

1.6 Erzeugende Funktionen

Erzeugende Funktionen sind ein zentrales Hilfsmittel bei der Analyse von Algorithmen. Es können mit ihnen nicht nur effizient Rekursionen gelöst und Momente berechnet werden – sie sind vielmehr (siehe [13, Kapitel 3]) eine natürliche Verbindung zwischen den Algorithmen und den analytischen Methoden, um die Eigenschaften der Algorithmen zu untersuchen. Erzeugende Funktionen sind sowohl ein kombinatiorisches Werkzeug um Abzählungen vorzunehmen, als auch ein analytisches Werkzeug, um aussagekräftige Abschätzungen tätigen zu können.

Definition 1.6.1 (Erzeugende Funktion) *Es sei die Folge* $a_0, a_1, ..., a_k, ...$ *gegeben. Die Funktion*

$$A(z) = \sum_{k \geq 0} a_k z^k$$

wird die (gewöhnliche) erzugende Funktion der Folge genannt. Um den k-ten Koeffizienten aus $A(z)$ *extrahieren zu können, verwendet man die Funktion* $[z^k]$*, also:* $[z^k]A(z) = a_k$*.*

Definition 1.6.2 (Wahrscheinlichkeitserzeugende Funktion) *Sei* X *eine Zufallsvariable, die nur natürliche Zahlen annehmen kann. Mit der Punktwahrscheinlichkeit* $p_k = \mathbb{P}\{X = k\}$ *wird die Funktion*

$$P(z) = \sum_{k \geq 0} p_k z^k$$

wahrscheinlichkeitserzugende Funktion für die Zufallszahl X *genannt.*

Man erkennt sofort, dass die Bedingung $P(1) = 1$ notwendig und hinreichend für: $P(z) = \sum_{k\geq 0} p_k z^k$ ist eine wahrscheinlichkeitserzeugende Funktion, ist.

Satz 1.6.3 *Mit einer wahrscheinlichkeitserzugenden Funktion $P(z)$ zu der Zufallsvariable X kann man den Erwatrungswert und die Varianz derart berechnen:*

$$\mathbb{E}(X) = P'(1), \quad \mathbb{V}(X) = P''(1) + P'(1) - P'(1)^2.$$

Beweis.

$$P'(1) = \left(\sum_{k\geq 0} p_k z^k\right)'\Bigg|_{z=1} = \sum_{k\geq 0} k p_k z^{k-1}\Bigg|_{z=1} = \sum_{k\geq 0} k p_k = \mathbb{E}(X),$$

$$\mathbb{V}(x) = \sum_{k\geq 0}(k - P'(1))^2 p_k = \sum_{k\geq 0} k^2 p_k - 2P'(1)\underbrace{\sum_{k\geq 0} k p_k}_{=P'(1)} + P'(1)^2 \underbrace{\sum_{k\geq 0} p_k}_{=1} = \sum_{k\geq 0} k^2 p_k - P'(1)^2$$

$$= \sum_{k\geq 0} k(k-1)p_k + \sum_{k\geq 0} k p_k - P'(1)^2 = P''(1) + P'(1) - P'(1)^2. \qquad \square$$

1.7 Baumstruktur

Hier sollen nun ein paar Grundbegriffe der Graphentheorie aus [19] eingeführt werden.

Definition 1.7.1 (Graph) *Ein Graph (V, E) besteht aus einer endlichen Knotenmenge V und einer Kantenmenge E. Bei einem gerichteten Graphen gilt $E \subseteq V \times V$, während bei einem ungerichteten Graphen $E \subseteq K : K = \{\{a, b\} : a, b \in V, a \neq b\}$ gilt. Sind Mehrfachkanten zugelassen spricht man von einem Multigraphen.*

Definition 1.7.2 (Pfad, Kreis, Zusammenhang) *In einem (gerichteten) Graph (V, E) wird eine durch (gerichtete) Kanten verbundene nichttriviale Knotenfolge (Auswahl der Folgenelemente > 1) mit lauter unterschiedlichen Knotenelemente als Pfad bezeichnet – die Anzahl der Kanten zwischen diesen Knoten ist die Länge des Pfades.*
Ein Kreis ist ein Pfad $(v_1, ..., v_n)$, bei dem zusätzlich v_n durch eine (gerichtete) Kante mit v_1 verbunden ist: $(v_1, ..., v_n, v_1)$. Im ungerichteten Graph gilt zusätzlich $n > 2$.
Ein ungerichteter Graph ist zusammenhängend, wenn für je zwei Knoten ein Pfad gefunden werden kann, der diese beiden Knoten verbindet. Im gerichteten Graph spricht man in diesem Fall von stark zusammenhängend; schwach zusammenhängend ist ein gerichteter Graph genau dann, wenn der entsprechende Graph, wo die Orientierung weggelassen wird, zusammenhängend ist.

Definition 1.7.3 (Wurzelaum) *Ein Wurzelaum ist ein kreisfreier, schwach zusammenhängender gerichteter Graph, mit einem ausgezeichneten Wurzelknoten, den jeder Knoten durch einen Pfad erreichen kann. Der Startpunkt einer jeden Kante in Richtung zum Wurzelknoten wird als eines der Kinder vom Endpunkt der Kante bezeichnet, der auch als Elternknoten bezeichnet wird. Ein Knoten ohne Kinder heißt Blatt.*

1.7.1 Binärer Suchbaum

Ein binärer Suchbaum ist eine abstrakte Datenstruktur in der Informatik. Zuerst soll der Begriff Binärbaum erklärt werden.

Definition 1.7.4 (Binärbaum) *Ein Binärbaum ist ein Wurzelaum, bei der jeder Knoten höchstens zwei Kinder haben darf. Dabei spricht man jeweils vom rechten und vom linken Kind.*

Definition 1.7.5 (Binärer Suchbaum) *Ein Binärer Suchbaum ist ein Binärbaum B, wo für jeden Knoten $x \in B$ gilt, dass alle Knoten im linken Teilbaum einen kleineren Wert als x haben, während alle Knoten im rechten Teilbaum einen Wert größer als oder gleich wie x haben.*

Beispiel 1.7.6 (Binärer Suchbaum) *Die Werte* 2, 5, 7, 15, 22, 29, 33, 35, 41 *können in einem binären Suchbaum beispielsweise derart dargestellt werden:*

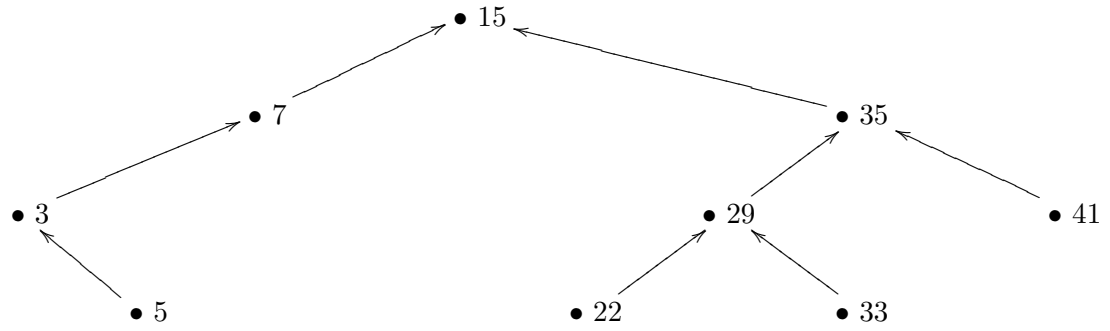


Abbildung 1.1: Ein Beispiel für einen binären Suchbaum

Hier sind die Knoten 5, 22, 33 *und* 41 *die Blätter des Baumes, während der Knoten* 15 *die ausgezeichnete Wurzel ist. Die Werte des gesamten linken Teilbaums von* 35*, also* 22, 29 *und* 33*, sind kleiner als* 35*, während die des rechten Teilbaums, hier* 41*, größer sind. Dies gilt für alle Elemente.*

1.8 Analyse von Algorithmen

Einer der ersten mathematischen Zugänge war jener von Knuth (siehe [2]). Er formulierte zwei Analyse-Typen, die im folgenden kurz vorgestellt werden.

1.8.1 Typ A: Analyse eines speziellen Algorithmus

Bei einer Typ A Analyse werden die wesentlichen Merkmale eines Algorithmus untersucht. Dazu zählen eine Laufzeitanalyse (englisch: “frequency analysis”), wo herausgefunden wird, wie oft jeder Teil des Algorithmus aufgerufen und abgearbeitet werden muss. Der ungünstigste Fall, im folgenden mit worst-case bezeichnet, gibt eine obere Schranke und eine – meist grobe – Abschätzung nach oben für das Laufzeitverhalten. Das Gegenstück dazu ist der günstigste Fall, der best-case, der also die best mögliche und damit niedrigste Anzahl an Berechnungsschritten benötigt. Der eigentlich interessante Wert ist aber die Durchschnittsanalyse, also die Ermittlung des avarage-case. Hier wird üblicherweise der Erwartungswert und, wenn möglich, auch die Varianz der benötigten Berechnungsschritte ermittelt.
Ein weiteres wichtiges Gütekriterium eines Algorithmus ist sein zusätzlich zur verarbeitenden Datenmenge benötigter Hilfsspeicher, also eine Speicherplatzanalyse (englisch: “storage analysis”).

1.8.2 Typ B: Analyse einer Klasse von Algorithmen

Im Gegensatz zur Typ A Analyse beschäftigt man sich bei einer Typ B Analyse mit einer ganzen Familie von Algorithmen, die ein bestimmtes Problem lösen und möchte den best möglichen ("best possible") darunter identifizieren. Man kann bei der Betrachtung einer ganzen Algorithmenfamilie obere bzw. untere Schranken für die Komplexität des Problems bestimmen. Wenn man beispielsweise das Sortierproblem für n Zahlen betrachtet, kann man die minimale Anzahl an Vergleichen $S(n)$ ermitteln, die in jedem Fall benötigt werden.

Auf den ersten Blick scheint die zeitlich erst später entwickelte Typ B Analyse der Typ A Analyse überlegen zu sein, da es hier möglich ist, unendlich viele Algorithmen gleichzeitig behandeln zu können. Denn natürlich ist man daran interessiert, anstatt jeden einzelnen erfundenen Algorithmus zu anaylsieren, gleich den für eine Problemstellung best möglichen zu ermitteln und zu beweisen, dass es keinen besseren mehr geben kann. Diesem Argument hält Knuth dagegen, dass Typ B Analysen extrem Technologie-abhängig sind: Kleine Änderungen in der Definition für "best possible" können große Auswirkungen darauf haben, welcher Algorithums der Familie nun tatsächlich der bestgeeignete ist. Als Beispiel nennt Knuth, dass x^{31} in nicht weniger als 9 Multiplikationen berechnet werden kann (die Zwischenergebnisse werden mitgespeichert):

$$x^{31} = \left(\left(\left(x^2\right)^2\right)^2\right)^2 \cdot \left(\left(x^2\right)^2\right)^2 \cdot \left(x^2\right)^2 \cdot x^2 \cdot x.$$

Lässt man aber auch Divisionen zu, kann x^{31} auch mit 6 arithmetischen Operationen berechnet werden:

$$x^{31} = \frac{\left(\left(\left(x^2\right)^2\right)^2\right)^2}{x}.$$

Das Problem bei Typ B Analysen ist auch, dass man oft ein sehr einfaches Model der Komplexität verwenden muss, um überhaupt zu Aussagen zu bekommen. Dabei ist die Gefahr groß, dass der zu hohe Abstraktionsgrad aller relevanter Aspekte einer betrachteten Klasse von Algorithmen zu unrealistischen und unpraktikablen Lösungen führen kann. Des weiteren ist eine Typ B Analyse oft äußerst schwierig durchzuführen, sodass man oft keine brauchbaren Resultate erhält.
Daraus schließt Knuth, dass Typ A Analysen wahrscheinlich wichtiger als Typ B Analysen sind. Er bemerkt auch, dass glücklicherweise auch Typ A Analysen eine intellektuelle Herausforderung darstellen, da fast jeder Algorithmus, der nicht übertrieben kompliziert ist, zu interessanten mathematischen Fragestellungen führt. Die folgenden Seiten sollen diese Aussage zumindest für eine Auswahl von In situ-Permutationsalgorithmen verifizieren.

Kapitel 2

Einführung

2.1 Problemstellung

Diese Arbeit beschäftigt sich mit Algorithmen, welche die Einträge eines gegebenen Datenfeldes gemäß einer vorgegebenen Permutation umordnen und dabei weniger als $O(n)$ Bits zusätzlichen Hilfsspeicher benötigen.
Im Konkreten sei also für $n \in \mathbb{N}$ ein Array $A = [a_1, a_2, ..., a_n]$ und eine Permutation $\pi : \{1, ..., n\} \to \{1, ..., n\}$ gegeben. Führt man jetzt gleichzeitig die n Operationen

$$A[i] \leftarrow A[\pi(i)] \text{ für } i \in \{1, ..., n\}$$

aus, dann kommt man zum gewünschten Resultat. Man versucht also den Inhalt von A, nämlich $(a_1, a_2, ..., a_n)$, in die Reihenfolge $(a_{\pi(1)}, a_{\pi(2)}, ..., a_{\pi(n)})$ umzuwandeln.

2.2 Motivation

Die Untersuchung von solchen Algorithmen ist nicht nur von theoretischem Interesse. In der Praxis werden Permutationsalgorithmen bei einer Vielzahl von Problemen und Subproblemen eingesetzt (siehe [1]). Hier eine kleine Auswahl:

- Rotation von digitalen Bildern
- Transponieren einer Matrix
- Bestandteil gewisser Sortierfunktionen
- Anlegen von Suchbäumen

Die ersten beiden Punkte liegen auf der Hand: Hier soll keine Information abhanden kommen, sondern lediglich nach einem vorgegebenen Schema neu arrangiert werden.
Es greifen auch gewisse Sortierfunktionen auf Permutationsalgorithmen zurück. Wenn man beispielsweise große Datensätze hat, kann es sinnvoll sein, vorerst nur mit Pointern die Sortierung zu erarbeiten und anschließend demenstprechend die Datensätze umzuordnen. Oder man sortiert, indem man die einzelnen Datensätze paarweise vergleicht, für jeden einzelnen zählt, wieviele vor ihm stehen müssen und abschließend dementsprechend umordnet, wo wieder ein Permutationsalgorithmus zum Tragen kommt.
Will man auf einem geordneten Datenfeld vermehrt Datensätze suchen, ist es sinnvoll, diese Daten entsprechend eines Suchbaumes neu anzuordnen. Wählt man einen binären Suchbaum – in Abbildung 2.1 die zweite Zeile – befinden sich für ein Element an der i-ten Stelle die Kinder

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

8	4	12	2	6	10	14	1	3	5	7	9	11	13	15

4	8	12	1	2	3	5	6	7	9	10	11	13	14	15

Abbildung 2.1: Anordnungsmöglichkeiten von Daten

auf den Feldern mit Index $2i$ und $2i + 1$. Es kann auch erwünscht sein, die Daten so neu zu ordnen, dass $(b + 1)$-äre Suchbäume implizit angelegt werden – die dritte Zeile zeigt den Fall $b = 3$. Das ist von Vorteil, wenn man in den virtuellen Speicher gleichzeitig b Elemente einlesen kann.

Lässt man die Forderung mit dem begrenzten Hilfsspeicher echt kleiner als $O(n)$ fallen, ist es nicht schwierig, Algortihmen zu finden, die das gestellte Problem in linearer Laufzeit lösen können. Ein Ansatz ist, dass man das Hilfsfeld $(\pi(1), \pi(2), ..., \pi(n))$ anlegt und erlaubt, darauf lesend und schreibend zugreifen zu können. Ein anderer Ansatz speichert in n extra Bits wie weit man in der vorliegenden Permutation schon vorangeschritten ist. Darauf wird später noch eingegangen.
Aufgrund der Forderung schränken sich die Lösungsmöglichkeiten allerdings sehr stark ein. Eine sehr natürliche Vorgehensweise wurde von J. C. Gower gefunden, dessen “Klassischer Algorithmus” im Folgenden untersucht wird. Der Erste, der sich mit der Analyse dieses Algorithmus beschäftigt hat, war D. E. Knuth (siehe [2]), dessen Ergebnisse als erstes hier im folgenden Kapitel vorgestellt werden.

Kapitel 3

Der klassische Algorithmus

3.1 Zyklendarstellung einer Permutation

Jede Permutation kann man als das Produkt ihrer einzelnen Zyklen anschreiben. Dabei beinhaltet ein Zyklus mit jedem Element auch dessen Bild unter der Permutation π.

Definition 3.1.1 (Zyklus) *Der Zyklus von j sei die Menge jener Elemente, die man durch wiederholtes Anwenden der Permutationsabbildung erhält:*

$$\text{Zyklus}(j) := \{\pi^k(j) \mid k \in \mathbb{N}\}.$$

Die Darstellung einer Permutation in Zyklenschreibweise ist nicht eindeutig. Zum einen kann man die Elemente von jedem Zyklus beliebig zyklisch vertauschen, also zum Beispiel (2 1 3) = (1 3 2) = (3 2 1). Zum anderen ist die Reihenfolge der einzelnen Zyklen nicht festgelegt, also etwa (1 3)(2 5 4) = (2 5 4)(1 3). Um die Eindeutigkeit, die für die algorithmische Abarbeitung essentiell ist, zu erreichen, wird pro Zyklus ein Element, genannt Zyklenführer, ausgezeichnet. Der Zyklenführer soll nun das kleinste Element eines jeden Zyklus sein.

Definition 3.1.2 (Zyklenführer) *Man nennt j genau dann einen Zyklenführer wenn für alle $k \in \mathbb{N}$ gilt:*

$$j \leq \pi^k(j).$$

Jetzt erreicht man die eindeutige Darstellung, indem man die Zyklen nach der Größe der Zyklenführer ordnet und jeden Zyklus mit dem Zyklenführer beginnen lässt.

Definition 3.1.3 (Kanonische Zyklendarstellung) *Ordnet man die Zyklen einer Zyklendarstellung zu einer gegebenen Permutation π absteigend nach der Größe der Zyklenführer und stellt man jedem Zyklus den Zyklenführer voran, erhält man die kanonische Zyklendarstellung der Permutation π.*

Beispiel 3.1.4 *Sei die Permutation π in der einzeiligen Darstellung gegeben:*

$$\pi = (\pi(1), \pi(2), ..., \pi(6)) = (5, 1, 3, 6, 2, 4)$$

Um jetzt die zyklische Darstellung zu erhalten, kann man zur zweizeiligen Darstellung übergehen, wo man leichter sieht, welche Elemente in welche übergehen, nämlich die oberen in die unmittelbar darunterliegenden:

$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 5 & 1 & 3 & 6 & 2 & 4 \end{pmatrix}$$

Wenn man jetzt bei 1 *startet und sich entlang des Zyklus bewegt, bekommt man den Zyklus* (1 5 2). 2 *kommt bereits in einem Zyklus vor, also geht es mit* 3 *weiter. Am Ende erhält man drei Zyklen, die man nun noch absteigend nach Zyklenführer ordnet, um so zur kanonischen Zyklendarstellung zu gelangen:*

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

Hier kann man jetzt auch auf die Klammern verzichten, da die Permutation auch aus der Zahlenfolge 4 6 3 1 5 2 *rekonstruierbar ist. Das Zeichen ")(" kommt nämlich genau dann vor, wenn die nachfolgende Zahl kleiner ist, als alle vorhergehenden.*

3.2 Algorithmus nach J. C. Gower

Folgender Algorithmus wurde von Knuth (siehe [2]) herangezogen, um eine Typ A Analyse exemplarisch durchzuführen. Dabei wird die Zyklenstruktur der Permutation ausgenützt.

Algorithmus 3.2.1

```
 1  for j := 1 to n do          ▷ 1
 2    k := π(j)                 ▷ n
 3    while k > j do            ▷ n + a
 4      k := π(k)               ▷ a
 5    end while                 ▷ a
 6    if k = j then             ▷ n
 7      element := A[j]         ▷ b
 8      m := π(j)               ▷ b
 9      while m ≠ j do          ▷ b + c
10        A[k] := A[m]          ▷ c
11        k := m                ▷ c
12        m := π(k)             ▷ c
13      end while                ▷ c
14      A[k] := element          ▷ b
15    end if                    ▷ b
16  end for                     ▷ n
```

3.3 Analyse

3.3.1 Korrektheit

Der erste Schritt einer jeden Analyse ist es, dass man überprüft, ob der Algorithmus auch das Gewünschte leistet.
In den Zeilen 2 bis 6 wird überprüft, ob das jeweilig betrachtete j ein Zyklenführer ist. Ist dies der Fall, muss die Bedingung in Zeile 3 für alle anderen Elemente des Zyklus gelten und somit der gesamte Zyklus durchschritten werden, um am Ende wieder beim Zyklenführer anzukommen. Somit ist die Bedingung in Zeile 6 im Fall "j ist Zyklenführer" erfüllt und es wird anschließend in den Zeilen 7 bis 14 die Umordnung mit einem Hilfsspeicherplatz "*element*" durchgeführt. Dabei überschreibt man sukzessive die entsprechenden Einträge des Arrays A in der Reihenfolge des Zyklus.

3.3.2 Laufzeit

Nun stellt sich die Frage, wie oft man - in Abhängigkeit von n, der Größe des Arrays - jede Zeile im Code abarbeiten muss. Dazu genügt es, vorerst drei Variablen einzuführen, nämlich a, b und c:
Es ist klar, dass man die Zeilen 2, 6 und 16 für jedes Element genau einmal abarbeitet, also insgesamt n-mal. Für die Zeile 3 bemüht Knuth die Kirchhoff'sche Regel, die besagt, dass die Anzahl, wie oft man zu einer Stelle im Programm kommt, gleich der Anzahl ist, wie oft man sie verlässt. Belegt man die Abarbeitung der Zeile 4 mit der Variablen a, dann folgt für die Zeile 3, dass sie $(n+a)$-mal ausgeführt wird. Eine analoge Überlegung führt zu $b+c$ in Zeile 9.

Nun zur Interpretation der Variablen. Die Variable b ist offensichtlich die Anzahl der Zyklen der gegebenen Permutation. In der kanonischen Zyklendarstellung $\pi = (q(1), q(2), ..., q(n))$ sind das also die links-rechts Minima. Es gilt also:

$$b(\pi) = |\{\, j : \quad 1 \leq j \leq n \quad \wedge \quad q(j) = \min\{q(i) \mid 1 \leq i \leq j\}\}| \,.$$

Weiters stellt man fest, dass jedes Element entweder in der Zeile 10 oder in der Zeile 14 umgespeichert wird. Daher ergibt sich der Zusammenhang: $c + b = n$.

Bemerkung 3.3.1 *Fixpunkte werden vom Algorithmus nicht explizit erkannt und als einelementige Zyklen gehandhabt, das heißt, dass Zeile 14 einmalig ausgeführt wird.*

Die Variable a ist etwas schwieriger zu interpretieren. Geht man wieder von der kanonischen Zyklendarstellung $\pi = (q(1), q(2), ..., q(n))$ aus, gilt

$$a(\pi) = |\{\, (i,j) : \quad 1 \leq i < j \leq n \quad \wedge \quad q(i) < q(i+1),\, q(i) < q(i+2), ..., q(i) < q(j)\}|. \tag{3.1}$$

worst-case

Das worst-case Szenario, also den denkbar ungünstigsten Fall, analysiert man, um eine obere Schranke für die mögliche Laufzeit zu bekommen.
In unserem Fall nimmt a den maximalen Wert bei $\pi = (\pi(1), \pi(2), ..., \pi(n)) = (2, ..., n, 1)$ an, nämlich $a = (n-1) + (n-2) + ... + 0 = \frac{1}{2}(n^2 - n)$. In diesem Fall hat die Permutation lediglich einen Zyklus und somit b den Wert 1, was der "best case" für b ist.
Bei der Permutation $\pi = (1, 2, ..., n)$, die aus lauter Fixpunkten besteht, wird b maximal, nämlich $b = n$. Hier wird a minimal, nämlich $a = 0$.

average-case

Interessanter und aussagekräftiger ist in der Regel die Durchschnittsanalyse. Zuerst muss man sich hierbei überlegen, welche Verteilung die Input-Daten haben. In unserem Fall nehmen wir an, dass alle möglichen $n!$ Permutationen gleich wahrscheinlich sind. Im Folgenden soll der Erwartungswert und die Varianz der Variablen a und b berechnet werden.

Satz 3.3.2 *Der Erwartungswert von b ist gegeben durch:*

$$\mathbb{E}(b) = H_n - 1 = O(\log(n)).$$

Definition 3.3.3 (Stirlingzahl 1.Art) *Die Stirlingzahl 1.Art $s_{n,k}$ ist die Anzahl der unterschiedlichen Möglichkeiten, eine Permutation mit n Elementen in k Zyklen zu zerlegen.*

Für die Stirlingzahl 1. Art gilt folgende Rekursion:

$$s_{n,k} = s_{n-1,k-1} + (n-1)s_{n-1,k}, \qquad s_{n,0} = \delta_{n,0}, \quad s_{0,k} = \delta_{0,k}.$$

Der erste Summand steht für die Möglichkeit, dass n ein Fixpunkt ist und somit einen eigenen Zyklus bildet. Der zweite Summand deckt die Fälle $\pi(n) \neq n$ ab.

Beweis. Bezeichnet man jetzt mit $P_{n,k}$ die Wahrscheinlichkeit, dass eine Permutation der Länge n genau k Zyklen hat, kommt man durch obige Überlegung zu der Rekursion

$$P_{n,k} = \frac{1}{n} P_{n-1,k-1} + \frac{n-1}{n} P_{n-1,k}. \tag{3.2}$$

Sei $G_n(z) := \sum_{k\geq 0} P_{n,k} z^k$ für $n \geq 1$ die wahrscheinlichkeitserzeugende Funktion. Multipliziert man Gleichung (3.2) mit z^k und summiert über alle $k \geq 0$ auf, erhält man

$$\begin{aligned} G_n(z) &= \frac{1}{n} \sum_{k\geq 0\,(1)} P_{n-1,k-1} z^k + \frac{n-1}{n} \sum_{k\geq 0} P_{n-1,k} z^k \\ &= \frac{z}{n} \sum_{k\geq 0} P_{n-1,k} z^k + \frac{n-1}{n} \sum_{k\geq 0} P_{n-1,k} z^k \\ &= \frac{z}{n} G_{n-1}(z) + \frac{n-1}{n} G_{n-1}(z) \\ &= \frac{z+n-1}{n} G_{n-1}(z). \end{aligned}$$

Iteriert man nun diese Rekursion und beachtet, dass $G_1(z) = 1$ ist, erhält man

$$G_n(z) = \prod_{j=2}^{n} \frac{z+j-1}{j}. \tag{3.3}$$

Es stellt sich allerdings heraus, dass es leichter ist, die Rekursion abzuleiten, um so auf den Erwartungswert zu kommen:

$$\begin{aligned} G'_n(z) &= \frac{1}{n} G_{n-1}(z) + \frac{z+n-1}{n} G'_{n-1}(z) \\ G'_n(1) &= \frac{1}{n} \underbrace{G_{n-1}(1)}_{=1} + G'_{n-1}(1) \\ G'_n(1) &= \frac{1}{n} + G'_{n-1}(1). \end{aligned}$$

Für den Erwartungswert gilt nun $\mathbb{E}(b) = G'_n(1) = \sum_{j=2}^{n} \frac{1}{j} = H_n - 1 = O(\log(n))$. □

Satz 3.3.4 *Die Varianz von b ist gegeben durch:*

$$\mathbb{V}(b) = H_n - H_n^{(2)} = O(\log(n)).$$

Beweis. Es ist also $\sum_{k\geq 0} k^2 P_{n,k} - (\sum_{k\geq 0} kP_{n,k})^2$ zu untersuchen. Hat man eine wahrscheinlichkeitserzeugende Funktion $F(z) = \sum_{k\geq 0} p_k z^k$ gegeben, so gilt (siehe auch Satz 1.6.3 auf Seite 11):

$$\sum_{k\geq 0} k^2 p_{n,k} - \left(\sum_{k\geq 0} kp_{n,k}\right)^2 = F''(1) + F'(1) - (F'(1))^2.$$

Lemma 3.3.5 *Seien $F(z)$ und $G(z)$ zwei wahrscheinlichkeitserzeugende Funktionen, also $F(1) = G(1) = 1$. Dann gilt für die zugehörigen Zufallsvariablen F und G:*

$$\mathbb{V}(FG) = \mathbb{V}(F) \cdot \mathbb{V}(G).$$

Beweis.

$$\mathbb{V}(FG) = \overbrace{F''(1)G(1) + 2F'(1)G'(1) + F(1)G''(1)}^{(FG)''} + \\ + \underbrace{F'(1)G(1) + F(1)G'(1)}_{(FG)'} - \underbrace{(F'(1)G(1) + F(1)G'(1))^2}_{((FG)')^2}.$$

Ordnet man jetzt die Summanden anders an und berücksichtigt $F(1) = G(1) = 1$, erhält man

$$\mathbb{V}(FG) = \underbrace{\left(F''(1) + F'(1) - (F'(1))^2\right)}_{\mathbb{V}(F)} \cdot \underbrace{\left(G''(1) + G'(1) - (G'(1))^2\right)}_{\mathbb{V}(G)}. \qquad \square$$

Nachdem nun für $g_j(z) := \frac{z+j-1}{j}$ gilt: $g(z) = 1$, ist $G_n(z) = \prod_{j=2}^{n} g_j(z)$ das Produkt von wahrscheinlichkeitserzeugenden Funktionen (siehe Darstellung (3.3)). Mit Lemma 3.3.5 erhält man:

$$\mathbb{V}(G_n) = \sum_{j=2}^{n} \mathbb{V}(g_j) = \sum_{j=2(1)}^{n} \left(\frac{1}{j} - \frac{1}{j^2}\right) = H_n - H_n^{(2)}. \qquad \square$$

Nun gilt es, sich mit dem Parameter a zu beschäftigen.

Satz 3.3.6 *Der Erwartungswert von a ist gegeben durch:*

$$\mathbb{E}(a) = (n+1)H_n - 2n.$$

Beweis. Aus der Darstellung (3.1) auf Seite 18 folgt für

$$y_{ij} = \begin{cases} 1, & \text{wenn } q(i) < q(k) \text{ für } i < k \leq j, \\ 0, & \text{in allen übrigen Fällen,} \end{cases}$$

dass a gegeben ist durch:

$$a = \sum_{1 \le i < j \le n} y_{ij}.$$

Im Folgenden wird aufgrund der besseren Übersicht die abkürzende Schreibweise: $\overline{a} := \mathbb{E}(a)$ und $\overline{y_{ij}} := \mathbb{E}(y_{ij})$ verwendet. $\overline{y_{ij}}$ ist die Anzahl an Permutationen mit $y_{ij} = 1$ gebrochen durch die Anzahl aller möglichen Permutationen, also durch $n!$. Die Wahrscheinlichkeit, dass $q(i) = \min\{q(k) | i \le k \le j\}$ ist, beträgt $\frac{1}{j-i+1}$. Somit erhält man

$$\overline{a} = \sum_{1 \le i < j \le n} \overline{y_{ij}} = \sum_{1 \le i < j \le n} \frac{1}{j-i+1} = \sum_{2 \le r \le n} \frac{n+1-r}{r}. \tag{3.4}$$

Im letzten Schritt wurde $j - i + 1$ durch die neue Variable r ersetzt. Es folgt

$$\overline{a} = (n+1) \sum_{2 \le r \le n} \frac{1}{r} - \sum_{2 \le r \le n} 1 = (n+1)(H_n - 1) - (n-1) = (n+1)H_n - 2n. \qquad \square$$

Satz 3.3.7 *Die Varianz von a ist gegeben durch:*

$$\mathbb{V}(a) = 2n^2 - (n+1)^2 H_n^{(2)} - (n+1)H_n + 4n.$$

Beweis. Die von D. E. Knuth angegebene Lösung (siehe [2]) erweist sich als sehr aufwändig und rechenintensiv. Es werden hier die wichtigsten Schritte angeführt, ohne jeden einzelnen Zwischenschritt auszuführen. Zuerst berechnet man den Erwartungswert von a^2:

$$\mathbb{E}(a^2) = \mathbb{E}\left(\left(\sum_{1 \le i < j \le n} y_{ij}\right)^2\right) = \mathbb{E}\left(\sum_{1 \le i < j \le n} y_{ij}^2 + \sum_{\substack{1 \le i < j \le n \\ 1 \le k < l \le n \\ (i,j) \ne (k,l)}} y_{ij} y_{kl}\right).$$

Da y_{ij} nur die Werte 0 und 1 annimmt, ist $y_{ij}^2 = y_{ij}$. Wenn man das berücksichtigt und Symmetrieeigenschaften ausnützt, kann man so fortfahren:

$$\begin{aligned}&\mathbb{E}\left(\sum_{1 \le i < j \le n} y_{ij} + 2 \sum_{1 \le i < j < k < l \le n} (y_{ij}y_{kl} + y_{ik}y_{jl} + y_{il}y_{jk}) + 2 \sum_{1 \le i < j < k \le n} (y_{ij}y_{jk} + y_{ik}y_{jk} + y_{ij}y_{ik})\right) \\ &= \overline{a} + 2(A + B + C + D + E + F).\end{aligned}$$

Auf die letzte Zeile kommt man, wenn man die Linearität des Erwartungswertes ausnützt und die Variablen A bis F entsprechend definiert, also $A := \sum \mathbb{E}(y_{ij}y_{kl}), B := \sum \mathbb{E}(y_{ik}y_{jl}), ..., F := \sum \mathbb{E}(y_{ij}y_{ik})$. Der Ausdruck $y_{ij}y_{kl}$ nimmt den Wert 1 genau dann an, wenn sowohl y_{ij} also

auch y_{kl} den Wert 1 annehmen. Der Erwartungswert von $y_{ij}y_{kl}$ bei fixem $i < j < k < l$ ist also $\frac{1}{(j-i+1)(l-k+1)}$. Analoges Vorgehen ergibt $\mathbb{E}(y_{ik}y_{jl}) = \frac{1}{(l-i+1)(l-j+1)}$ und $\mathbb{E}(y_{il}y_{jk}) = \frac{1}{(l-i+1)(k-j+1)}$. Aufgrund der Monotonie $i < j < k < l$ und der Definition von y_{mn} für $m < n$ mit $m, n \in \{i, j, k, l\}$ ergibt sich $y_{ij}y_{jk} = y_{ik}y_{jk}$ und $y_{ij}y_{ik} = y_{ik}$ und somit $\mathbb{E}(y_{ij}y_{jk}) = \mathbb{E}(y_{ik}y_{jk}) = \frac{1}{(k-i+1)(k-j+1)}$ und $\mathbb{E}(y_{ij}y_{ik}) = \frac{1}{(k-i+1)}$.

Definiert man nun

$$
\begin{aligned}
X &:= \sum_{1 \le i < j \le n} \frac{1}{j-i+1}, \\
Y &:= \sum_{1 \le i < j \le n} H_{j-i}, \\
Z &:= \sum_{1 \le i < j \le n} \frac{1}{j-i+1} H_{j-i},
\end{aligned}
$$

so erhält man

$$
\begin{aligned}
B &= \binom{n}{2} - 2Z, \\
C &= Y - Z - 2\binom{n}{2} + 3X, \\
D &= E = Z - X, \\
F &= \binom{n}{2} - 2X.
\end{aligned}
$$

X wurde bereits berechnet, indem man $j - i + 1$ durch eine neue Variable ersetzt hat. Analog geht man bei Y und Z vor. Unter Verwendung bekannter Formeln für harmonische Zahlen erhält man:

$$
\begin{aligned}
X &= (n+1)H_n - 2n, \\
Y &= \frac{1}{2}(n^2+n)H_n - \frac{3}{4}n^2 - \frac{1}{4}n, \\
Z &= \frac{1}{2}(n+1)\left(H_n^2 - H_n^{(2)}\right) - nH_n + n.
\end{aligned}
$$

Nachdem man nun die Größen B bis F im Griff hat, muss noch A untersucht werden. Es ergibt sich zunächst:

$$
A = \sum_{1 \le i < j < k < l \le n} \frac{1}{(j-i+1)(l-k+1)} = \sum_{\substack{r \ge 2,\ s \ge 2 \\ r+s \le n}} \frac{1}{rs} \binom{n-r-s+2}{2}.
$$

Es wurde also wieder $j-i+1$ durch die neue Variable r und $l-k+1$ durch s ersetzt. Im nächsten Schritt soll nun $r+s =: t$ gesetzt werden und man erhält unter Ausnützung von Symmetrien:

$$\begin{aligned} A &= \sum_{\substack{2\le r\le t-2\\ 4\le t\le n}} \frac{1}{r(t-r)}\binom{n-t+2}{2} = \sum_{\substack{2\le r\le t-2\\ 4\le t\le n}} \frac{1}{t}\left(\frac{1}{r}+\frac{1}{t-r}\right)\binom{n-t+2}{2} \\ &= 2\sum_{\substack{2\le r\le t-2\\ 4\le t\le n}} \frac{1}{rt}\binom{n-t+2}{2} = \sum_{\substack{2\le r\le t-2\\ 4\le t\le n}} \frac{1}{rt}\left((n+2)(n+1)-t(2n+3)+t^2\right). \end{aligned}$$

Führt man nun wieder drei neue Variablen ein, kann man die Größe A endlich beschreiben. Aus

$$\begin{aligned} U &:= \frac{1}{2}(H_n-1)^2 - \frac{1}{2}H_n^{(2)} + \frac{1}{n}, \\ V &:= (n-1)H_{n-2} - 2n + 4, \\ W &:= \frac{1}{2}\left(\left(n^2+n-2\right)(H_{n-2}-1) - \frac{1}{2}(n-1)(n-2) + 1 - 3(n-3)\right) \end{aligned}$$

folgt nämlich:

$$A = (n+2)(n+1)U - (2n+3)V + W.$$

Fasst man nun alles zusammen und subtrahiert man noch $\overline{a}^2$ von dem errechneten $\mathbb{E}(a^2)$, erhält man die gesuchte Varianz von a:

$$\mathbb{V}(a) = 2n^2 - (n+1)^2 H_n^{(2)} - (n+1)H_n + 4n. \qquad \square$$

3.4 Berechnung höherer Momente

Die Berechnung der Varianz des Parameters a hat sich im vorigen Abschnitt als äußerst mühsam herausgestellt, da bereits die Untersuchung des zweiten Moments größere Probleme bereitet. Es ist also nicht zielführend, sich mit dieser Methode an höhere Momente heranzutasten. In diesem Abschnitt soll ein Verfahren vorgestellt werden, mit dem man beliebig hohe faktorielle Momente ausrechnen kann und im Speziellen wird die Varianz von a auf eine einfachere Weise berechnet, siehe [3].

Ruft man sich die Darstellung (3.1) des Parameters a in Erinnerung, kann man jetzt $a_{n,k}$ wie folgt definieren.

Definition 3.4.1 *Mit $a_{n,k}$ bezeichnet man die Anzahl an Permutationen der Länge n, für die $a(\pi) = k$ gilt, also*

$$a_{n,k} := |\{\pi : a(\pi) = k\}|, \qquad \textit{für} \quad \pi : \{1,...,n\} \to \{1,...,n\} \textit{ bijektiv.}$$

Wir führen eine entsprechende wahrscheinlichkeitserzeugende Funktion ein:

$$G_n(z) := \sum_{k\geq 0} a_{n,k} \frac{z^k}{n!}. \tag{3.5}$$

Es gilt $G_n(1) = 1$, da $\sum_{k\geq 0} a_{n,k}$ gleich der Anzahl aller möglichen Permutationen ist, also $n!$, weil man ja über alle möglichen k summiert. Für $G_n(z)$ findet man folgende Rekursionsgleichung.

Satz 3.4.2 *Sei die wahrscheinlichkeitserzeugende Funktion $G_n(z)$ wie in Darstellung (3.5) definiert. Dann gilt:*

$$G_n(z) = \frac{1}{n} \sum_{k=0}^{n-1} z^k \, G_k(z) \, G_{n-1-k}(z), \quad \textit{für } n \geq 1,$$
$$G_0(z) = 1.$$

Beweis. Aufgrund von $a_{0,k} = 1$ für $k = 1$ und $a_{0,k} = 0$ für $k \neq 1$ gilt $G_0(z) = 1$.
Sei die Permutation π in der kanonischen Zyklendarstellung gegeben. Nun soll rekursiv nach dem kleinsten Element aufgespalten werden. Zu Beginn unterteilt man also π nach dem Einselement:

$$\pi = \rho 1 \sigma.$$

Bezeichnet man mit $|\rho|$ und $|\sigma|$ die Längen der Folgen ρ und σ ergibt sich $|\rho| = n - 1 - |\sigma|$ und aus der Interpretation des Parameters a (Darstellung (3.1), Seite 18) folgende Gleichung, da die 1 als kleinstes Element immer kleiner als alle Nachfolger sein wird und somit mit jedem Nachfolger einen Beitrag zur Größe a liefert:

$$a(\pi) = a(\rho) + a(\sigma) + |\sigma|.$$

Zur Veranschaulichung sei hier ein kleines Beispiel gegeben.

Beispiel 3.4.3 *Sei die Permutation $\pi = (2\ 4\ 5\ 1\ 3)$ in kanonischer Zyklenschreibweise gegeben. Der Parameter a hat den Wert vier, da die folgenden Paare je einen Beitrag leisten:*

$(2, 4)$, *da* $2 < 4$,
$(2, 5)$, *da* $2 < 4 < 5$,
$(4, 5)$, *da* $4 < 5$,
$(1, 3)$, *da* $1 < 3$.

Mit Hilfe der Rekursion kommt man so zum Resultat (dabei bezeichnet ϵ das Nullwort):

$$\begin{aligned}
a(2\ 4\ 5\ \underline{1}\ 3) &= a(\underline{2}\ 4\ 5) + a(\underline{3}) + 1 \\
&= \overbrace{\underbrace{a(\epsilon)}_{=0} + a(4\ 5) + 2} + \overbrace{\underbrace{a(\epsilon)}_{=0} + \underbrace{a(\epsilon)}_{=0} + 0} + 1 \\
&= a(\underline{4}\ 5) + 3 \\
&= \overbrace{\underbrace{a(\epsilon)}_{=0} + \underbrace{a(5)}_{=0} + 1} + 3 = 4.
\end{aligned}$$

Wenn σ die Länge m hat, dann gibt es $\binom{n-1}{m}$ Möglichkeiten, die m Elemente aus den verbleibenden $n-1$ (denn das kleinste Element ist ausgenommen) auszuwählen. Summiert man über alle möglichen m auf, ergibt sich

$$a_{n,k} = \sum_{m=0}^{n-1} \binom{n-1}{m} \sum_{i+j+m=k} a_{n-1-m,i}\, a_{m,j}.$$

Die zweite Summation ergibt sich aus den unterschiedlichen Möglichkeiten für einen Beitrag zum Parameter a. ρ liefert den Beitrag i, während σ den Beitrag j liefert. Insgesamt muss es sich immer auf k ausgehen. Die Multiplikation mit $\frac{z^k}{n!}$ und anschließende Summation über $k \geq 0$ führt zur aufgestellten Behauptung des Satzes:

$$\begin{aligned}
G_n(z) &= \sum_{m=0}^{n-1} \frac{(n-1)!}{n!\,m!\,(n-1-m)!} \sum_{k\geq 0} \sum_{i+j+m=k} a_{n-1-m,i}\, a_{m,j}\, z^k \\
&= \frac{1}{n} \sum_{m=0}^{n-1} \sum_{k\geq 0} \sum_{i+j+m=k} \frac{a_{n-1-m,i}}{(n-1-m)!} \frac{a_{m,j}}{m!}\, z^{(i+j+m)} \\
&= \frac{1}{n} \sum_{m=0}^{n-1} z^m \underbrace{\left(\sum_{i\geq 0} \frac{a_{n-1-m,i}}{(n-1-m)!}\, z^i \right)}_{=G_{n-1-m}} \cdot \underbrace{\left(\sum_{j\geq 0} \frac{a_{m,j}}{m!}\, z^j \right)}_{=G_m}.
\end{aligned}$$

□

Mit diesem Satz kann man jetzt Erwartungswert und Varianz direkt ausrechnen. Dabei geht man wie folgt vor.

$$\begin{aligned}
\overline{a}_n &= \frac{1}{n!} \sum_{k\geq 0} k\, a_{n,k} = G'_n(1) \\
&= \frac{1}{n} \sum_{k=0}^{n-1} \Big(k\, z^{k-1}\, G_{n-1-k}(z)\, G_k(z) + z^k\, G'_{n-1-k}(z)\, G_k(z) + z^k\, G_{n-1-k}(z)\, G'_k(z) \Big) \Big|_{z=1} \\
&= \frac{1}{n} \sum_{k=0}^{n-1} \big(k + G'_{n-1-k}(1) + G'_k(1) \big) \\
&= \frac{1}{n} \sum_{k=0}^{n-1} (k + \overline{a}_{n-1-k} + \overline{a}_k).
\end{aligned}$$

Aus Symmetriegründen ergibt sich weiters:

$$\overline{a}_n = \frac{1}{n} \left(\binom{n}{2} + 2 \sum_{k=0}^{n-1} \overline{a}_k \right) = \frac{2}{n} \sum_{k=0}^{n-1} \overline{a}_k + \frac{n-1}{2}.$$

Betrachtet man jetzt zwei direkt aufeinanderfolgende n-Werte und subtrahiert die beiden zugehörigen Gleichungen, erhält man eine lineare Rekursion erster Ordnung:

$$I: \qquad n\,\overline{a}_n = 2\sum_{k=0}^{n-1}\overline{a}_k + \frac{n(n-1)}{2}, \qquad n \geq 1$$

$$II: \qquad (n-1)\,\overline{a}_{n-1} = 2\sum_{k=0}^{n-2}\overline{a}_k + \frac{(n-1)(n-2)}{2}, \qquad n \geq 2$$

$$I - II: \qquad n\,\overline{a}_n - (n-1)\,\overline{a}_{n-1} = 2\,\overline{a}_{n-1} + \frac{n-1}{2}\,2, \qquad n \geq 2$$

Man erhält also:

$$n\,\overline{a}_n = (n+1)\,\overline{a}_{n-1} + n - 1, \qquad \text{für} \quad n \geq 2.$$

Betrachtet man den Fall $n = 1$ erkennt man, dass die Gleichheit in der Rekursionsgleichung erhalten bleibt und somit obige Rekursion auch für $n \geq 1$ gilt:

$$\overline{a}_1 = 2\,\overline{a}_0 + 0 = 0.$$

$\overline{a}_0$ und $\overline{a}_1$ müssen nämlich den Wert null haben, da $a_{0,k}$ für alle k null ist bzw. $a_{1,k}$ nur bei $k = 0$ den Wert eins annimmt.

Im nächsten Schritt wird ein geeigneter Summationsfaktor gesucht, um die Rekursion lösen zu können. Hat man eine Gleichung der Form $\alpha_n a_n = \beta_n a_{n-1} + \gamma_n$ so ist der passende Summationsfaktor $\frac{\prod_{k=1}^{n-1}\alpha_k}{\prod_{k=1}^{n}\beta_k}$. Wenn man nämlich die Gleichung damit multipliziert erhält man folgenden Ausdruck:

$$\underbrace{\prod_{k=0}^{n}\frac{\alpha_k}{\beta_k}a_n}_{=:b_n} = \underbrace{\prod_{k=0}^{n-1}\frac{\alpha_k}{\beta_k}a_{n-1}}_{=:b_{n-1}} + \frac{\prod_{k=1}^{n-1}\alpha_k}{\prod_{k=1}^{n}\beta_k}\,\gamma_n.$$

Jetzt ist es leicht, die Lösung durch Summation für die neu eingeführte Variable b_n zu erhalten. Bei unserer Rechnung sieht das nun so aus:

Mit dem Summationsfaktor $\dfrac{(n-1)!}{(n+1)!} = \dfrac{1}{n(n+1)}$

wird die Gleichung mutlipliziert und man erhält:

$$\underbrace{\frac{\overline{a}_n}{n+1}}_{=:b_n} = \underbrace{\frac{\overline{a}_{n-1}}{n}}_{=:b_{n-1}} + \frac{n-1}{n(n+1)}, \qquad n \geq 1.$$

Nach der Variablentransformation ergibt sich:

$$b_n = \sum_{j=1}^{n}\frac{j-1}{j(j+1)} + b_0, \qquad b_0 = \frac{\overline{a}_0}{1} = 0.$$

Rücktransformation ergibt nun den gesuchten Erwartungswert, der mit Hilfe einer Partialbruchzerlegung auch explizit bestimmt werden kann:

$$\begin{aligned}\overline{a_n} = \mathbb{E}(a) &= (n+1)\sum_{j=1}^{n}\frac{j-1}{j(j+1)} = (n+1)\sum_{j=1}^{n}\left(-\frac{1}{j}+\frac{2}{j+1}\right)\\ &= (n+1)\left(-H_n + 2(H_{n+1}-1)\right) = (n+1)\left(-H_n + 2\left(H_n + \frac{1}{n+1}\right) - 2\right)\\ &= (n+1)H_n - 2n.\end{aligned}$$

Auch die Varianz lässt sich mit diesem Ansatz bestimmen (siehe Satz 1.6.3 auf Seite 11). Dazu berechnet man zuerst für das zweite Moment den Ausdruck $G_n''(1)$ auf genau die gleiche Weise wie $G_n'(1)$. Man muss im Laufe der Rechnung den bereits bekannten Erwartungswert einsetzen und kommt so wieder zu einer linearen Rekursion, die sich von obiger nur im Term γ_n unterscheidet. Es bleiben also Summationsfaktor und Lösungsmethode die gleiche. Um für γ_n einen geschlossenen Ausdruck zu erhalten, bedarf es folgenden Lemmata und den im Unterkapitel 1.4 vorgestellten Summationsformeln.

Lemma 3.4.4 *Für $n \in \mathbb{N}, n \geq 1$ gilt:*

$$\sum_{k=1}^{n} H_k = (n+1)H_n - n.$$

Beweis.

$$\sum_{k=1}^{n} H_k = \sum_{k=1}^{n}\sum_{j=1}^{k}\frac{1}{j} = \sum_{j=1}^{n}\sum_{k=j}^{n}\frac{1}{j} = \sum_{j=1}^{n}\frac{1}{j}\sum_{k=j}^{n}1 = \sum_{j=1}^{n}\frac{1}{j}(n-j+1) = (n+1)H_n - n. \qquad \square$$

Lemma 3.4.5 *Für $n \in \mathbb{N}, n \geq 1$ gilt:*

$$\sum_{k=1}^{n} k\,H_k = \binom{n+1}{2}H_n - \frac{1}{2}\binom{n}{2}.$$

Beweis. Wie im Beweis des Lemmas 3.4.4 werden die Summen vertauscht und mit Hilfe der Summenformeln folgt die Behauptung.

$$\begin{aligned}\sum_{k=1}^{n} k\,H_k &= \sum_{k=1}^{n}\sum_{j=1}^{k} k\,\frac{1}{j} = \sum_{j=1}^{n}\sum_{k=j}^{n} k\,\frac{1}{j} = \sum_{j=1}^{n}\frac{1}{j}\sum_{k=j}^{n} k = \sum_{j=1}^{n}\frac{1}{j}\left(\frac{n(n+1)}{2} - \frac{(j-1)j}{2}\right)\\ &= \binom{n+1}{2}H_n - \sum_{j=1}^{n}\frac{j-1}{2} = \binom{n+1}{2}H_n - \frac{1}{2}\binom{n}{2}. \qquad \square\end{aligned}$$

Lemma 3.4.6 *Für $n \in \mathbb{N}, n \geq 1$ gilt:*

$$\sum_{k=1}^{n} k^2\,H_k = \frac{n(n+1)(2n+1)}{6}H_n + \frac{n(n+1)(2n+1)}{18} - \frac{n(n+1)}{4} + \frac{n}{6}.$$

Beweis. Wie im Lemma 3.4.5 führt auch hier die Summenvertauschung und die Ausnützung der Summenformeln zum Ziel.

$$\begin{aligned}\sum_{k=1}^{n} k^2\, H_k &= \sum_{j=1}^{n} \frac{1}{j} \sum_{k=j}^{n} k^2 = \sum_{j=1}^{n} \frac{1}{j} \left(\frac{n(n+1)(2n+1)}{6} - \frac{(j-1)j(2j-1)}{6} \right) \\ &= \frac{n(n+1)(2n+1)}{6} H_n + \sum_{j=1}^{n} \frac{(j-1)(2j-1)}{6} \\ &= \frac{n(n+1)(2n+1)}{6} H_n + \frac{1}{3}\sum_{j=1}^{n} j^2 - \frac{1}{2}\sum_{j=1}^{n} j + \frac{1}{6}\sum_{j=1}^{n} 1 \\ &= \frac{n(n+1)(2n+1)}{6} H_n + \frac{n(n+1)(2n+1)}{18} - \frac{n(n+1)}{4} + \frac{n}{6}. \end{aligned}$$

□

Für die Berechnung höherer Momente eignet sich aber auch dieser direkte Ansatz nicht. Das Ziel ist nun, die faktoriellen Momente mit Hilfe einer Integralgleichung bestimmen zu können (siehe [3]). Dazu wird die Doppelt-Erzeugende Funktion $H(z,u)$ eingeführt.

Definition 3.4.7 *Sei $H(z,u)$ die erzeugende Funktion der wahrscheinlichkeitserzeugenden Funktion $G_n(z)$, also*

$$H(z,u) := \sum_{n\geq 0} G_n(z) u^n.$$

Korollar 3.4.8 *Es gilt für die wahrscheinlichkeitserzeugende Funktion $H(z,u)$:*

$$\begin{aligned}\frac{\partial}{\partial u} H(z,u) &= H(z,u)\, H(z,zu), \\ H(1,u) &= \frac{1}{1-u}. \end{aligned}$$

Beweis. Multipliziert man die Rekursionsgleichung aus Satz 3.4.2 auf Seite 24 mit nu^{n-1} und summiert über alle $n \geq 0$ auf, folgt die Aussage unmittelbar:

$$\sum_{n\geq 0} G_n(z) n u^{n-1} = \sum_{n\geq 0} \sum_{k=0}^{n-1} z^k\, u^k\, G_k(z)\, u^{n-1-k}\, G_{n-1-k}(z).$$

Da $G_n(1)$ eine wahrscheinlichkeitserzeugende Funktion ist und somit $G_n(1) = 1$ gilt, ergibt sich für $H(1,u) = \sum_{n\geq 0} u^n$ die geometrische Reihe, von der man die Summenformel kennt. □

Definition 3.4.9 *Sei $\beta_s(n)$ das s-te faktorielle Moment, das durch die wahrscheinlichkeitserzeugende Funktion $G_n(z)$ gegeben ist:*

$$\beta_s(n) = \frac{d^s}{dz^s} G(z) \bigg|_{z=1}. \tag{3.6}$$

Mit $f_s(u)$ sei die erzeugende Funktion für die faktoriellen Momente bezeichnet:

$$f_s(u) = \sum_{n\geq 0} \beta_s(n) u^n. \tag{3.7}$$

Aus der Taylorentwicklung um den Punkt $z = 1$ ergibt sich mit (3.6) und (3.7):

$$H(z,u) = \sum_{s\geq 0} f_s(u)\frac{(z-1)^s}{s!}. \tag{3.8}$$

Satz 3.4.10 *Sei $f_s(u)$ wie in Definition 3.4.9, Gleichung* (3.7) *definiert. Dann gilt:*

$$f_s'(u) - \frac{2}{1-u}f_s(u) = h_s(u), \quad \textit{für } s \geq 1, \textit{ mit}$$
$$h_s(u) = \sum_{i=1}^{s-1}\binom{s}{i}f_i(u)\sum_{r=0}^{s-1}\binom{s-i}{r}u^r f_{s-r-i}^{(r)}(u) + \frac{1}{1-u}\sum_{r=1}^{s}\binom{s}{r}u^r f_{s-r}^{(r)}(u);$$

$$f_0(u) = \frac{1}{1-u}, \quad h_0(u) = -\frac{1}{(1-u)^2},$$
$$f_s(0) = 0 \textit{ für } s \geq 1.$$

Beweis. Wenn wir mit Hilfe der Taylorreihe die Funktion $f_j(zu)$ um u entwickeln, ergibt sich:

$$f_j(zu) = \sum_{k\geq 0} f_j^k(u)\frac{(z-1)^k u^k}{k!}.$$

Setzt man die Entwicklung aus (3.8) in die Gleichung aus Korollar 3.4.8 ein, erhält man:

$$\begin{aligned}\sum_{s\geq 0} f_s(u)'\frac{(z-1)^s}{s!} &= \left(\sum_{i\geq 0} f_i(u)\frac{(z-1)^i}{i!}\right)\left(\sum_{j\geq 0} f_j(zu)\frac{(z-1)^j}{j!}\right)\\ &= \left(\sum_{i\geq 0} f_i(u)\frac{(z-1)^i}{i!}\right)\left(\sum_{j\geq 0}\frac{(z-1)^j}{j!}\left(\sum_{k\geq 0} f_j^{(k)}(u)\frac{(z-1)^k u^k}{k!}\right)\right)\\ &= \sum_{m\geq 0}\sum_{i+j+k=m} u^k f_i(u) f_j^{(k)}(u)\frac{(z-1)^m}{i!\,j!\,k!}.\end{aligned}$$

Vergleicht man jetzt die Koeffizienten von $\frac{(z-1)^s}{s!}$ $\Big($mit $f_0(u) = \frac{1}{1-u}$ (siehe unten)$\Big)$ führt das zu:

$$\begin{aligned}f_s(u)' &= \sum_{i+j+k=s} s!\,u^k f_i(u) f_j^{(k)}(u)\frac{1}{i!\,j!\,k!}\\ &= \sum_{i=0}^{s}\binom{s}{i}f_i(u)\sum_{k=0}^{s-i}\binom{s-i}{k}u^k f_{s-i-k}^{(k)}(u)\\ &= \underbrace{\frac{1}{1-u}f_s(u)}_{i=s} + \sum_{i=1}^{s-1}\binom{s}{i}f_i(u)\sum_{k=0}^{s-i}\binom{s-i}{k}u^k f_{s-i-k}^{(k)}(u) + \underbrace{\frac{1}{1-u}\sum_{k=0}^{s}\binom{s}{k}u^k f_{s-k}^{(k)}(u)}_{i=0}\\ &= \frac{2}{1-u}f_s(u) + \sum_{i=1}^{s-1}\binom{s}{i}f_i(u)\sum_{k=0}^{s-i}\binom{s-i}{k}u^k f_{s-i-k}^{(k)}(u) + \frac{1}{1-u}\sum_{k=1}^{s}\binom{s}{k}u^k f_{s-k}^{(k)}(u).\end{aligned}$$

Die letzte Zeile folgt aus der Tatsache, dass $\beta_0(n) = 1$ für alle n und daher unter Ausnützung der Summenformel für die geometrische Reihe $f_0(u) = \frac{1}{1-u}$.
Aus $G_0(z) = 1$ folgt $\beta_s(0) = 0$ für $s \geq 1$ und somit $f_s(0) = 0$ für $s \geq 0$.
Abschließend ergibt sich

$$h_0(u) = f_0'(u) - \frac{2}{1-u} f_0(u) = \left(\frac{1}{1-u}\right)' - \frac{2}{(1-u)^2} = \frac{1}{(1-u)^2} - \frac{2}{(1-u)^2} = -\frac{1}{(1-u)^2}. \quad \square$$

Löst man jetzt die lineare Differentialgleichung erster Ordnung aus diesem Satz, erhält man die gesuchte Integralformel zur Berechnung beliebig hoher faktorieller Momente.

Korollar 3.4.11 *$f_s(u)$ und $h_s(u)$ seien wie in Definition 3.4.9, Gleichung* (3.7)*, bzw. wie in Satz 3.4.10 definiert. Für $s \geq 1$ gilt dann*

$$f_s(u) = \frac{1}{(1-u)^2} \int_0^u h_s(t)(1-t)^2 dt. \tag{3.9}$$

Mit diesem Werkzeug sollen jetzt Erwartungswert und Varianz bestimmt werden. Der nachfolgende Satz liefert die dazu notwendigen ersten beiden Momente.

Satz 3.4.12 *Sei $L(u) := -\log(1-u)$. Dann gilt:*

$$f_1(u) = L(u)\frac{1}{(1-u)^2} - \frac{1}{(1-u)^2} + \frac{1}{(1-u)},$$
$$f_2(u) = \frac{2L^2(u)}{(1-u)^3} - \frac{2L(u)}{(1-u)^3} + \frac{2}{(1-u)^3} - \frac{L^2(u)}{(1-u)^2} - \frac{2}{(1-u)^2},$$

$$\beta_1(n) = (n+1)H_n - 2n,$$
$$\beta_2(n) = (n+1)^2(H_n^2 - H_n^{(2)}) - (4n+2)(n+1)H_n + 6n(n+1).$$

Beweis. Duch Einsetzen erhält man $h_1(u) = \frac{1}{1-u} u f_0'(u) = \frac{u}{(1-u)^3}$. Damit lässt sich jetzt $f_1(u)$ ausrechnen:

$$\begin{aligned} f_1(u) &= \frac{1}{(1-u)^2} \int_0^u h_1(t)(1-t)^2 dt = \frac{1}{(1-u)^2} \int_0^u \frac{t}{(1-t)^3}(1-t)^2 dt \\ &= \frac{1}{(1-u)^2} \int_0^u \frac{t}{1-t} dt = \frac{1}{(1-u)^2} \int_0^u \frac{1}{1-t} - \frac{1-t}{1-t} dt \\ &= \frac{1}{(1-u)^2}(-\log(1-t) - t)|_0^u = \frac{1}{(1-u)^2}(-\log(1-u) - 1 + 1 - u) \\ &= L(u)\frac{1}{(1-u)^2} - \frac{1}{(1-u)^2} + \frac{1}{(1-u)}. \end{aligned}$$

$\square$

Analog geht man für f_2 vor. Zuerst soll also h_2 mit der Formel aus Satz 3.4.10 berechnet werden. Der äußere Laufindex i nimmt nur den Wert 1 an und man erhält

$$\begin{aligned}
h_2(u) &= \binom{2}{1} f_1(u) \sum_{r=0}^{1} \binom{1}{r} u^r f_{1-r}^{(r)}(u) + \frac{1}{1-u} \sum_{r=1}^{2} \binom{2}{r} u^r f_{2-r}^{(r)}(u) \\
&= 2\left(L(u)\frac{1}{(1-u)^2} - \frac{u}{(1-u)^2}\right)\left(f_1(u) + uf_0'(u)\right) + \frac{1}{1-u}\left(2uf_1'(u) + u^2 f_0''(u)\right) \\
&= 2\left(L(u)\frac{1}{(1-u)^2} - \frac{u}{(1-u)^2}\right)\left(L(u)\frac{1}{(1-u)^2} - \frac{u}{(1-u)^2} + \frac{u}{(1-u)^2}\right) + \\
&+ \frac{1}{1-u}\left(2u\left(\frac{1}{(1-u)^3} + L(u)\frac{2}{(1-u)^3} - \frac{1-u+2u}{(1-u)^3}\right) + u^2\frac{2}{(1-u)^3}\right) \\
&= L^2(u)\frac{2}{(1-u)^4} - L(u)\frac{2u}{(1-u)^4} + \frac{2u}{(1-u)^4} + L(u)\frac{4u}{(1-u)^4} - 2u\frac{1+u}{(1-u)^4} + \frac{2u^2}{(1-u)^4} \\
&= L^2(u)\frac{2}{(1-u)^4} + L(u)\frac{2u}{(1-u)^4}.
\end{aligned}$$

Das ausgerechnete h_2 wird jetzt in die Integrationsgleichung eingesetzt, um f_2 zu berechnen. Der auftretende Integrand lässt sich unter Ausnützung der Linearität des Integrals mit geeigneter Substitution und anschließender partieller Integration behandeln.

$$\begin{aligned}
f_2(u) &= \frac{1}{(1-u)^2}\, 2\int_0^u \left(L^2(t)\frac{1}{(1-t)^4} + L(t)\frac{t}{(1-t)^4}\right)(1-t)^2 dt \\
&= \frac{1}{(1-u)^2}\underbrace{2\int_0^u L^2(t)\frac{1}{(1-t)^2}dt}_{I} + \underbrace{2\int_0^u L(t)\frac{t}{(1-t)^2}dt}_{II}.
\end{aligned}$$

Jetzt wird für die beiden Integrale die Substitution $-\log(1-t) = r$ gewählt. Daraus ergibt sich $\frac{1}{1-t}dt = dr$ und $\frac{1}{1-t} = e^r$ bzw. $t = 1 - \frac{1}{e^r}$. Weiters müssen die Grenzen angepasst werden: Wenn sich t von 0 bis u erstreckt, dann muss sich r im Bereich von 0 bis $-\log(1-u)$ liegen.

$$\begin{aligned}
I: \quad & 2\int_0^u (-\log(1-t))^2 \frac{1}{(1-t)^2}dt = 2\int_0^{-\log(1-u)} r^2 e^r dr \\
&= 2L^2(u)\frac{1}{1-u} - 2\int_0^{-\log(1-u)} 2re^r dr = 2L^2(u)\frac{1}{1-u} - 4L(u)\frac{1}{1-u} + 4\int_0^{-\log(1-u)} e^r dr \\
&= 2L^2(u)\frac{1}{1-u} - 4L(u)\frac{1}{1-u} + 4\frac{1}{1-u} - 4.
\end{aligned}$$

$$\begin{aligned}
II: \quad & 2\int_0^u -\log(1-t)\frac{t}{(1-t)^2}dt = 2\int_0^{-log(1-u)} re^r\left(1 - \frac{1}{e^r}\right)dr \\
&= 2\int_0^{-log(1-u)} re^r dr - 2\int_0^{-log(1-u)} r\ dr = 2L(u)\frac{1}{1-u} - 2\int_0^{-\log(1-u)} e^r dr - 2\frac{L^2(u)}{2} \\
&= 2L(u)\frac{1}{1-u} - 2\frac{1}{1-u} + 2 - L^2(u).
\end{aligned}$$

Fasst man alles zusammen, ergibt sich:

$$\begin{aligned}
f_2(u) &= 2L^2(u)\frac{1}{(1-u)^3} - 4L(u)\frac{1}{(1-u)^3} + 4\frac{1}{(1-u)^3} - 4\frac{1}{(1-u)^2} + \\
&\quad + 2L(u)\frac{1}{(1-u)^3} - 2\frac{1}{(1-u)^3} + 2\frac{1}{(1-u)^2} - L^2(u)\frac{1}{(1-u)^2} \\
&= 2L^2(u)\frac{1}{(1-u)^3} - 2L(u)\frac{1}{(1-u)^3} + 2\frac{1}{(1-u)^3} - 2\frac{1}{(1-u)^2} - L^2(u)\frac{1}{(1-u)^2}.
\end{aligned}$$

Da $\beta_1(n)$ der Koeffizient von u^n im Term $f_1(u)$ ist, wird die Darstellung von $f_1(u)$ im Folgenden angepasst. Dazu benötigt man die Potenzreihendarstellung von $-\log(1-u)$, die man erhält, indem man zuerst $-\log(1-u)$ ableitet, dann für die geometrische Reihe die bekannte Darstellung verwendet und anschließend diese Reihe gliedweise integriert.

$$\begin{aligned}
f_1(u) &= -\log(1-u)\frac{1}{1-u}\cdot\frac{1}{1-u} - \frac{1}{1-u}\cdot\frac{1}{1-u} + \frac{1}{1-u} \\
&= \sum_{n\geq 0}\frac{u^{n+1}}{n+1}\sum_{n\geq 0}u^n\sum_{n\geq 0}u^n - \sum_{n\geq 0}u^n\sum_{n\geq 0}u^n + \sum_{n\geq 0}u^n \\
&= \sum_{n\geq 0}\sum_{k=0}^{n}\frac{u^{k+1}}{k+1}u^{n-k}\sum_{n\geq 0}u^n - \sum_{n\geq 0}\sum_{k=0}^{n}u^k u^{n-k} + \sum_{n\geq 0}u^n \\
&= \sum_{n\geq 0}H_{n+1}u^{n+1}\sum_{n\geq 0}u^n - \sum_{n\geq 0}\sum_{k=0}^{n}u^n + \sum_{n\geq 0}u^n \\
&= \sum_{n\geq 0}\sum_{k=0}^{n}H_{k+1}u^{k+1}u^{n-k} - \sum_{n\geq 0}(n+1)u^n + \sum_{n\geq 0}u^n \\
&= \sum_{n\geq 0}\underbrace{(-1)}\left((n+2)\underbrace{H_{n+1}}_{=H_{n+2}-\frac{1}{n+2}} - (n+1)\right)u^{n+1} - \sum_{n\geq 0}(n+1)u^n + \sum_{n\geq 0}u^n \\
&= \sum_{n\geq 0}\left((n+1)H_{n+1} - (n+1)\right)u^n - \sum_{n\geq 0}(n+1)u^n + \sum_{n\geq 0}u^n.
\end{aligned}$$

Die vorletzte Zeile folgt aus Lemma 3.4.4 auf Seite 27 und die letzte Zeile aus einer Indexverschiebung der ersten Reihe. Abschließend wird also der gesuchte Koeffizient abgelesen.

$$\beta_1(n) = [u^n]f_1(n) = (n+1)\underbrace{H_{n+1}}_{=H_n+\frac{1}{n+1}} - (n+1) - (n+1) + 1 = (n+1)H_n - 2n.$$

Um auch die Koeffizienten von $f_2(u)$ ablesen zu können, sollen nun auch diese Summanden in eine Reihendarstellung gebracht werden. Dazu werden folgende zwei Formeln verwendet (siehe [4, Seite 14]).

Lemma 3.4.13 *Sei $L(u) = -\log(1-u)$. Dann gilt:*

$$L(u)\frac{1}{(1-u)^{m+1}} = \sum_{n\geq 0}(H_{n+m} - H_m)\binom{n+m}{m}u^n, \tag{3.10}$$

$$L^2(u)\frac{1}{(1-u)^{m+1}} = \sum_{n\geq 0}\left((H_{n+m} - H_m)^2 - (H^{(2)}_{n+m} - H^{(2)}_m)\right)\binom{n+m}{m}u^n. \tag{3.11}$$

Beweis. Ausgegangen wird von der Reihendarstellung der potenzierten geometrischen Reihe, die im Unterkapitel 1.4 in Lemma 1.4.7 auf Seite 9 gezeigt wurde (Idee stammt von [5]). Anschließend wird die Gleichung nach der Variablen m abgeleitet. Somit ergibt sich:

$$\begin{aligned}
\frac{1}{(1-u)^{m+1}} &= \sum_{n\geq 0}\binom{n+m}{n}u^n,\\
\frac{\partial}{\partial m}e^{(m+1)\log\frac{1}{(1-u)}} &= \sum_{n\geq 0}u^n\frac{\partial}{\partial m}\left(\frac{(n+m)(n+m-1)\cdots(m+1)}{n!}\right),\\
\frac{1}{(1-u)^{m+1}}\log\frac{1}{1-u} &= \sum_{n\geq 0}u^n\frac{1}{n!}\left(\sum_{k=1}^{n}\frac{1}{k+m}\right)(n+m)(n+m-1)\cdots(m+1)\\
&= \sum_{n\geq 0}\binom{n+m}{m}(H_{n+m} - H_m)u^n.
\end{aligned}$$

Bemerkung 3.4.14 *Im Unterkapitel 1.5 wurde in Definition 1.5.4 bereits die komplexe Erweiterung von den harmonischen Zahlen vorgestellt:*

$$H(x) = \sum_{k\geq 1}\left(\frac{1}{k} - \frac{1}{x+k}\right) \qquad \textit{für} \quad x \in \mathbb{C}\setminus\{0,-1,-2,...\}.$$

Damit erkennt man, dass die Formel auch für komplexe Wert m stimmt, denn es gilt:

$$\begin{aligned}
H(n+m) - H(m) &= \sum_{k\geq 1}\left(\frac{1}{k} - \frac{1}{n+m+k}\right) - \sum_{k\geq 1}\left(\frac{1}{k} - \frac{1}{m+k}\right)\\
&= \sum_{k\geq 1}\left(\frac{1}{m+k} - \frac{1}{n+m+k}\right) = \sum_{k=1}^{n}\frac{1}{m+k}.
\end{aligned}$$

Um die zweite Gleichung zu zeigen, wird noch einmal partiell nach m abgeleitet.

$$\begin{aligned}
\frac{\partial}{\partial m}\frac{1}{(1-u)^{m+1}}\log\frac{1}{1-u} &= \sum_{n\geq 0}\frac{u^n}{n!}\frac{\partial}{\partial m}\left(\sum_{k=1}^{n}\frac{1}{k+m}\right)(n+m)(n+m-1)\cdots(m+1)\\
\frac{1}{(1-u)^{m+1}}\left(\log\frac{1}{1-u}\right)^2 &= \sum_{n\geq 0}\frac{u^n}{n!}\left[\left(\sum_{k=1}^{n}\frac{-1}{(k+m)^2}\right)(n+m)(n+m-1)\cdots(m+1)\right.\\
&\quad\left.+\left(\sum_{k=1}^{n}\frac{1}{k+m}\right)^2(n+m)(n+m-1)\cdots(m+1)\right]\\
&= \sum_{n\geq 0}\binom{n+m}{m}\left(-(H^{(2)}_{n+m} - H^{(2)}_m) + (H_{n+m} - H_m)^2\right)u^n. \qquad \square
\end{aligned}$$

Da nur gewisse Spezialfälle von Interesse sind, sollen diese nun ermittelt werden. Zuerst wird $m = 1$ in Gleichung (3.11) eingesetzt und man erhält

$$\begin{aligned}
L^2(u)\frac{1}{(1-u)^2} &= \sum_{n\geq 0}\left((H_{n+1}-H_1)^2-(H_{n+1}^{(2)}-H_1^{(2)})\right)\binom{n+1}{1}u^n\\
&= \sum_{n\geq 0}\left(H_{n+1}^2-2H_{n+1}+1-H_{n+1}^{(2)}+1\right)(n+1)u^n\\
&= \sum_{n\geq 0}\left(\left(H_n+\frac{1}{n+1}\right)^2-2\left(H_n+\frac{1}{n+1}\right)+2-\left(H_n^{(2)}+\left(\frac{1}{n+1}\right)^2\right)\right)(n+1)u^n\\
&= \sum_{n\geq 0}\left(H_n^2+2H_n\frac{1}{n+1}-2H_n-\frac{2}{n+1}+2-H_n^{(2)}\right)(n+1)u^n\\
&= \sum_{n\geq 0}\left(H_n^2-H_n^{(2)}+2H_n\left(\frac{1}{n+1}-1\right)-\frac{2}{n+1}+\frac{2n+2}{n+1}\right)(n+1)u^n\\
&= \sum_{n\geq 0}\left((n+1)(H_n^2-H_n^{(2)})-2nH_n+2n\right)u^n.
\end{aligned}$$

Nun soll in Gleichung (3.10) $m = 2$ eingesetzt werden. Nachdem $H_2 = 1+\frac{1}{2} = \frac{3}{2}$ gilt, ergibt sich:

$$\begin{aligned}
L(u)\frac{1}{(1-u)^3} &= \sum_{n\geq 0}(H_{n+2}-H_2)\binom{n+2}{2}u^n\\
&= \sum_{n\geq 0}\left(H_n+\frac{1}{n+1}+\frac{1}{n+2}-\frac{3}{2}\right)\frac{(n+2)(n+1)}{2}u^n\\
&= \sum_{n\geq 0}\left(\binom{n+2}{2}H_n+\frac{n+2}{2}+\frac{n+1}{2}-\frac{3(n^2+3n+2)}{4}\right)u^n\\
&= \sum_{n\geq 0}\left(\binom{n+2}{2}H_n+n+\frac{3}{2}-\frac{3n^2}{4}-\frac{9n}{4}-\frac{3}{2}\right)u^n\\
&= \sum_{n\geq 0}\left(\binom{n+2}{2}H_n-\frac{3}{4}n^2-\frac{5}{4}n\right)u^n.
\end{aligned}$$

Abschließend muss noch $m = 2$ in Gleichung (3.11) eingesetzt werden. Mit $H_2^{(2)} = 1^2+\left(\frac{1}{2}\right)^2 = 1+\frac{1}{4} = \frac{5}{4}$ erhält man:

$$
\begin{aligned}
L^2(u)\frac{1}{(1-u)^3} &= \sum_{n\geq 0}\left((H_{n+2}-H_2)^2-\left(H^{(2)}_{n+2}-H^{(2)}_2\right)\right)\binom{n+2}{2}u^n\\
&= \sum_{n\geq 0}\left(H^2_{n+2}-3H_{n+2}+\frac{9}{4}-H^{(2)}_{n+2}+\frac{5}{4}\right)\frac{(n+2)(n+1)}{2}u^n\\
&= \sum_{n\geq 0}\left(H^2_{n+1}+2H_{n+1}\frac{1}{n+2}-3\left(H_{n+1}+\frac{1}{n+2}\right)+\frac{7}{2}-H^{(2)}_{n+1}\right)\frac{(n+2)(n+1)}{2}u^n\\
&= \sum_{n\geq 0}\left(H^2_{n}+2H_{n}\frac{1}{n+1}+H_{n+1}\left(\frac{2}{n+2}-3\right)-\frac{3}{n+2}+\frac{7}{2}-H^{(2)}_{n}\right)\binom{n+2}{2}u^n\\
&= \sum_{n\geq 0}\left(H^2_{n}-H^{(2)}_{n}+H_n\left(\frac{2}{n+1}+\frac{2}{n+2}-3\right)+\frac{2}{(n+2)(n+1)}-\frac{3}{n+1}-\frac{3}{n+2}+\frac{7}{2}\right)\binom{n+2}{2}u^n\\
&= \sum_{n\geq 0}\left(\binom{n+2}{2}\left(H^2_n-H^{(2)}_n\right)+\frac{1}{2}H_n\left(2n+4+2n+2-3n^2-9n-6\right)\right.\\
&\quad \left.+\frac{1}{2}\left(2-3n-6-3n-3+\frac{7}{2}\left(n^2+3n+2\right)\right)\right)u^n\\
&= \sum_{n\geq 0}\left(\binom{n+2}{2}\left(H^2_n-H^{(2)}_n\right)+\frac{1}{2}H_n\left(-3n^2-5n\right)+\frac{1}{4}\left(-14-12n+7n^2+21n+14\right)\right)u^n\\
&= \sum_{n\geq 0}\left(\binom{n+2}{2}\left(H^2_n-H^{(2)}_n\right)-\frac{n}{2}(5+3n)H_n+\frac{7}{4}n^2+\frac{9}{4}n\right)u^n.
\end{aligned}
$$

Mit Hilfe von den gerade berechneten Termen

$$
\begin{aligned}
L^2(u)\frac{1}{(1-u)^2} &= \sum_{n\geq 0}\left((n+1)(H^2_n-H^{(2)}_n)-2nH_n+2n\right)u^n,\\
L(u)\frac{1}{(1-u)^3} &= \sum_{n\geq 0}\left(\binom{n+2}{2}H_n-\frac{3}{4}n^2-\frac{5}{4}n\right)u^n,\\
L^2(u)\frac{1}{(1-u)^3} &= \sum_{n\geq 0}\left(\binom{n+2}{2}(H^2_n-H^{(2)}_n)-\frac{n}{2}(5+3n)H_n+\frac{7}{4}n^2+\frac{9}{4}n\right)u^n
\end{aligned}
$$

und mit der Umformung

$$
\begin{aligned}
\frac{1}{(1-u)^3} &= \sum_{n\geq 0}u^n\sum_{n\geq 0}u^n\sum_{n\geq 0}u^n = \sum_{n\geq 0}\sum_{k=0}^{n}u^k u^{n-k}\sum_{n\geq 0}u^n = \sum_{n\geq 0}\sum_{k=0}^{n}u^n\sum_{n\geq 0}u^n\\
&= \sum_{n\geq 0}(n+1)u^n\sum_{n\geq 0}u^n = \sum_{n\geq 0}\sum_{k=0}^{n}(k+1)u^k u^{n-k} = \sum_{n\geq 0}\sum_{k=0}^{n}(k+1)u^n\\
&= \sum_{n\geq 0}\binom{n+2}{2}u^n
\end{aligned}
$$

kann man jetzt $f_2(u)$ folgendermaßen umschreiben:

$$\begin{aligned}
f_2(u) =& 2L^2(u)\frac{1}{(1-u)^3} - 2L(u)\frac{1}{(1-u)^3} + 2\frac{1}{(1-u)^3} - L^2(u)\frac{1}{(1-u)^2} - 2\frac{1}{(1-u)^2} \\
=& 2\sum_{n\geq 0}\left(\binom{n+2}{2}(H_n^2 - H_n^{(2)}) - \frac{n}{2}(5+3n)H_n + \frac{7}{4}n^2 + \frac{9}{4}n\right)u^n \\
& - 2\sum_{n\geq 0}\left(\binom{n+2}{2}H_n - \frac{3}{4}n^2 - \frac{5}{4}n\right)u^n + 2\sum_{n\geq 0}\binom{n+2}{2}u^n \\
& - \sum_{n\geq 0}\Big((n+1)(H_n^2 - H_n^{(2)}) - 2nH_n + 2n\Big)u^n - 2\sum_{n\geq 0}(n+1)u^n.
\end{aligned}$$

Jetzt kann der gesuchte Koeffizient und somit das zweite Moment abgelesen werden:

$$\begin{aligned}
\beta_2(n) = [u^n]f_2(n) =& (n+2)(n+1)(H_n^2 - H_n^{(2)}) - (5n+3n^2)H_n + \frac{7}{2}n^2 + \frac{9}{2}n \\
& - (n+2)(n+1)H_n + \frac{3}{2}n^2 + \frac{5}{2}n + (n+2)(n+1) \\
& - (n+1)(H_n^2 - H_n^{(2)}) + 2nH_n - 2n - 2(n+1) \\
=& (n+1)^2(H_n^2 - H_n^{(2)}) - H_n(5n+3n^2+n^2+3n+2-2n) \\
& + n^2\left(\frac{7}{2}+\frac{3}{2}+1\right) + n\left(\frac{9}{2}+\frac{5}{2}+3-4\right) + 2 - 2 \\
=& (n+1)^2(H_n^2 - H_n^{(2)}) - H_n(4n^2+6n+2) + 6n^2 + 6n \\
=& (n+1)^2(H_n^2 - H_n^{(2)}) - (4n+2)(n+1)H_n + 6n(n+1). \qquad \square
\end{aligned}$$

Nun hat man alle Werkzeuge, um die Varianz auszurechnen und es ergibt sich:

$$\begin{aligned}
\mathbb{V}(a) =& \beta_2(n) + \beta_1(n) - \beta_1^2(n) \\
=& (n+1)^2(H_n^2 - H_n^{(2)}) - (4n+2)(n+1)H_n + 6n(n+1) + (n+1)H_n - 2n \\
& - ((n+1)H_n - 2n)^2 \\
=& (n+1)^2(H_n^2 - H_n^{(2)}) - (4n+1)(n+1)H_n + 6n^2 + 4n \\
& - \big((n+1)^2H_n^2 - 4n(n+1)H_n + 4n^2\big) \\
=& -(n+1)^2H_n^{(2)} - (n+1)H_n + 2n^2 + 4n.
\end{aligned}$$

3.5 Asymptotik

Unter Verwendung der Integralformel (3.9) aus Korollar 3.4.11 auf Seite 30 kann man $f_s(u)$ für jedes beliebig große s berechnen. Mit wachsendem s werden die Terme jedoch wesentlich komplizierter. Es soll hier nun der Führungskoeffizient, also der Koeffizient der höchsten Potenz, einer asymptotischen Entwicklung um die Singularität $u = 1$ gefunden werden. Die entscheidende Beobachtung hierbei ist, dass $f_s(u)$ eine Linearkombination von Termen der Form $L^i(u)\frac{1}{(1-u)^{j+1}}$ ist.

Satz 3.5.1 *$f_s(u)$ sei wie in Definition 3.4.9, Gleichung (3.7), auf Seite 28 definiert. Sei $\mathfrak{R}_{p,q}(u)$ eine nicht näher definierte Linearkombination der Form $L^i(u)\frac{1}{(1-u)^{j+1}}$ mit $i,j \in \mathbb{Z}$ und entweder $j < q$ und i beliebig, oder $j = q$ und $i \leq p$. Dann gilt:*

$$f_s(u) = s!L^s(u)\frac{1}{(1-u)^{s+1}} + \mathfrak{R}_{s-1,s}(u). \tag{3.12}$$

Beweis. Um induktiv den Satz für alle s zu zeigen, starten wir mit dem Induktionsanfang bei $s = 0$. Hier gilt die Aussage wegen $f_0(u) = \frac{1}{1-u}$. Laut Induktionsannahme ist der Satz gültig für alle j mit $0 \leq j \leq s-1$. Nun soll im Induktionsschritt gezeigt werden, dass die Gleichung auch für s gilt. Dabei ist folgende Beobachtung wesentlich. Wir definieren einen Term $g(u)$:

$$g(u) := cq!L^p(u)\frac{1}{(1-u)^{q+1}} + \mathfrak{R}_{p-1,q}(u).$$

Betrachtet man jetzt die erste Ableitung von $g(u)$, wobei

$$(L^p(u))' = ((-\log(1-u))^p)' = p(-\log(1-u))^{p-1}\frac{1}{1-u}$$

gilt, also

$$\begin{aligned} g'(u) &= cq!\left(\underbrace{pL^{p-1}(u)\frac{1}{(1-u)^{q+2}}}_{\in \mathfrak{R}_{p-1,q+1}(u)} + L^p(u)(q+1)\frac{1}{(1-u)^{q+2}}\right) + \mathfrak{R}_{p-1,q+1}(u) \\ &= c(q+1)!L^p(u)\frac{1}{(1-u)^{q+2}} + \mathfrak{R}_{p-1,q+1}(u), \end{aligned}$$

erkennt man, dass für die i-te Ableitung folgendes gelten muss:

$$g^{(i)}(u) = c(q+i)!L^p(u)\frac{1}{(1-u)^{q+i+1}} + \mathfrak{R}_{p-1,q+i}(u).$$

Im Speziellen folgt daraus für $j \leq s-1$:

$$f_j^{(i)}(u) = (j+i)!L^j(u)\frac{1}{(1-u)^{j+i+1}} + \mathfrak{R}_{j-1,j+i}(u).$$

Eingesetzt in die Gleichung für $h_s(u)$ aus Satz 3.4.10 auf Seite 29 ergibt das:

$$\begin{aligned} h_s(u) = &\sum_{i=1}^{s-1}\binom{s}{i}\left(i!L^i(u)\frac{1}{(1-u)^{i+1}} + \mathfrak{R}_{i-1,i}(u)\right)\sum_{r=0}^{s-1}\binom{s-i}{r}u^r \\ &\cdot\left((s-i)!L^{s-i-r}(u)\frac{1}{(1-u)^{s-i+1}} + \mathfrak{R}_{s-i-r-1,s-i}(u)\right) \\ &+ \frac{1}{1-u}\sum_{r=1}^{s}\binom{s}{r}u^r\left(s!L^{s-r}(u)\frac{1}{(1-u)^{s+1}} + \mathfrak{R}_{s-r-1,s}(u)\right). \end{aligned}$$

Beachtet man die oberen und unteren Grenzen der Summen, erkennt man, dass alle Terme $\mathfrak{R}_{p,q}(u)$ und die zweite Summe in $\mathfrak{R}_{s-1,s+1}(u)$ liegen müssen. Übrig bleibt die mit Hilfe des binomischen Lehrsatzes gefundene Form:

$$s!L^s(u)\frac{1}{(1-u)^{s+2}}\sum_{i=1}^{s-1}\left(1+\frac{u}{L(u)}\right)^{s-i} = s!(s-1)L^s(u)\frac{1}{(1-u)^{s+2}} + \mathfrak{R}_{s-1,s+1}(u)$$

übrig und somit ist $h_s(u)$ abgearbeitet. Eingesetzt in die Integralformel (3.9) auf Seite 30 ergibt das durch partielle Integration

$$\begin{aligned} f_s(u) &= \frac{1}{(1-u)^2}\int_0^u s!(s-1)L^s(t)\frac{1}{(1-t)^s}dt + \frac{1}{(1-u)^2}\int_0^u \mathfrak{R}_{s-1,s-1}(t)dt \\ &= s!L^s(u)\frac{1}{(1-u)^{s+1}} + \mathfrak{R}_{s-1,s}(u). \end{aligned}$$ □

Der nächste Schritt ist also die asymptotische Abschätzung des Führungsterms. Hierbei wird nicht wie in der Arbeit von P. Kirschenhofer, H. Prodinger und R. F. Tichy (siehe [3]) vorgegangen und es werden keine Tauber'schen Sätze verwendet, sondern die erst zeitlich später entwickelte Singularitätenanalyse von P. Flajolet und A. Odlyzko (siehe [6]) angewandt.

Definition 3.5.2 (Gammafunktion) *Für $x \in \mathbb{R}, x > 0$ ist die Gammafunktion $\Gamma(x)$ so definiert:*

$$\Gamma(x) = \int_0^\infty t^{x-1}e^{-t}dt.$$

Die Gammafunktion erfüllt die Funktionalgleichung $\Gamma(x+1) = x\Gamma(x)$ und kann aufgrund von $\Gamma(1) = 1$ als eine Verallgemeinerung der Fakultätfunktion angesehen werden. Sie kann auch mit

$$\Gamma(x) = \lim_{n\to\infty}\frac{n!n^x}{x(x+1)(x+2)\cdots(x+n)}$$

auf den Wertebereich $x \in \mathbb{R} \setminus \{0, -1, -2, ...\}$ ausgeweitet werden (siehe [18, Seiten 406-407]).

Satz 3.5.3 *Seien α und $\gamma \in \mathbb{R}$ und $\alpha \notin \mathbb{N}$. Die Funktion $f(z)$ sei gegeben durch*

$$f(z) = (1-z)^\alpha\left(\frac{1}{z}\log\frac{1}{1-z}\right)^\gamma.$$

Dann erfüllen die Taylorkoeffizienten von $f(z)$

$$f_n = [z^n]f(z) \sim \frac{n^{-\alpha-1}}{\Gamma(-\alpha)}(\log n)^\gamma\left(1+\sum_{k\geq 1}\frac{e_k^{(\alpha,\gamma)}}{\log^k n}\right),$$

mit

$$e_k^{(\alpha,\gamma)} = (-1)^k\binom{\gamma}{k}\Gamma(-\alpha)\frac{d^k}{ds^k}\left(\frac{1}{\Gamma(-s)}\right)\Big|_{s=\alpha}.$$

Bemerkung 3.5.4 *Hier steht das Zeichen "$\sim$" für asymptotische Gleichheit. Diese ist so definiert:* $f(x) \sim g(x)$ *ist äquivalent mit* $\lim_{x\to\infty} \frac{f(x)}{g(x)} = 1$.
In der Arbeit von Flajolet/Odlyzko (siehe [6]) wird der Beweis für $\alpha, \gamma \in \mathbb{C}$ *geführt. Für unsere Zwecke reicht es aber aus, wenn* α *und* γ *reelle Zahlen sind.*

Satz 3.5.3 kann auf den interessanten ersten Summanden der Resttermdarstellung (3.12) aus Satz 3.5.1 angewandt werden, da $(1-z)^\alpha \left(\frac{1}{z}\log\frac{1}{1-z}\right)^\gamma$ und $(1-z)^\alpha \left(\log\frac{1}{1-z}\right)^\gamma$ gleiches asymptotisches Verhalten aufweisen. Setzt man also $\alpha = -s-1$ und $\gamma = s$ erhält man, mit $e_k^{(\alpha,\gamma)}$ definiert wie in Satz 3.5.3, als Resultat

$$[u^n]s!L^s(u)\frac{1}{(1-u)^{s+1}} \sim s!\frac{n^s}{\Gamma(s+1)}(\log n)^s\left(1+\sum_{k\geq 1}\frac{e_k^{(-s-1,s)}}{\log^k n}\right).$$

Da $\Gamma(s+1) = s!$ für $s \in \mathbb{N}$ gilt, ist die höchste auftretende Potenz $n^s \log^s n$. Mit diesen Überlegungen kann man nun folgenden Satz formulieren.

Satz 3.5.5 $f_s(u)$ *sei wie in Definition 3.4.9, Gleichung* (3.7)*, auf Seite 28 definiert. Dann verhält sich* $[u^n]f_s(u)$ *asymptotisch wie* $n^s \log^s n$. *Es gilt also:*

$$[u^n]f_s(u) \sim n^s \log^s n \quad \textit{bzw.} \quad \lim_{n\to\infty}\frac{[u^n]f_s(u)}{n^s\log^s n} = 1.$$

3.5.1 Grenzverteilung

Abschließend soll folgendes Resultat bezüglich Grenzverteilung präsentiert werden. Bezeichnet man mit X_n die Zufallsvariable für den Parameterwert a zu einer zufällig gewählten Permutation von $\{1, ..., n\}$, dann folgt aufgrund der Arbeit über die Quicksort-Rekursion von H. K. Hwang und R. Neininger, siehe [16], folgender Grenzverteilungssatz:

$$\frac{X_n - E(X_n)}{n} \xrightarrow{d} Y$$

wobei die Verteilung von Y durch folgende Fixpunktgleichung charakterisiert ist:

$$Y \stackrel{d}{=} UY + (1-U)Y^* + U\log U + (1-U)\log(1-U) + U.$$

Dabei bezeichnet Y^* eine unabhängig und identisch wie Y verteilte Zufallsvariable und U eine unabhängig von Y und Y^* auf $[0,1]$ gleichverteilte Zufallsvariable.

Kapitel 4

Varianten und Modifikationen des klassischen Algorithmus

Im vorigen Kapitel haben wir uns sehr ausführlich damit beschäftigt, welche durchschnittliche Laufzeit der klassische Algorithmus hat. Es wurde aber auch auf den worst-case Fall eingegangen, der bei $O(n^2)$ liegt. Das Ziel in diesem Abschnitt ist es, diesen worst-case Fall zu verbessern. Dabei werden die Resultate von F. E. Fich, J. I. Munro und P. V. Poblete (siehe [1]) vorgestellt und untersucht.

4.1 Modifikation "In place"-Permutation

Da die In place-Permutation eine spezielle In situ-Permutation ist, bleibt auch die Problemstellung im Wesentlichen die gleiche. Wir wollen in diesem Abschnitt aber trotzdem die Aufgabenstellung ein wenig modifizieren. Es soll in diesem Kapitel das Datenfeld $A = [a_1, a_2, ..., a_n]$ nicht wie bisher in $A_{neu} = [a_{\pi(1)}, a_{\pi(2)}, ..., a_{\pi(n)}]$ umgespeichert werden, sondern es sollen nun die n Operationen

$$A[\pi(i)] \leftarrow A[i] \text{ für } i \in \{1, ..., n\}$$

gleichzeitig ausgeführt werden. In diesem Fall soll also das Ergebnis des Algorithmus $A_{neu} = [a_{\pi^{-1}(1)}, a_{\pi^{-1}(2)}, ..., a_{\pi^{-1}(n)}]$ sein. Der Rest bleibt gleich: π ist wieder eine nicht näher spezifizierte Permutation, die beliebig vorgegeben ist und im Laufe des Algorithmus nicht verändert werden darf (das kommt etwa vor, wenn ein anderes Programm gleichzeitig auf diese Permution zugreifen muss).

In place bedeutet jetzt, dass das Umspeichern erreicht wird, indem man wiederholt Einträge im Datenfeld A miteinander paarweise vertauscht. Im Mittelpunkt steht also die Prozedur

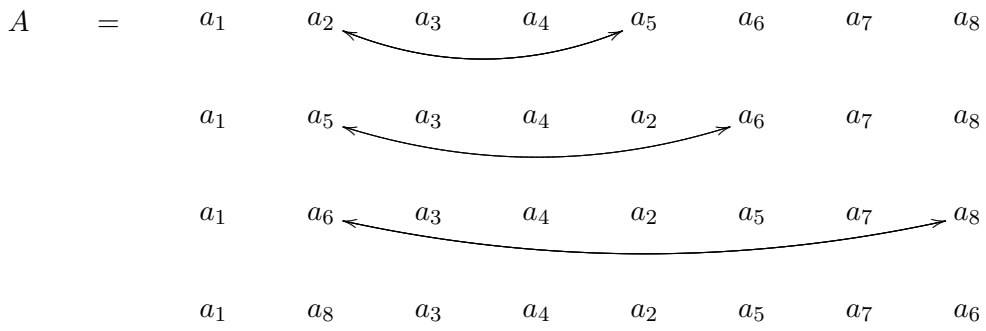
Algorithmus 4.1.1

```
1  procedure ROTATE_{π^-1}(leader)
2  i := π(leader)
3  while i ≠ leader do
4     interchange the values in A[leader] and A[i]
5     i := π(i)
6  end while.
```

Übergeben wird der Index des Zyklenführers, der in der Hilfsvariablen *leader* gespeichert wird. Hier wird jedes Element nach dem Zyklenführer sukzessive an diese Position *leader* getauscht und

dann anschließend an die endgültig richtige Stelle im Datenfeld. Ein Beispiel soll den Sachverhalt verdeutlichen.

Beispiel 4.1.2 *Es sei die Permutation* $\pi = (1)(2\ 5\ 6\ 8)(3\ 4\ 7)$ *in Zyklendarstellung gegeben. Für den Zyklenführer 2 soll jetzt die Prozedur* $\mathrm{ROTATE}_{\pi^{-1}}(2)$ *aufgerufen werden.*



Nachdem also jedes Element im Zyklus genau zweimal bewegt wird – mit Außnahme des Elements mit dem kleinsten und des mit dem größten Index, die je einmal bewegt werden – und zu jeder Vertauschung zwei Elemente gehören, benötigt die Prozedur genau so viele Schritte, wie der Zyklus lang ist. Nachdem jeder Zyklenführer abgearbeitet werden muss, wird jeder Zyklus durchschritten und die Laufzeit ist daher proportional zur Anzahl der Elemente und liegt somit in $O(n)$.

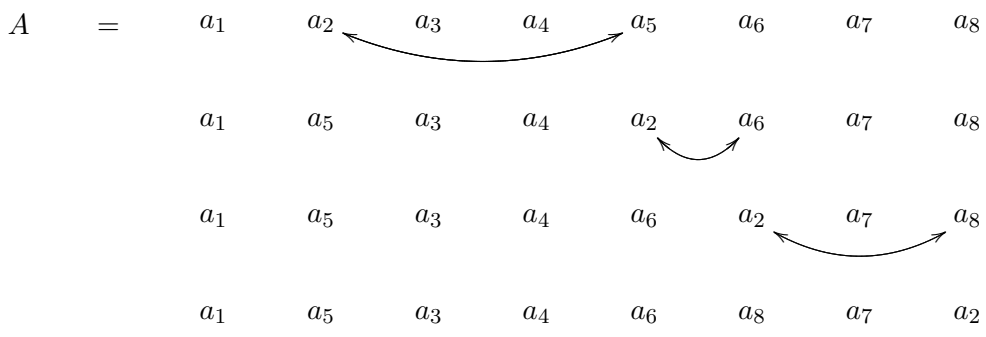
Man kann die Prozedur auch leicht für die ursprüngliche Problemstellung angeben. Will man also $A_{neu} = [a_{\pi(1)}, a_{\pi(2)}, ..., a_{\pi(n)}]$ aus $A = [a_1, a_2, ..., a_n]$ erhalten, dann verwendet man diese Prozedur:

Algorithmus 4.1.3

```
1  procedure ROTATE_π(leader)
2  i := leader
3  while π(i) ≠ leader do
4     interchange the values in A[i] and A[π(i)]
5     i := π(i)
6  end while.
```

Hier wird jedes Element bis auf jenes, das als Index den Zyklenführer hat, genau einmal bewegt. Dieses spezielle Element selbst ist bei jeder Vertauschung beteiligt und wandert so bis an das Ende des Zyklus, wo es seinen endgültigen Platz findet. Für obiges Beispiel würde das also fogendermaßen aussehen.

Beispiel 4.1.4 (Fortsetzung von Beispiel 4.1.2) *Für dieselbe Permutation* $\pi = (1)(2\ 5\ 6\ 8)(3\ 4\ 7)$ *soll nun die Prozedur* $\mathrm{ROTATE}_{\pi}(2)$ *aufgerufen werden.*



Man sieht also, dass man auch hier wieder n Vertauschungen benötigt, wenn man alle Zyklen abarbeitet. Somit ist klar, wo man bei der Optimierung von Algorithmen, die entlang von Zyklen operieren, ansetzen muss. Es gilt, möglichst effizient die Zyklenführer zu bestimmen. Mit Hilfe der folgenden Varianten soll zunächst das worst-case Szenario laufzeittechnisch verbessert werden.

4.2 Variante mit einem Bit-Vektor der Länge b

Wie schon in der Motivation der Arbeit angedeutet, ist es bei In situ-Permutationsalgorithmen wesentlich, dass für das Umspeichern des Datenfeldes weniger als $O(n)$ Bits zusätzlicher Hilfsspeicher benötigt wird. In diesem Abschnitt wird klar, warum diese Forderung Sinn macht – würde man sie weglassen, hätte man sofort lineare Laufzeit, wie wir sehen werden. Trotzdem wollen wir nun ein wenig mehr Speicher zulassen, um die Laufzeit des worst-case Szenarios verringern zu können. Dabei soll uns ein Bit-Vektor der Länge b helfen, also ein Vektor, dessen Einträge entweder 0 oder 1 sind.

Satz 4.2.1 *Im schlechtesten Fall (worst-case) lässt sich ein Datenfeld der Länge n unter Verwendung von $b+O(\log_2(n))$ Bits Hilfsspeicher mit $O(\frac{1}{b}n^2)$ Vergleichen umspeichern, wobei $b \leq n$ und der Hilfsspeicher aus einem Bit-Vektor der Länge b und aus einer konstanten Anzahl an Indexvariablen besteht.*

Beweis. Es ist nicht von Bedeutung, welche Einträge tatsächlich im Datenfeld A stehen, da es reicht, wenn man mit den Indizes (pointern) zu diesen Einträgen arbeitet. Die Indexvariablen müssen in der Lage sein, alle möglichen Positionen im Datenfeld A wiederzugeben, also die Zahlen von 1 bis n aufzunehmen. Daher muss die Bit-Folge für die Indexvariablen die Länge $\log_2(n)$ haben, um die Zahlen bis n binär darstellen zu können. Es wird lediglich eine konstante Anzahl an Indexvariablen benötigt, daher erhält man $O(\log_2(n))$ Bits zusätzlichen Hilfsspeicher.
Die Idee ist nun, dass man das Datenfeld A in $\lceil \frac{n}{b} \rceil$ Teile aufteilt. Jeder dieser Teile hat die Größe b, außer der letzte, der eventuell kleiner sein kann. Im Bit-Vektor V wird ständig mitgespeichert, welche Plätze man beim Durchlaufen eines Zyklus bereits passiert hat, für die man weiß, dass es bereits ein kleineres Element im jeweiligen Zyklus gibt. Somit hat man eine Ersparnis, wenn man ein Element daraufhin untersucht, ob es ein Zyklenführer ist, und man im Vektor V abliest, dass man bereits beim Abarbeiten eines vorigen Zyklus an diese Stelle bei der Untersuchung eines kleineren Elements gekommen ist. □

Das folgende Beispiel soll die Arbeitsweise mit so einem Bit-Vektor veranschaulichen. Weiters soll gleich der Spezialfall $b = n$ beleuchtet werden. Dabei wird die Aussage aus der Einführung zur Problemstellung dieser Diplomarbeit verifiziert – dass man nämlich sofort lineare Laufzeit erhält, wenn man linear großen Hilfsspeicher zulässt.

Beispiel 4.2.2 *Es sei die Permutation* $\pi = (4, 1, 5, 2, 3)$ *gegeben. Nun sollen die Zyklenführer mit Hilfe des Bit-Vektors* $V = [0, 0, 0, 0, 0]$ *gefunden werden. Dabei startet man mit dem kleinsten Element* 1*, was* $V = [1, 0, 0, 0, 0]$ *ergibt und bewegt sich entlang des ersten Zyklus zum nächsten Element, also zur* 4*. Da* $4 > 1$ *ergibt sich* $V = [1, 0, 0, 1, 0]$*, weil sich* 4 *in einem Zyklus befindet, wo es offensichtlich ein kleineres Element, hier also* 1*, gibt. Auch* $\pi(4) = 2$ *ist größer als* 1 *und somit erhält man* $V = [1, 1, 0, 1, 0]$*. Mit* $\pi(2) = 1$ *hat man den ersten Zyklus* (1 4 2) *durchschritten und erhält* 1 *als ersten Zyklenführer. Normalerweise würde die Zyklenführer-Überprüfung mit* 2 *fortfahren, doch der Eintrag* 1 *an der zweiten Stelle im Bit-Vektor* V *verrät, dass man* 2 *schon zuvor bei einem Zyklus mit kleinerem Element durchschritten hat. Also wird* 2 *jetzt nicht mehr überprüft und es geht mit* 3 *weiter, was zu* $V = [1, 1, 1, 1, 0]$ *führt. Mit* $\pi(3) = 5 > 3$ *erhält man* $V = [1, 1, 1, 1, 1]$ *und erkennt anschließend mit* $\pi(5) = 3$*, dass der Zyklus abgearbeitet wurde. Der zweite Zyklenführer ist also* 3*. Bei allen folgenden Elementen unterbindet das Datenfeld* V *eine weitere Behandlung – es wird also jedes Element nur einmal im jeweiligen Zyklus untersucht und anschließend nie wieder behandelt, was zu linearer Laufzeit führt.*

Die Überlegungen aus Satz 4.2.1 motivieren diesen Algorithmus:

Algorithmus 4.2.3 (Bit-Vektor)

```
 1  for k := 1 to ⌈n/b⌉ do
 2    s := (k − 1)b
 3    if k ≤ ⌊n/b⌋ then
 4      l := b
 5    else
 6      l := n − b⌊n/b⌋
 7    end if
 8    for i := 1 to l do
 9      V[i] := 0
10    end for
11    for i := 1 to l do
12      if V[i] = 0 then
13        V[i] := 1
14        j := π(s + i)
15        while j > s + i do
16          if j ≤ s + l then
17            V[j − s] := 1
18          end if
19          j := π(j)
20        end while
21        if j = s + i then
22          ROTATE(s + i)
23        end if
24      end if
25    end for
26  end for.
```

Der Bit-Vektor V startet mit dem Index 1. Die Hilfsvariable k durchläuft alle Regionen, für die dann die Variable s stets den um eins verringerten Startindex der jeweiligen Region speichert; die Variable l nimmt die Länge der Region auf. In Zeile 8 bis 10 wird der Bit-Vektor V auf den Nullvektor zurückgesetzt. Ab Zeile 11 wird jetzt die aktuelle Region abgearbeitet. Zeile 13 könnte man weglassen – sie steht nur aus ästhetischen Gründen im Code. So ist nämlich sicher gestellt, dass jedes Element der aktuellen Region, das abgearbeitet wurde, im Bit-Vektor V das zugehörige Bit auf 1 gesetzt bekommt. Die eigentliche Arbeit findet in Zeile 15 bis 20 statt. Hier durchwandert man den Zyklus und überprüft in Zeile 16 zusätzlich, ob man dabei auf einen Index stößt, der sich in der aktuell behandelten Region befindet. Wenn ja, wird der Vektor V in Zeile 17 aktualisiert.

Bemerkung 4.2.4 *Hat man n Bits zusätzlichen Hilfsspeicher, kann man die Einträge im Bit-Vektor V auch in der* ROTATE*-Prozedur aktualisieren. Dabei werden jene Einträge auf 1 gesetzt, deren zugehörige Plätze durch die Prozedur umgespeichert werden. Nachdem man die Elemente vom kleinsten weg betrachtet, muss man nur jene auf die Zyklenführereigenschaft überprüfen, die im Bit-Vektor noch eine Null stehen haben.*

4.3 Variante mit π und π^{-1}

Ist nicht nur die Permutation π selbst gegeben, sondern hat man auch noch die dazu inverse Permutation, also π^{-1}, so kann man einen effizienteren Algorithmus finden. Es ist nun nämlich möglich, den Zyklus in zwei Richtungen zu durchlaufen. Dadurch findet man ein größeres Element – sollte es eines geben – im worst-case um einiges schneller.

Satz 4.3.1 *Sei die Permutation π und die dazu inverse Permutation π^{-1} gegeben. Dann kann man ein Datenfeld der Größe n im worst-case mit $O(n \log_2 n)$ Vergleichen umspeichern. Der Hilfsspeicher beträgt $O(\log_2 n)$ Bits.*

Für den Beweis sei auf das Problem verwiesen, wie man in einem Ring das kleinste bzw. größte Element finden kann. In der Beschreibung von Hirschberger und Sinclair (siehe [7]) werden Abschätzungen über den Informationsfluss in beide Richtungen einer Ringstruktur gemacht, wobei stets Pfade der Länge 2^j betrachtet werden. Für einen Index i betrachtet man die Elemente entlang des Pfades 2^j nur dann, wenn man vorher noch kein kleineres Element gefunden hat, also die 2^{j-1} Vorgänger-Elemente von i, sowie die 2^{j-1} Nachfolger von i müssen stets größer sein als i. Hat man also $2^{j-1}+1$ aufeinanderfolgende Zyklenelemente, kann es nur ein Element geben, für das man seine 2^j Vorgänger und Nachfolger betrachten muss. Bezeichne k die aktuelle Zyklenlänge. Für jedes Element überprüft man seinen unmittelbaren Vorgänger und Nachfolger. Maximal $\lfloor \frac{k}{2} \rfloor$ Elemente überprüfen zwei Vorgänger und Nachfolger – für die andere Hälfte ist der unmittelbare Nachbar bereits größer. Bei einem Pfad der Länge 4 sind es nur mehr maximal $\lfloor \frac{k}{3} \rfloor$, bei einem Pfad der Länge 8 nur mehr maximal $\lfloor \frac{k}{5} \rfloor$ aller Elemente des Zyklus. Algemein gibt es also maximal so viele Überprüfungen:

$$2\left(1k + 2\left\lfloor\frac{k}{2}\right\rfloor + 4\left\lfloor\frac{k}{3}\right\rfloor + 8\left\lfloor\frac{k}{5}\right\rfloor + ... + 2^j \underbrace{\left\lfloor\frac{k}{2^{j-1}+1}\right\rfloor}_{\leq \frac{k}{2^{i-1}}}\right). \tag{4.1}$$

Der Vorfaktor 2 kommt von den beiden Richtungen, in die überprüft wird – entlang π und entlang π^{-1}. Es gibt maximal $1 + \lfloor \log_2(k) \rfloor$ Summanden, denn setzt man $j = 1 + \lfloor \log_2(k) \rfloor$ in

den Term (4.1) ein, erkennt man, aufgrund der Gültigkeit von $\left\lfloor \frac{k}{2^{1+\lfloor \log_2(k) \rfloor - 1}+1} \right\rfloor = \left\lfloor \frac{k}{k+1} \right\rfloor = 0$, dass es kein Element gibt, welches $2^{1+\lfloor \log_2(k) \rfloor} = 2k$ oder gar mehr Vorgänger und Nachfolger überprüft. Spätestens nach k Schritten kann die Suche nämlich beendet werden, da man im Zyklus wieder beim Anfangselement i angekommen ist.
Alle Summanden in der Abschätzung für die maximale Anzahl an Überprüfungen sind kleiner als $2k$ und somit ergibt sich als obere Abschätzung für die Überprüfungen:

$$2\,(2k(1+\log_2(k))) = 4k + 4(k\log_2(k)) = O(k\log_2(k)).$$

Nachdem die Summe der Länge aller Zyklen n ergibt, erhält man für die Laufzeit:

$$4(k_1 + k_1 \underbrace{\log_2(k_1)}_{\leq \log_2(n)} + k_2 + k_2 \underbrace{\log_2(k_2)}_{\leq \log_2(n)} + ...) \leq 4n + 4(\underbrace{k_1 + k_2 + ...}_{=n})\log_2(n) = O(n\log_2(n)).$$

□

Eine Implementierung dieser Variante ist:

Algorithmus 4.3.2 (Permutation mit inverser Permutation)

```
 1  for i := 1 to n do
 2    j := π(i)
 3    if j ≠ i then
 4      k := π^{-1}(i)
 5      while i < j and i < k do
 6        if j = k then
 7          ROTATE(i)
 8          exit
 9        end if
10        j := π(i)
11        if j = k then
12          ROTATE(i)
13          exit
14        end if
15        k := π^{-1}(k)
16      end while
17    end if
18  end for.
```

In Zeile 3 wird überprüft, ob es sich bei i um einen Fixpunkt handelt. Sollte dies nicht der Fall sein, bewegt man sich also in der while-Schleife in den Zeilen 5 bis 16, bis man entweder ein kleineres Element gefunden hat, oder man auf einen der beiden Durchlaufrichtungen wieder zum untersuchten Element i zurückkommt. Denn dann ist entweder die Bedingung in Zeile 6 oder jene in Zeile 11 erfüllt und man kann sofort umspeichern.

4.4 Modifikation mit zufälligen Hash-Funktionen

Bisher war es immer ein wesentlicher Bestandteil aller betrachteten Algorithmen, dass man den Zyklenführer als das kleinste Index-Element eines Zyklus bestimmt. Genau so gut könnte man etwa auch das größte Element eines Zyklus auszeichnen und dort die Rotation der Daten

starten. Geht man noch einen Schritt weiter, erkennt man, dass man auch auf die Indizes eines Zyklus eine Hash-Funktion anwenden kann und dann jenes Element auszeichnet, das den kleinsten Hash-Wert aufweist. Bei großen Datenfeldern kann dies durch eine geeignet gewählte Hash-Funktion durchaus zu kleineren, weniger den Speicher belastenden Hash-Werten führen. Klarerweise muss dann im Algorithmus eine Routine eingebaut werden, die überprüft, ob Kollisionen innerhalb eines Zyklus auftreten.
Eine weitere Idee ist es nun, dass man statt einer, gleich fünf verschiedene und vor allem voneinander unabhängige Hash-Funktionen verwendet und für jedes Element per Zufall die konkret zu verwendende Hash-Funktion auswählt. Man erhält auf diese Weise einen zufällig operierenden Algorithmus, der $O(\log_2 n)$ Hilfsspeicher benötigt und eine Laufzeit von $O(n \log_2 n)$ für alle Permutationen aufweist (siehe [1]).

Beispiel 4.4.1 *Es sei ein Datenfeld der Größe $n = 1000$ gegeben. Weiters sei eine Permutation π gegeben, die den Zyklus (327 115 891 733 416) beinhaltet. In diesem Zyklus wollen wir nun ein Element auszeichnen, das den kleinsten Hash-Wert zurückliefert. Dazu stehen uns die fünf unabhängigen Hash-Funktionen $h_1(x) = x \mod 500$, $h_2(x) = x \mod 600$, $h_3(x) = x \mod 700$, $h_4(x) = x \mod 800$ und $h_5(x) = x \mod 900$ zur Verfügung. Jetzt soll für jeden Eintrag im Zyklus zufällig eine Hash-Funktion bestimmt und der Hash-Wert ausgerechnet werden.*

betrachteter Zyklus:	(327	115	891	733	416)
zufällig gewählte Hash-Funktion:	h_3	h_5	h_1	h_3	h_4
Hash-Wert:	(327	115	391	33	416)

Glücklicherweise kommen hier keine zwei gleich großen Hash-Werte vor, das heißt, es entfällt eine Kollisionsbehandlung. Aufgrund des kleinsten Hash-Wertes wird das Element 733 in diesem Zyklus ausgezeichnet.

4.5 Neuer Ansatz mit einer Hierarchie der lokalen Minima

Das beste Resultat zur worst-case Verbesserung wurde bis jetzt bei der Variante mit der zusätzlich gegebenen inversen Permutation π^{-1} erzielt. Leider ist es aber in der Praxis sehr unwahrscheinlich, dass sowohl π, als auch π^{-1} gegeben sind. Eine Berechnung von π^{-1} wäre sehr aufwendig und würde sich kontraproduktiv auf die Laufzeit auswirken. In diesem Abschnitt wird ein Verfahren entwickelt, das ähnlich effizient den worst-case behandelt: Die maximale Laufzeit soll wieder $O(n \log_2 n)$ betragen und der Hilfsspeicher darf $O(\log_2^2 n)$ Bits groß sein.

Definition 4.5.1 *Sei eine Permutation π der Länge n gegeben. Wir definieren E_r als die Menge der lokalen Minima der Ordnung $r \geq 1$ induktiv, nämlich:*

$$E_r := \{i \in E_{r-1} \mid \pi_{r-1}^{-1}(i) > i < \pi_{r-1}(i)\} \quad \textit{mit}$$
$$E_1 := \{1, 2, ..., n\} \quad \textit{und}$$
$$\pi_r : E_r \to E_r,$$
$$\pi_r(i) := \pi_{r-1}^m(i) \quad \textit{für} \quad r > 1,\ i \in E_r \quad \textit{mit} \quad \pi_1 := \pi, \quad m = \min_{m \in \mathbb{N}} \{m > 0 \mid \pi_{r-1}^m(i) \in E_r\}.$$

π_r ist eine Permutation, die jedem Element von E_r "das nächste" Element in E_r zuordnet. Dabei bezieht sich "das nächste" auf die ursprüngliche Reihenfolge der Elemente in den Zyklen der Permutation π. Man erkennt aus der Definition nämlich sofort, dass $\pi_r(i) = \pi^M(i)$ für $r \geq 1,\ i \in E_r$ mit $M = \min\{M > 0 \mid \pi^M(i) \in E_r\}$ gelten muss. Ein Beispiel, übernommen von [1, Seite 270], soll den Sachverhalt illustrieren.

Beispiel 4.5.2 *Die Permutation $\pi = (1\ 5\ 3\ 6\ 10\ 4\ 2\ 9\ 8\ 7\ 11)$ sei in Zyklendarstellung gegeben. In Abbildung 4.1 sind die Elemente ihrer Größe und Reihenfolge entsprechend dargestellt, sodass man leicht die lokalen Minima, also die Menge E_r, und die Permutationen π_r erkennen kann.*

$$
\begin{array}{ll}
E_1 = \{1,2,3,4,5,6,7,8,9,10,11\}, & \pi_1 = (1\ 5\ 3\ 6\ 10\ 4\ 2\ 9\ 8\ 7\ 11),\\
E_2 = \{1,2,3,7\}, & \pi_2 = (1\ 3\ 2\ 7),\\
E_3 = \{1,2\}, & \pi_3 = (1\ 2),\\
E_4 = \{1\}, & \pi_4 = (1).
\end{array}
$$

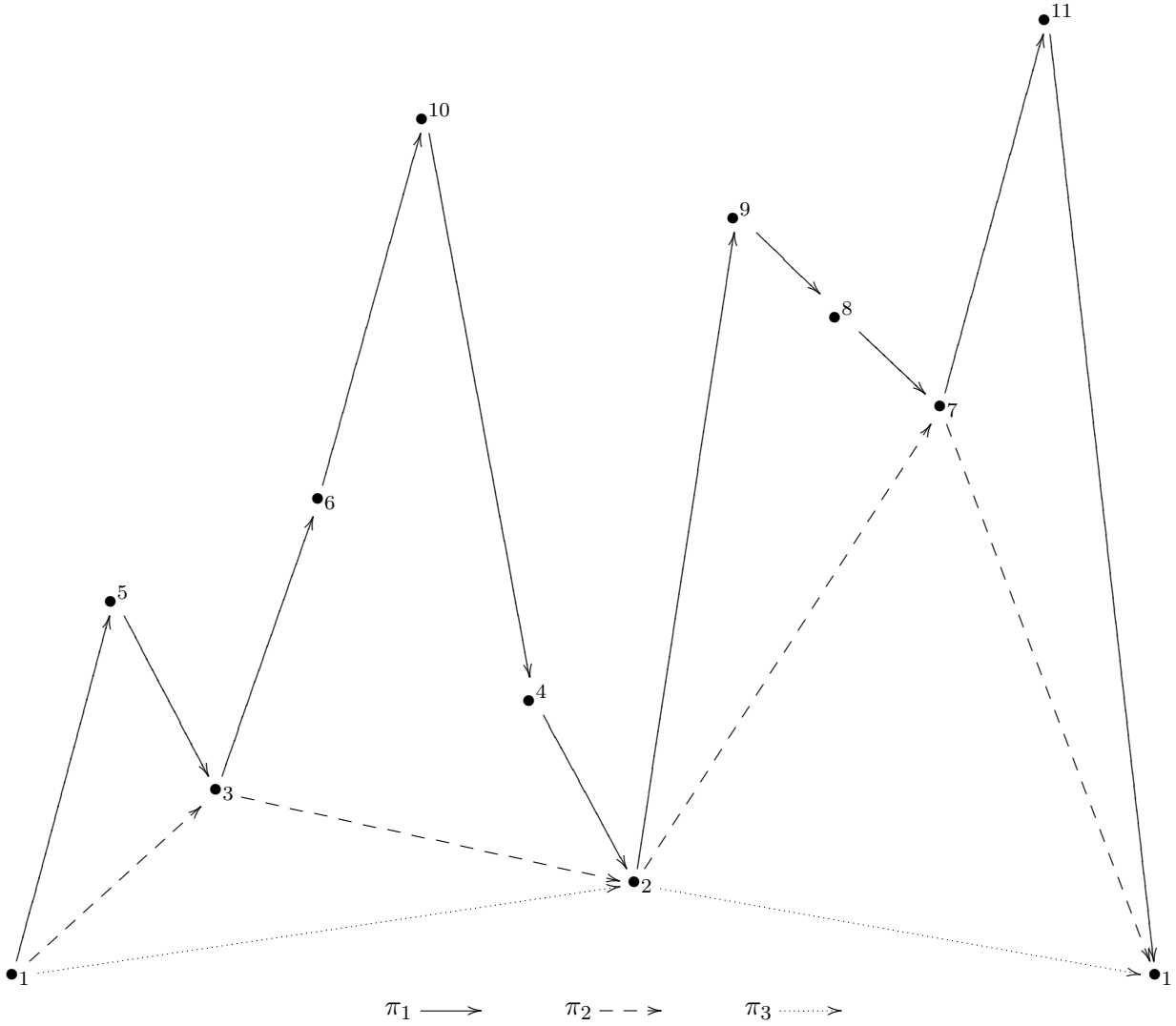


Abbildung 4.1: Ein Zyklus mit seinen lokalen Minima

Aus der Definition ist klar, dass $E_r \subseteq E_{r-1}$ gelten muss. Aus der Überlegung, dass nicht mehr als die Hälfte der Elemente lokale Minima sein können (da die Nachbarn von lokalen Minima

sicher keine lokalen Minima sind), ergibt sich sogar die echte Inklusion $E_r \subset E_{r-1}$ und man erhält $|E_r| < \frac{|E_{r-1}|}{2}$. Das kleinste Element eines gesamten Zyklus von π, sozusagen das absolute Minimum des Zyklus, ist auch ein lokales Minimum von allen π_r, die mehr als ein Element beinhalten und befindet sich somit in allen zugehörigen E_r. Daraus ergibt sich, dass dieses absolute Minimum eines Zyklus nichts anderes ist, als das eindeutige und einzige lokale Minimum maximaler Ordnung.
Nun geht es darum, ein Programm zu finden, das die Folge der $E_r, r > 1$, sowie die dazugehörigen π_r berechnet. Dazu gibt es prinzipiell zwei mögliche Ansätze. Der erste ist der, dass man eine "top-down"-Konstruktion verwendet, was in der Regel in einem rekursiven Algorithmus, also ein Algorithmus, der sich selbst aufruft, seine Implementierung findet. In der Computerwissenschaft ist auch die Rede von "divide and conquer" (D&C). Die Idee ist also, ein Problem in zwei oder mehrere Teile so lange zu zerlegen, bis die Teilprobleme leicht zu lösen sind und diese Resultate anschließend wieder zusammen zu fügen, um das ursprüngliche Problem lösen zu können. Der zweite Ansatz geht quasi den umgekehrten Weg, verwendet also ein "bottom-up"-Design. Hier beschäftigt man sich zunächst mit den kleinsten Einheiten und baut von diesen ausgehend eine komplexere Struktur Stück für Stück auf. Im Folgenden soll beiden Ansätzen nachgegangen werden und es sollen zwei Algorithmen vorgestellt werden, die obiger Problemstellung begegnen.

Satz 4.5.3 *Sei die Permutation π und ein Datenfeld der Größe n gegeben. Im worst-case lassen sich die Daten in $O(n \log_2 n)$ Schritten umspeichern. Der Hilfsspeicher beträgt $O(\log_2^2 n)$ Bits.*

Beweis. Wir haben schon festgestellt, dass es bei der betrachteten Aufgabenstellung leicht ist, $\pi(i)$ zu bekommen, während es sehr aufwendig wäre, würde man $\pi^{-1}(i)$ berechnen wollen. Wenn man überprüfen möchte, ob i ein lokales Minimum ist, ist es daher viel schwieriger, wenn man bei i startet, als bei dem Vorgänger von i. Im Beispiel 4.5.2 erkennt man sehr gut, dass wenn man von 5 startet, $5 > \pi(5) = 3 < \pi(\pi(5)) = 6$ gilt und man bei 3 ziemlich leicht ein lokales Minimum in π_1 gefunden hat. Von 3 kann man dann auf dieselbe Weise leicht erkennen, dass 2, der Nachfolger in π_2, ein lokales Minimum in π_2 sein muss.
Diese Vorgangsweise motiviert den ersten Algorithmus mit der Nummer 4.5.4. Für jeden Zyklus der Permutation π wird jenes eindeutige Element gesucht und ausgezeichnet, sodass $\pi_{s-1}...\pi_1(i) \in E_s$ gilt, wobei s die maximale Ordnung sein soll. Dieses Element ist nicht der Zyklenführer laut unserer ursrünglichen Definition – es wird also nicht das kleinste Element im Zyklus ausgezeichnet. Um noch einmal Beispiel 4.5.2 zu bemühen: Das gesuchte Element i ist dort die Fünf, da

$$\pi_3(\overbrace{\pi_2(\underbrace{\pi(5)}_{=3})}^{=2}) = 1 \in E_4.$$

Daraus folgt unmittelbar, dass man Elemente, die sicher nicht das lokale Minimum größter Ordnung sind, sofort verwerfen kann. In [1] wird nun auf ein Ergebnis von Dolev, Klawe und Rodeth [8] sowie jenes von Peterson [9] verwiesen, die für die Ermittlung dieses ausgezeichneten Elements gezeigt haben, dass es in einem gerichteten Ring $O(n \log_2 n)$ Überprüfungen benötigt. Das könnte für die Beweissetzung verwendet werden – hier soll allerdings gleich ein Algorithmus angegeben werden und anschließend eine Laufzeit- und Speicheranalyse Satz 4.5.3 belegen.

Zuerst soll nun der "top-down" Algorithmus vorgestellt und im Anschluss untersucht und analysiert werden.

Algorithmus 4.5.4 (Top-Down)

```
 1  procedure NEXT(r)
 2  if r = 1 then
 3     elbow[0] := π(elbow[1])
 4  else
 5     while elbow[r − 1] < elbow[r − 2] do
 6        elbow[r − 1] := elbow[r − 2]
 7        NEXT(r − 1)
 8     end while
 9     while elbow[r − 1] > elbow[r − 2] do
10        elbow[r − 1] := elbow[r − 2]
11        NEXT(r − 1)
12     end while
13  end if
14  return
15
16  for i := 1 to n do
17     elbow[0] := i
18     elbow[1] := i
19     for r := 1 to ⌈log2 n⌉ do
20        NEXT(r)
21        if elbow[r] > elbow[r − 1] then
22           elbow[r] := elbow[r − 1]
23           NEXT(r)
24           if elbow[r] > elbow[r − 1] then
25              exit
26           end if
27           elbow[r + 1] := elbow[r]
28        else
29           if elbow[r] = elbow[r − 1] then
30              ROTATE(i)
31           end if
32           exit
33        end if
34     end for
35  end for
```

Die Prozedur NEXT(r) berechnet nachfolgende Elemente entlang der Zyklen der Permutation π_r. Dabei kommt der schon vorhin erwähnte Aspekt der "top-down"-Algorithmen zum Tragen, nämlich die rekursive Behandlung. Um also das Folgeelement in π_r zu berechnen, werden die Elemente von π_{r-1} betrachtet. Diese bekommt man wiederum über die Elemente von π_{r-2}, usw. Ist das Problem nun leicht genug, also die maximale Rekursionstiefe erreicht, hat r den Wert eins und es wird in Zeile 2 das Folgeelement der ursprünglichen Permutation $\pi = \pi_1$ generiert und gespeichert. Bevor nun die Prozedur im Detail analysiert wird, soll das Hauptprogramm (Zeile 16 bis 35) erklärt werden.
Für alle i von 1 bis n wird der Reihe nach überprüft, ob $\pi_r\pi_{r-1}...\pi_1(i) \in E_{r+1}$ gilt. Es wird die Ordnung r beginnend mit $r = 1$ sukzessive gesteigert, bis entweder das Element verworfen werden kann, oder man die maximale Ordnung erreicht hat und so fündig geworden ist. Im Konkreten wird also zuerst beim ersten Durchlauf der Schleife in Zeile 19 überprüft, ob $\pi_1(i) \in E_2$, indem $\pi_1(i)$ mit i und $\pi_1\pi_1(i)$ verglichen wird. Grundvoraussetzung für Nicht-Fixpunkte

ist die Erfüllung der Bedingung in Zeile 21 – wenn $\pi_1(i)$ ein Minimum sein soll, dann muss es zuerst mit dem Wert runter und anschließend wieder hoch gehen. Die nächste Forderung ist also das Nicht-Erfüllen der Abfrage in Zeile 24 – sollte eine der beiden Forderungen (in einem nichttrivialen Zyklus) nicht erfüllt sein, kann man das Element bereits verwerfen. Fallen beide aber positiv aus, dann wird im zweiten Durchlauf der Schleife in Zeile 19 ($r = 2$) überprüft, ob $\pi_2\pi_1(i) \in E_3$, das heißt ob $\pi_1(i) < \pi_2\pi_1(i) < \pi_2\pi_2\pi_1(i)$ gilt. Das geht so lange so weiter, bis ein r gefunden wurde, sodass $\pi_{r-1}...\pi_1(i) \in E_r$ und $\pi_r\pi_{r-1}...\pi_1(i) \notin E_r$. Jetzt gibt es drei Fälle, die das Programm beenden. Entweder ist $\pi_{r-1}...\pi_1(i) = \pi_r\pi_{r-1}...\pi_1(i)$, also die Bedingung in Zeile 29 erfüllt; dann besteht die Permutation π_r nur aus einem einzigen Element, da klarerweise aus $\pi_r(x) = x$ sofort $\pi_r = (x)$ folgt. Man hat in diesem Fall das Minimum höchster Ordnung, nämlich $\pi_{r-1}...\pi_1(i)$, gefunden und kann in Zeile 30 die Umspeicherung mit dem ausgezeichneten Element i starten. Auch triviale Zyklen, also Fixpunkte der Permutation, fallen unter diesen Punkt. Die anderen beiden Fälle sind $\pi_{r-1}...\pi_1(i) < \pi_r\pi_{r-1}...\pi_1(i)$ oder $\pi_r\pi_{r-1}...\pi_1(i) > \pi_r\pi_r\pi_{r-1}...\pi_1(i)$. In beiden Fällen kann $\pi_r\pi_{r-1}...\pi_1(i)$ kein Minimum der Ordnung $r + 1$ mehr sein und wird verworfen.
Um der Forderung von lediglich $O(\log_2^2 n)$ Bits Hilfsspeicher gerecht werden zu können, kann nicht jede Permutation π_r komplett gespeichert werden. Es wird nun gezeigt, dass es reicht, wenn nur das jeweils letzte gefundene Minimum der Ordnung r für jede Permutation π_r gespeichert wird. Algorithmus 4.5.4 speichert diese Information im Vektor *elbow*. Am Ende der inneren Schleife gilt stets folgende Invariante: $elbow[r] = \pi_{r-1}...\pi_2\pi_1(i) \in E_r$ für $r > 1$. Unmittelbar vor einem Aufruf der Prozedur $NEXT(r)$ erfüllt der Inhalt von *elbow*

$$elbow[r] = elbow[r-1] \overset{\pi_{r-1}}{\to} ... \overset{\pi_1}{\to} elbow[0] \tag{4.2}$$

und unmittelbar danach

$$elbow[r] \overset{\pi_r}{\to} elbow[r-1] \overset{\pi_{r-1}}{\to} ... \overset{\pi_1}{\to} elbow[0]. \tag{4.3}$$

Dass $elbow[r] = elbow[r-1]$ vor dem Aufruf in Zeile 20 gelten muss ist klar: entweder es wurde vor der inneren Schleife in den Zeilen 17 und 18 so festgelegt, oder Zeile 27 wurde ausgeführt. Bei dem Aufruf der Prozedur NEXT(r) in Zeile 23 sorgt die Zeile unmittelbar darüber für diese Ausgangslage. Dort, in Zeile 22, findet auch die Verwerfung von Information statt: Das neu berechnete Element $elbow[r-1]$ ersetzt $elbow[r]$. Die Prozedur NEXT(r) berechnet also $\pi_r(elbow[r])$ und speichert das Resultat in $elbow[r-1]$. Dabei werden auch alle anderen Einträge von *elbow* entsprechend geändert - einzig der Eintrag von $elbow[r]$ bleibt wie er ist.
Aufgrund der Invarianten $elbow[r] \in E_r$ folgt unmittelbar $elbow[r] < \pi_{r-1}(elbow[r])$, da ja der Zyklus von π_{r-1} alle Elemente von E_r als lokale Minima beinhaltet. Nachdem $\pi_r(elbow[r-2]) = \pi_{r-1}(elbow[r])$ gilt, hat man also $elbow[r-1] < elbow[r-2]$. Die erste while-Schleife in Zeile 5 berechnet jetzt für $elbow[r-1]$ solange das Folgeelement bezüglich π_{r-1}, solange π_{r-1} strikt steigend ist. Analog wird rekursiv auch für $elbow[r-1]$ bis $elbow[0]$ vorgegangen. Anschließend wird in der zweiten while-Schleife in Zeile 9 solange das Folgeelement von $elbow[r-1]$ bezüglich π_{r-1} berechnet, solange π_{r-1} strikt fallend ist. Rekursiv passiert wieder das gleiche für $elbow[r-1]$ bis $elbow[0]$. Am Ende der zweiten while-Schleife muss die Abbruchbedingung $elbow[r-1] < elbow[r-2] = \pi_{r-1}(elbow[r-1])$ gelten. Die Werte waren also in der zweiten while-Schleife fallend und jetzt gibt es wieder eine Steigerung. Daher ist $elbow[r-1]$ das nächste lokale Minimum von π_{r-1}. Also hat man $\pi_r(elbow[r])$ in $elbow[r-1]$ erfolgreich berechnet.
Nun zur Laufzeitanalyse. Sei $i' = \pi_r\pi_{r-1}...\pi_1(i) \in E_r \setminus E_{r+1}$. Vor dem Aufruf der NEXT-

Prozedur in Zeile 20, also am Beginn der r-ten Iteration der inneren for-Schleife des Hauptprogrammes, gilt $elbow[r] = elbow[r-1] = \pi_{r-1}...\pi_1(i) = \pi_r^{-1}(i') \in E_r$. Sobald in der Programmabfolge klar wird, dass $i' \notin E_{r+1}$ gilt, ist entweder $elbow[r] = \pi_r^{-1}(i')$ und $elbow[r-1] = i'$ bei Nichterfüllung der Bedingung in Zeile 21, oder $elbow[r] = i'$ und $elbow[r-1] = \pi_r(i')$, wenn das Abbruchkriterium in Zeile 24 erfüllt ist. Der Eintrag in $elbow[0]$ ist in den beiden Fällen unterschiedlich, hängt aber wegen (4.3) immer von $elbow[r-1]$ in der Weise ab: $elbow[0] = \pi_1...\pi_{r-1}(elbow[r-1])$. Man kann leicht mit einer Induktion nach r zeigen, dass wenn $j \in E_r$ und $M \in \mathbb{N}$ die kleinste mögliche Zahl ist, sodass $\pi_1...\pi_{r-1}(j) = \pi^M(j)$ gilt, dann hat man $\pi^L(j) \notin E_r$ für $1 \leq L \leq M$ und $r > 1$. Daraus ergibt sich, dass wenn man bei i' startet und sich entlang π bewegt, das Element $\pi_1...\pi_{r-1}\pi_r(i')$ vor dem Element $\pi_r^2(i')$ erreicht wird. Analog wird von i' ausgehend rückwärts entlang π das Element $i = \pi_1^{-1}...\pi_{r-1}^{-1}(i')$ vor π_r^{-2} erreicht. Testet man also für ein i, ob es das gesuchte auszuzeichnende Element ist, muss man sich entlang π in einem Subsegment der Region zwischen $\pi_r^{-2}(i')$ und $\pi_r^2(i')$ bewegen.
Aufgrund dieser Überlegung kann man jetzt für alle $i' \in E_r \setminus E_{r+1}$ feststellen, dass die Permutation π höchstens $4n$-mal berechnet werden muss; für jedes Element nämlich maximal 4 Mal, da man ja für jedes Element entlang π_r maximal zweimal vor und zweimal zurückgehen muss und somit auch für alle Zwischenelemente x von π höchstens 4 Mal $\pi(x)$ durchgeführt werden muss. Die Abschätzung ist insofern ein wenig großzügig, weil man davon ausgeht, dass man prinzipiell für alle Elemente in E_r die zweimalige Überprüfung durchführen könnte. In Wirklichkeit werden aber ja nur jene betrachtet, die nicht in E_{r+1} liegen. Nachdem E_r für alle $r > \lceil \log_2 n \rceil$ leer ist, muss der Algorithmus insgesamt $O(n \log_2 n)$ Schritte entlang π berechnen.
Jeder Aufruf von NEXT(1) berechnet in Zeile 3 einen Schritt entlang π. Bei einem Aufruf NEXT(r) mit $r > 1$ wird die Prozedur zumindest zweimal rekursiv mit NEXT($r-1$) aufgerufen. Die exakte Arbeit für so einen Aufruf ist also proportional zur Anzahl der rekursiven Aufrufe (exklusive der Arbeit für die rekursiven Aufrufe). Für jedes i wird in den Zeilen 19 bis 34, also der inneren for-Schleife des Hauptprogrammteils, eine konstante Anzahl an Vergleichen gemacht. Somit ist die Arbeit hierfür in $O(1)$, wenn man von den aufgerufenen Subroutinen absieht. Alle Iterationen dieser inneren Schleife ziehen zwei Aufrufe der Prozedur NEXT mit sich, bis auf die letzte, die möglicherweise lediglich einen Aufruf durchführt. Die Kosten der Prozedur ROTATE in Zeile 30 wurden schon besprochen - sie betragen für alle Zyklen der Permutation $O(n)$. Damit ist die gesamte Arbeit des Algorithmus proportional zur Anzahl der Schritte, die entlang der Permutation π durchgeführt werden müssen plus $O(n)$, also $O(n \log_2 n) + O(n) = O(n \log_2 n)$.
Nun wollen wir den worst-case für diesen Algorithmus betrachten. Dieser ist gegeben, wenn π aus einem einzigen Zyklus der Länge n besteht, wobei die Indizes die Elemente $\{0, ..., n-1\}$ sind. Zusätzlich müssen diese in der Zyklendarstellung bezüglich ihrer $(\log_2 n)$-Bit Binärdarstellung rückwärts lexikographisch geordnet sein. Ein Beispiel soll dies veranschaulichen.

Beispiel 4.5.5 *Die Permutation π habe die Länge 8. Wir wollen, um den worst-case des Algorithmus 4.5.4 erzeugen zu können, zuerst die Zahlen in die rückwärts lexikographisch geordnete Binärdarstellung bringen. In der nachfolgenden rechten Spalte sind die Elemente in herkömmlicher lexikographischer Binärdarstellung angeordnet, während in der zweiten Spalte die Reihenfolge bezüglich der rückwärts lexikographisch geordneten Binärdarstellung gewählt wurde:*

$0 =$	$\overleftarrow{000}$,	$0 =$	$\overrightarrow{000}$,
$1 =$	001,	$4 =$	100,
$2 =$	010,	$2 =$	010,
$3 =$	011,	$6 =$	110,
$4 =$	100,	$1 =$	001,
$5 =$	101,	$5 =$	101,
$6 =$	110,	$3 =$	011,
$7 =$	111,	$7 =$	111.

Die Permutation sieht in Zyklenschreibweise also so aus: $\pi = (0\ 4\ 2\ 6\ 1\ 5\ 3\ 7)$. *Damit ergibt sich für* E_r *und* π_r*:*

$$\begin{array}{ll} E_1 = \{0,1,2,3,4,5,6,7\}, & \pi_1 = (0\ 4\ 2\ 6\ 1\ 5\ 3\ 7), \\ E_2 = \{0,1,2,3\}, & \pi_2 = (0\ 2\ 1\ 3), \\ E_3 = \{0,1\}, & \pi_3 = (0\ 1), \\ E_4 = \{0\}, & \pi_4 = (0). \end{array}$$

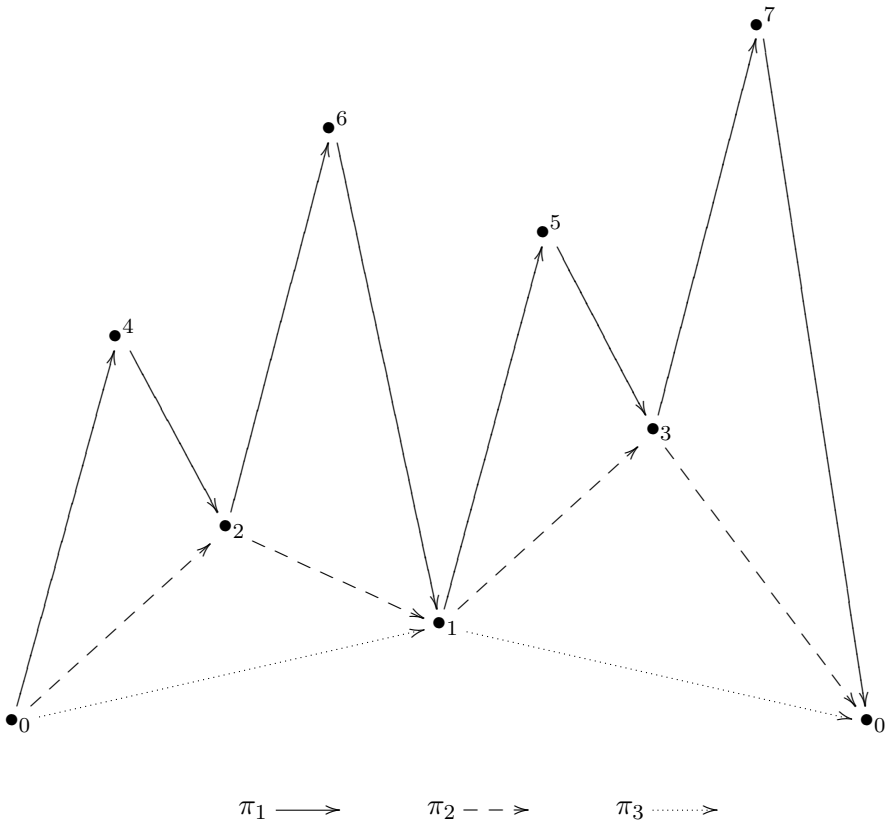


Abbildung 4.2: Zyklus des worst-case Szenarios mit seinen lokalen Minima

Man sieht also, dass es Fälle gibt, wo der Algorithmus $\Omega(n \log_2 n)$ Schritte benötigt. Somit ist die Laufzeit des Algorithmus im worst-case Fall $\Theta(n \log_2 n)$.

Der Algorithmus verwendet $\Theta(\log_2 n)$ Variablen (das entspricht der Größe des Vektors *elbow*), wovon jede in der Lage ist, ein Element von $\{0, ..., n\}$ aufzunehmen, also $\log_2 n$ groß sein muss. Es wird also im Algorithmus 4.5.4 zusätzlicher Hilfsspeicher benötigt, der $\Theta(\log_2^2 n)$ Bits groß ist. □

Als nächstes wird für den besprochenen Algorithmus eine "bottom-up"-Lösung vorgestellt. Ähnlich wie bei Algorithmus 4.5.4 sollen nach und nach die lokalen Minima berechnet und in den Vektor *elbow* gespeichert werden. Weiters wird ein Vektor *state* eingeführt, der die Zustände in einem Zyklus speichert und zwar für jede Ordnung den gerade aktuellen Zustand. Dadurch kann also auf eine Rekursion verzichtet werden und ein Algorithmus gefunden werden, der wieder eine Laufzeit von $O(n \log_2 n)$ hat und $O(\log_2^2 n)$ Bits zusätzlichen Hilfsspeicher benötigt.

Algorithmus 4.5.6 (Bottom-Up)

```
 1  for i := 1 to n do
 2    elbow[1] := i
 3    state[1] := GOT_ONE
 4    repeat
 5      r := 1
 6      elbow[0] := π(elbow[1])
 7      while state[r] = DOWN and elbow[r] < elbow[r − 1] do
 8        state[r] = UP
 9        elbow[r] := elbow[r − 1]
10        r := r + 1
11      end while
12      select case state[r]
13
14        case GOT_ONE
15          if elbow[r] > elbow[r − 1] then
16            state[r] := GOT_TWO
17            elbow[r] := elbow[r − 1]
18          else
19            if elbow[r] = elbow[r − 1] then
20              ROTATE(i)
21            end if
22          end if
23
24        case GOT_TWO
25          if elbow[r] > elbow[r − 1] then
26            exit
27          end if
28          state[r + 1] := GOT_ONE
29          elbow[r + 1] := elbow[r]
30          state[r] := UP
31          elbow[r] := elbow[r − 1]
32
33        case UP
34          if elbow[r] > elbow[r − 1] then
35            state[r] := DOWN
36          end if
37          elbow[r] := elbow[r − 1]
38
39        case DOWN
40          elbow[r] := elbow[r − 1]
41      end select
42    end repeat
43  end for
```

Der Vektoreintrag $elbow[0]$ bewegt sich entlang des Zyklus von π, der das zu untersuchende i beinhaltet. Wie schon bei Algorithmus 4.5.4 speichert $elbow[r]$ für $r > 1$ das zuletzt gefundene Element von E_r. Der Vektor *state* speichert zusätzlich, in welchem Teilbereich man sich im Zyklus befindet.

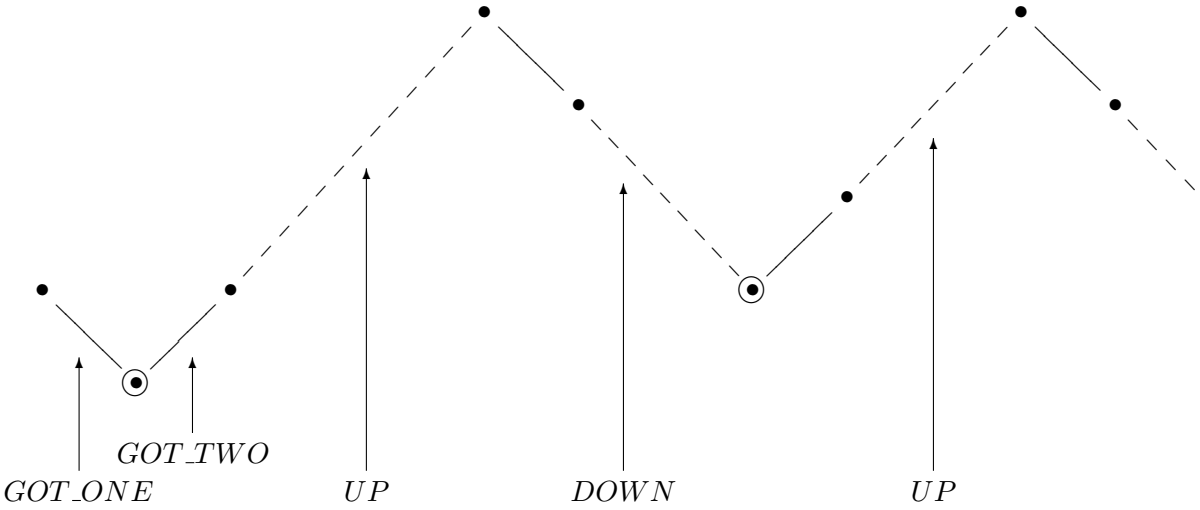


Abbildung 4.3: Zustände im Bottom-Up Algorithmus

Wenn das erste Element von E_r ausgemacht wurde, wird der zugehörige Eintrag im Vektor *state* auf $state[r] = GOT\_ONE$ gesetzt. Das geschieht am Beginn der Untersuchung für das jeweilige i, also in Zeile 3, und in Zeile 28, wo das erste Minimum von E_{r-1} erkannt wurde. Weiters gilt wieder $elbow[r] = \pi_{r-1}...\pi_1(i)$. Kommt man nun zum zweiten Elemtent von E_r entlang π_r, so wird in Zeile 15 überprüft, ob das zweite Element kleiner als das erste ist. Denn nur so kann potentiell bei einer nichttrivialen Permutation π_r ein Minimum von E_r vorliegen. Sollte dies der Fall sein, erhält $state[r]$ den neuen Zustand GOT_TWO (Zeile 16) und es wird im Folgenden in Zeile 25 überprüft, ob $\pi_r\pi_{r-1}...\pi_1(i) > \pi_{r-1}...\pi_1(i)$ gilt und somit $\pi_{r-1}...\pi_1(i)$ ein lokales Minimum von π_{r-1} ist, also $\pi_{r-1}...\pi_1(i) \in E_r$ gilt. Sollte auch diese Überprüfung positiv ausfallen, kann man eine Ebene höher gehen und im Weiteren untersuchen, ob $\pi_r\pi_{r-1}...\pi_1(i)$ ein lokales Minimum von π_r ist und damit in E_{r+1} liegt. Diese Überprüfungen gehen nun so lange weiter, bis entweder feststeht, dass die nächste Ebene kein Minimum mehr darstellt und das betrachtete i verworfen werden kann, oder bis die Überprüfung in Zeile 19 positiv ausfällt. Dort wird nämlich untersucht, ob E_r nur ein Element beinhaltet und somit π_r eine triviale Permutation ist. Sollte dies der Fall sein, hat man das gesuchte i gefunden und kann gleich in Zeile 20 die Umspeicher-Routine aufrufen.
Bei $state[r] = UP$ ist $elbow[r]$ stets größer als $\pi_r^{-1}(elbow[r])$, also größer als der unmittelbare Vorgänger in π_r. Analoges liegt bei $state[r] = DOWN$ vor, dass nämlich $elbow[r] < \pi_r^{-1}(elbow[r])$ gilt, die Werte also fallend sind. Das "Umschalten" zwischen diesen beiden Zuständen wird in der while-Schleife (Zeilen 7 bis 11) durchgeführt. Dabei ist zu beachten, dass dies für alle Ebenen geschieht und der Befehl in Zeile 10 die Variable r auf die aktuell zu betrachtende Ebene setzt, für die dann die Untersuchungen fortgesetzt werden. Somit ist sichergestellt, dass alle Minima aller Ebenen gefunden werden können.
Die Graphik 4.3 veranschaulicht die wechselnden Einträge von $state[r]$ beim Durchlaufen von π_r, wobei r fest bleibt. Dies geschieht für alle r auf dieselbe Weise. Es wird also auf jeder Ebene des Algorithmus nach diesem Schema verfahren.

4.6 "In order"- und "Out of order"-Permutation

Sehr oft erfüllen die zu permutierenden Einträge des Datenfelds eine Totalordnung. Daher kann man eine Permutation σ definieren, die auf dem bezüglich der Permutation π umgespeicherten Datenfeld A folgende Bedingung erfüllt:

$$A[\sigma^{-1}(1)] \leq A[\sigma^{-1}(2)] \leq ... \leq A[\sigma^{-1}(n)]. \tag{4.4}$$

Der Eintrag $A[i]$ hat also den $\sigma(i)$-t kleinsten Eintrag. Daraus ergibt sich auch, dass wenn $\sigma(i) > \sigma(\pi(i))$ gilt, $A[i] > A[\pi(i)]$ gelten muss. Zwei Beispiele verdeutlichen den Zusammenhang zwischen den Permutationen π und σ: Wenn π so gewählt wird, dass es die Einträge des Datenfelds A in eine aufsteigend geordnete Reihenfolge umspeichert, so ist σ nichts anderes als die Identität. War A vor der Permutation π bereits vom kleinsten bis zum größten Element sortiert, dann gilt $\sigma = \pi^{-1}$.
Hat man eine derartige Ordnungspermutation σ zur Hilfe, kann der bisher kürzeste Permutationsalgorithmus angegeben werden:

Algorithmus 4.6.1 (In-Order mit Elemente von A verschieden)

```
1  for i := 1 to n do
2    if σ(i) > σ(π(i)) and A[i] < A[π(i)] then
3      ROTATE(i)
4    end if
5  end for
```

Die einzige Einschränkung dieses Algorithmus ist jene, dass alle Einträge von A verschieden sein müssen, also für alle $i \neq j$ auch $A[i] \neq A[j]$ gilt. Um noch einmal auf eines der beiden Beispiele für die Permutation σ zurück zu kommen: ollte σ die Identität sein, also bereits π die Elemente in die geforderte Ordnung bringen, dann sucht der Algorithmus nach Platznummern, wo die Permutation π ein Element auf den Platz mit einer niedrigeren Platznummer tauschen möchte und dabei das Element selbst kleiner als jenes ist, das sich im Moment dort befindet.
Nun soll die Korrektheit dieses vorgestellten Algorithmus untersucht werden. Nach der Rotation des Zyklus, der ein beliebiges j beinhaltet, impliziert $\sigma(j) > \sigma(\pi(j))$, dass $A[j] \geq A[\pi(j)]$ gilt. Somit ist die Bedingung in Zeile 2 für kein Element des Zyklus erfüllt. Daher wird jeder nichttriviale Zyklus höchstens einmal rotiert.
Damit jeder Zyklus zumindest einmal rotiert wird, muss es eine Position i im Zyklus geben, für die die Bedingung in Zeile 2 erfüllt ist, also $\sigma(i) > \sigma(\pi(i))$ und $A[i] < A[\pi(i)]$. Sicherlich gibt es eine Position i im Zyklus, für die $\sigma(i) > \sigma(\pi(i)) < \sigma(\pi^2(i))$ gilt. Zum Beispiel ist dies der Fall, wenn $\pi(i)$ jene Position ist, die den kleinsten zugehörigen Wert im Datenfeld A hat. Es wird nun gezeigt, dass das auszuzeichnende, gesucht Element, das dann der ROTATE-Prozedur übergeben werden sollte, das erste ist, das gefunden wird und diese Bedingung erfüllt. Zunächst wird also auf alle Fälle so ein i gefunden und es soll jetzt die ROTATE-Prozedur mit Eingangsparameter i aufgerufen werden. Nachdem der Zyklus umgespeichert wurde, gilt $A[\pi(i)] \leq A[\pi^2(i)]$, wegen $\sigma(\pi(i)) < \sigma(\pi^2(i))$. Die Werte von A der Plätze $\pi(i)$ und $\pi^2(i)$ waren vor der Rotation auf den Plätzen i und $\pi(i)$. Somit galt vor dem Umspeichern $A[i] \leq A[\pi(i)]$. Aufgrund der geforderten Verschiedenheit der Elemente von A gilt $A[i] \neq A[\pi(i)]$, da $i \neq \pi(i)$. Somit folgt die zu zeigende Ungleichung $A[i] < A[\pi(i)]$, die jetzt auch, zusammen mit $\sigma(i) > \sigma(\pi(i))$, als Kriterium zum Auffinden des auszuzeichnenden Elements verwendet werden kann.

Im Folgenden soll Algorithmus 4.6.1 so adaptiert werden, dass es auch möglich ist, die Forderung, dass alle Elemente von A verschieden sein müssen, fallen zu lassen. Dazu soll zuerst analysiert werden, wo das Problem liegt. Denn Zyklen, die alle dieselben zugehörigen Werte im Datenfeld A beinhalten, müssen klarerweise nicht permutiert werden. Sollten allerdings mehrere Werte mehrmals vorkommen, kann es zu Problemen kommen. Um so einen Problemfall zu sehen, soll hier das Beispiel von [1, Seite 275] angeführt werden.

Beispiel 4.6.2 *Der Einfachheit halber soll die Permutation π die Elemente eines Datenfeldes A in eine geordnete Reihenfolge bringen. Die Ordnungspermutation σ ist also die Identität. Weiters beinhaltet A nur die Werte α und β, für die folgende strenge Totalordnung (siehe Definition 1.1.3 auf Seite 2) gilt: $\alpha < \beta$. Das Datenfeld A ist schon vorab geordnet, bis auf die zwei mittleren Elemente:*

$$A = [\alpha \ ... \ \alpha \ \beta \ \alpha \ \beta \ ... \ \beta].$$

Die Abbildung 4.4 zeigt nun, dass kein Element gefunden wird, das die geforderte Bedingung in Zeile 2 erfüllt, wodurch es nie zum Aufruf der ROTATE*-Routine kommt.*

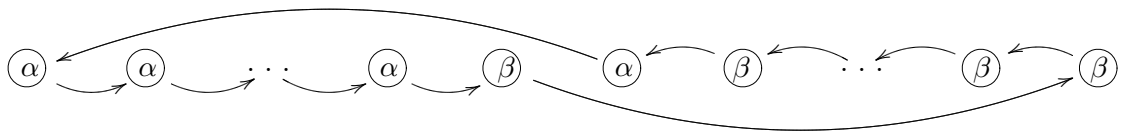


Abbildung 4.4: Gegenbeispiel zur Korrektheit von Algorithmus 4.6.1 bei nicht verschiedenen Einträgen

Das Problem hierbei ist, dass die beiden Nachbareinträge des Datenfelds A im Zyklus beliebig weit voneinander getrennt sein können. So kann ein rein lokaler Test hier nie fündig werden.

Definition 4.6.3 *Es sei ein Datenfeld A der Größe n gegeben. Darauf sei eine Permutation π gegeben und die Ordnungspermutation σ erfülle die Bedingung (4.4). Gilt für $j \in \{1, ..., n\}$: $A[\sigma^{-1}(j)] = ... = A[\sigma^{-1}(j+k)]$ und zusätzlich $A[\sigma^{-1}(j)] \neq A[\sigma^{-1}(j-1)]$, wenn $j > 1$ bzw. $A[\sigma^{-1}(j+k)] \neq A[\sigma^{-1}(j+k+1)]$, wenn $j+k < n$, dann bezeichnet die Folge $(j, j+1, ..., j+k)$ einen Lauf (englisch: "run") des Wertes $A[\sigma^{-1}(j)]$.*
Man hat also bei einem Lauf der Länge $k+1$ für $j \in \{2, ..., n-k-1\}$ diese Situation:

$$... \leq A[\sigma^{-1}(j-1)] < \underbrace{A[\sigma^{-1}(j)] = ... = A[\sigma^{-1}(j+k)]}_{\text{"run"}} < A[\sigma^{-1}(j+k+1)] \leq ...$$

Die Idee, wie man der Problematik mit gleichen Werten begegnet, ist nun, dass man einen Lauf als ein einziges Element betrachtet. Denn beinhaltet ein Zyklus einen oder mehrere Läufe, so ist es unerheblich, ob man alle Elemente durchexerziert, oder lediglich das erste Element eines Laufes betrachtet, die folgenden des selben Laufes überspringt und wieder mit dem ersten Element des nächsten Laufes fortsetzt. Nachdem man auf diese Weise lediglich verschiedene Werte betrachten muss, kann wieder die Überprüfung aus Algorithmus 4.6.1 verwendet werden. Diese Überlegungen motivieren den folgenden Algorithmus.

Algorithmus 4.6.4 (In-Order mit A beliebig)

```
 1  for i := 1 to n do
 2    if A[i] ≠ A[π(i)] then
 3      j := π(i)
 4      while A[j] = A[π(j)] do
 5        j := π(j)
 6      end while
 7      if σ(π(i)) > σ(π(j)) and A[π(i)] < A[π(j)] then
 8        ROTATE(i)
 9      end if
10    end if
11  end for
```

Mit diesem Algorithmus wird das vorige Gegenbeispiel korrekt abgehandelt.

Beispiel 4.6.5 (Fortsetzung von Beispiel 4.6.2) *Die Aufgabenstellung aus Beispiel 4.6.2 wird jetzt gelöst, indem der Algorithmus, aufgrund der Abfrage in Zeile 2 und der while-Schleife in Zeile 4, immer an das Ende eines jeden Laufes springt und somit nach ein paar Iterationen schließlich zu folgender Situation kommt:*

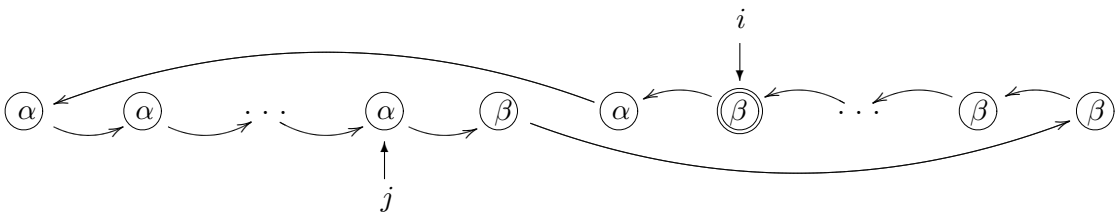


Abbildung 4.5: Korrekte Abarbeitung des vorigen Gegenbeispiels 4.6.2

Jetzt ist die Bedingung in Zeile 7 erfüllt und es kann der ROTATE*-Prozedur das markierte Element übergeben werden.*

Die Laufzeit des Algorithmus hängt sehr stark von der Auswertung der beiden Permutationen π und σ ab. Kann sowohl π als auch $\sigma\pi$ in konstanter Zeit berechnet werden, benötigt der Algorithmus lediglich lineare Laufzeit, also $O(n)$. Dies ist der Fall, wenn π die Einträge des Datenfelds A in eine Totalordnung umspeichert, oder wenn π auf eine schon existierende Ordnung angewandt wird. Lineare Laufzeit ist deshalb möglich, weil die innere while-Schleife (Zeilen 4 bis 6) für jeden Lauf in einem Zyklus höchstens einmal abgearbeitet wird.
Die Forderung von der konstanten Auswertung von π und σ kann noch abgeschwächt werden, wie der folgende Satz zeigt.

Satz 4.6.6 *Nachdem das Datenfeld A der Länge n bezüglich der Permutation π umgespeichert wurde, erfülle die Permutation σ*

$$A[\sigma^{-1}(1)] \leq A[\sigma^{-1}(2)] \leq ... \leq A[\sigma^{-1}(n)].$$

Können alle Elemente $\pi(1), ..., \pi(n), \sigma\pi(1), ..., \sigma\pi(n)$ in linearer Zeit berechnet werden, dann kann A, unter Verwendung von $O(\log_2 n)$ Bits zusätzlichen Speicher, entlang π im worst-case in $O(n)$ Schritten umgespeichert werden.

Beweis. Für jedes Element i wird $\pi(i)$ und $\sigma\pi(i)$ lediglich $O(1)$-mal berechnet. Daher wird die Laufzeit von der Auswertung genau dieser Funktionen dominiert. □

Als Anwendungsgebiet für "In order"-Permutationen wird in [1] die Konstruktion einer Datenstruktur angegeben, die implizit einen binären Suchbaum darstellt. Dafür sei ein Datenfeld der Länge n mit schon geordneten Einträgen gegeben. Dazu soll also der Median auf der ersten Position gespeichert werden, das erste und dritte Quartil auf den Stellen zwei und drei, und so fort. Die Kinder eines Elements sollen sich (analog zu einem heap) an den Positionen $2j$ und $2j + 1$ befinden. Es gelang bereits Bentley (siehe [10]) einen Algorithmus zu finden, der diesen Spezialfall in linearer Zeit lösen konnte.
Der Schlüssel zum Verständnis des Folgenden ist das Operieren von π auf einem Element i in der $\lceil \log_2(n) \rceil$-Bit-Binärdarstellung.

Lemma 4.6.7 *Sei $i = x10^j$ in der $\lceil \log_2(n) \rceil$-Bit-Binärdarstellung gegeben, wobei x einen beliebigen String der Länge $\lceil \log_2(n) \rceil - j - 1$ aus den Elementen Null und Eins darstellen soll. Will man mit der Permutation π ein bereits geordnetes Datenfeld A der Länge n so umspeichern, dass implizit ein binärer Suchbaum entsteht, dann gilt $\pi(i) = 0^j 1x$.*

Beweis. Es soll hier eine Induktion nach der Länge des Strings x gemacht werden. Diese Länge soll eine neue Variable erhalten, nämlich $k := \lceil \log_2(n) \rceil - j - 1$. Somit hat man $x = (x_1 x_2 ... x_k)$, mit $x_m \in \{0, 1\}$ für alle m von 1 bis k. Für den Induktionsanfang $k = 0$, also $x = \{\}$, nimmt $j = \lceil \log_2(n) \rceil - 1$ den maximal möglichen Wert an. Die folgenden Rechnungen werden in der Binärschreibweise angeschrieben. Es müsste laut Formel gelten:

$$\pi(10...0) = (0...01)$$

Das Element (10...0) sollte also an die erste Position permutiert werden und wäre somit das Wurzel-Element. Das muss auch so sein, denn der Wurzelknoten soll den gesamten Wertebereich in zwei gleich große Teile aufspalten. In der Baumstruktur sind alle Elemente links davon kleiner, während die rechten Kinder mitsamt allen nachfolgenden Kindeskindern größer sind.
Im Induktionsschritt kann nun angenommen werden, dass $\pi(x10^j) = 0^j 1x$ für alle x der Länge $k - 1$ gilt. Zu zeigen ist, dass die Gleichung ebenso für die Länge k richtig ist. Nachdem jede Permutation eine Bijektion ist, soll hier vom Bild-Element der Permutation π, nämlich von $i' = 0^j 1 x_1 x_2 ... x_k$ ausgegengen werden. Angenommen es gilt $x_k = 0$, dann soll $\frac{i'}{2}$ ermittelt werden. In der Binärdarstellung entspricht das einem Verschieben aller Eins-Elemente um eine Stelle nach rechts und man erhält

$$\tfrac{i'}{2} = 0^{j+1} 1 \underbrace{x_1 x_2 ... x_{k-1}}_{\text{Länge } k-1}.$$

Jetzt kann man die Induktionsannahme verwenden, um das Urbild von $\frac{i'}{2}$ zu erhalten:

$$\pi^{-1}\left(\tfrac{i'}{2}\right) = \pi^{-1}(0^{j+1} 1 x_1 x_2 ... x_{k-1}) = x_1 x_2 ... x_{k-1} 10^{j+1}.$$

Dieses Element muss ein Kind von $\pi^{-1}(i')$ sein, da ja zu jedem Element mit Index j ein Kind auf die Positionen $2j$ permutiert wird. Der Elternknoten von $x_1 x_2 ... x_{k-1} 10^{j+1}$ liegt in der Baumstruktur eine Ebene höher und ist somit $x_1 x_2 ... x_{k-1} 010^j$. Man hat also gezeigt, dass

$$\pi^{-1}\left(0^j 1 x_1 x_2 ... x_k\right) = x_1 x_2 ... x_{k-1} 010^j = x_1 x_2 ... x_{k-1} x_k 10^j$$

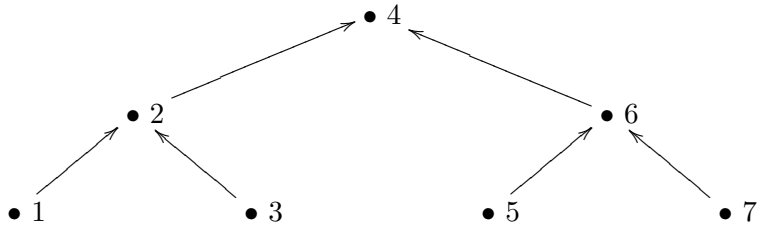


Abbildung 4.6: Ein voll belegter binärer Suchbaum

gilt. Den Fall $x_k = 1$ kann man auf den eben beschriebenen zurückführen, da man vor der Division durch 2 noch den Wert 1 abziehen kann, da sich ja auch auf der Position $2j + 1$ (nach der Permutation) ein Kind von dem Element mit Platznummer j befindet. □

Es soll nun abschließend ein Algorithmus gefunden werden, der die Daten so umspeichert, dass implizit ein binärer Suchbaum gegeben ist. Dabei soll der Speicherplatz effizient genutzt werden, denn während $A = (4\ 2\ 6\ 1\ 3\ 5\ 7)$ (siehe Abbildung 4.6) eine gute Speicherung der Daten ist, so geht bei $A = (8\ 4\ .\ 2\ 6\ .\ .\ 1\ 3\ 5\ 7\ .\ .\ .\ .)$ (siehe Abbildung 4.7) eine Menge an Speicherplatz ungenützt verloren.

Für jede mögliche Datenfeldgröße $n \in \mathbb{N}$ kann eine Darstellung gefunden werden, sodass $n = 2^h - 1 + r$ mit $1 \leq r < 2^h$ gilt. Es kann nun das geordnete Datenfeld A derart permutiert werden, dass implizit ein binärer Suchbaum gegeben ist, dessen Einträge voll besetzt sind mit Außnahme der letzten Ebene. Für diese letzte Ebene wird aber zusätzlich gefordert, dass sich die Einträge möglichst weit links befinden. Die erste Möglichkeit ist die, dass man zwei Permutationen durchführt, wobei die erste, π', lediglich die Einträge der letzten Ebene an die richtigen Stellen bringt und die Ordnung der anderen Elemente erhält. Anschließend muss die zweite Permutation die ersten $2^h - 1$ Einträge in die gewünschte Struktur umspeichern. Die erste Permutation π' sieht so aus:

$$\pi'(i) = \begin{cases} 2h + \frac{i-1}{2} & \text{wenn } i \text{ ungerade ist und } i < 2r \text{ gilt,} \\ \frac{i}{2} & \text{wenn } i \text{ gerade ist und } i < 2r \text{ gilt,} \\ i - r & \text{wenn } i > 2r \text{ gilt.} \end{cases}$$

Damit werden also die Werte $1, 3, 5, ..., 2r - 1$ an die richtige Stelle in der letzten Ebene gespeichert. Mit dieser Überlegung kann man sich jetzt gleich an einen Algorithmus heranwagen, der beide Permutationen kombiniert und somit in einem Durchlauf die gesuchte Permutation liefern kann.

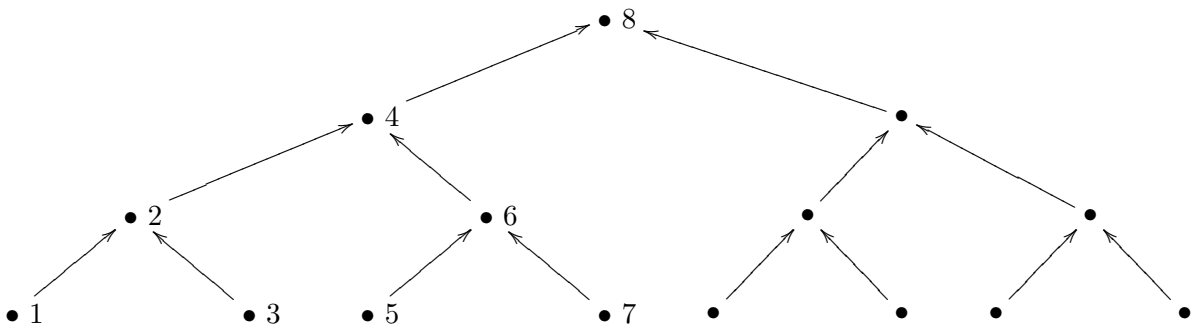


Abbildung 4.7: Ein schwach besetzter binärer Suchbaum

Algorithmus 4.6.8 (Berechnung von π für bin. Suchbaum-Struktur out-of-order)

```
1  for i := 1 to n do
2    if i/2 ≠ ⌊i/2⌋ and i < 2r then
3      π(i) := 2^h + (i−1)/2
4    else
5        π(i) := BINARY_TRANS(i)
6    end if
7  end for
8
9  procedure BINARY_TRANS(i)
10 if i/2 = ⌊i/2⌋ and i < 2r then
11   j := i/2
12 elseif i ≥ 2r then
13   j := i − r
14 end if
15 translate j → x10^j
16 translate j ← 0^j 1x
17 return j
```

In Zeile 2 werden alle ungeraden Elemente, die kleiner als $2r$ sind, herausgesucht. Für jedes solche Element i wird $\pi(i)$ in Zeile 3, wie beim ersten Ansatz mit π', berechnet. Andernfalls wird die Funktion BINARY_TRANS aufgerufen, die dann in Hinblick auf Lemma 4.6.7 und die noch freien Positionen den richtigen Permutationswert ermittelt. Dabei wandelt der Befehl **translate** Variablen in ihre h-Bit-Binärdarstellung um bzw. Binärzahlen wieder in ihre Dezimaldarstellung zurück (abhängig von der Pfeilrichtung).
Im worst-case kann π in $O(\log_2(n))$ Schritten berechnet werden. Wenn man allerdings nur Links- und Rechtsverschiebungen zulässt und die einzelnen Bitfolgen abarbeiten muss, dann werden $O(n)$ Operationen benötigt, um $\pi(i)$ für alle $i \in \{1, ..., n\}$ berechnen zu können. Damit sind alle Voraussetzungen für Satz 4.6.6 erfüllt, da ja das Datenfeld eingangs geordnet war und somit $\sigma = \pi^{-1}$ ist und daher $\sigma\pi(i) = \pi^{-1}\pi(i) = i$ auch in $O(n)$, ja sogar in konstanter Zeit ermittelt werden kann. Es kann also in $O(n)$ Schritten aus einem geordneten Datenfeld der Länge n eine implizite binäre Suchbaum-Datenstruktur erzeugt werden.

Kapitel 5

Erstes Abbruchkriterium: "Fastbreak 1"

Während im vorigen Kapitel verschiedenste Modifikationen und Varianten des klassischen Algorithmus vorgestellt wurden, deren Ziel es war, die worst-case Laufzeit zu verbessern, werden in den beiden Kapiteln "Fastbreak 1" und "Fastbreak 2" Abbruchkriterien untersucht, die deutliche Verbesserungen auch für die erwartete Laufzeit mit sich bringen. Erstmals sind die folgenden Überlegungen bei J. Keller (siehe [11]) gemacht worden. Mit der mathematischen Ausarbeitung beschäftigten sich dann A. Panholzer, H. Prodinger und M. Riedel in [12]. Die Resultate dieser beiden Arbeiten sollen hier betrachtet werden.

5.1 Kernidee der Heuristik "Fastbreak 1"

Bisher waren alle untersuchten Algorithmen ähnlich aufgebaut. Zuerst wurde immer ein auszuzeichnendes Element in jedem Zyklus gesucht und anschließend der Umspeichervorgang mit diesem Element eingeleitet und entlang dieses Zyklus durchgeführt. Dabei hat aber keiner der vorgestellten Algorithmen berücksichtigt, dass wenn man einen Zyklus rotiert, die Länge dieses Zyklus leicht ermittelt und im weiteren Verlauf verwendet werden kann. Addiert man diese im Verlauf des Algorithmus gefundenen Zyklenlängen auf, kann man das Programm beenden, sobald diese Summe n, also die Länge des Datenfelds A, ergibt. Denn dann ist bereits sichergestellt, dass schon alle Elemente umgespeichert wurden. Diese Heuristik bezeichnet J. Keller mit "Fastbreak 1". Die Terminologie soll in dieser Arbeit übernommen werden. Folgender Algorithmus hat diese Kernidee zur Grundlage und entspricht im Wesentlichen jenen von [11, Seite 120]. Er wurde lediglich in die der gesamten Arbeit zugrunde liegenden Form gebracht.

5.2 Algorithmus nach J. Keller

Algorithmus 5.2.1 (Fastbreak 1)

```
1  function ROTATE_π(leader)
2  i := leader
3  length := 1
4  while π(i) ≠ leader do
5     interchange the values in A[i] and A[π(i)]
6     length := length + 1
7     i := π(i)
8  end while.
9  return length
```

```
10  remaining := n
11  for j := 1 to n do
12    k := π(j)
13    while k > j do
14      k := π(k)                                   ▷ a^[FB1]
15    end while
16    if k = j then
17      remaining := remaining − ROTATE_π(k)
18    end if
16    if remaining = 0 then
17      break
18    end if
19  end for
```

5.3 Analyse

5.3.1 Korrektheit

Die Grundstruktur dieses Algorithmus gleicht dem klassischen Algorithmus 3.2.1 auf Seite 17. Folgende Änderungen wurden vorgenommen: Es wird im Hauptprogrammteil zu aller erst die Zählvariable *remaining* eingeführt und ihr gleich zu Beginn die Länge des Datenfelds n zugewiesen. Die in Algorithmus 4.1.3 kennen gelernte Prozedur ROTATE wird nun zu einer Funktion, die zusätzlich zum Umspeichern auch die Länge des Zyklus zurückgeben soll. So kann in Zeile 17 nicht nur implizit die Umspeicher-Routine aufgerufen werden, sondern auch der Wert der Zählvariablen *remaining* aktualisiert werden. Dazu wird in der ROTATE-Funktion zusätzlich eine Variable *length* benötigt, in der die Länge des Zyklus mitgezählt und anschließend zurückgegeben wird. Dieser Wert wird für jeden permutierten Zyklus von der Variable *remaining* abgezogen, bis alle Elemente behandelt wurden und somit $remaining = 0$ gelten muss. In diesem Fall kann das Programm mit dem **break**-Befehl in Zeile 17 terminieren, auch wenn eventuell nicht alle Elemente i auf die Zyklenführereigenschaft hin überprüft wurden. Diese Anzahl der nicht überprüften Elemente ist auch die Ersparnis der Heuristik "Fastbreak 1" gegebenüber dem klassischen Algorithmus.

Bemerkung 5.3.1 *Hier wurde die Prozedur* ROTATE_π *modifiziert. Analog kann man aber auch die Prozedur* $\text{ROTATE}_{\pi^{-1}}$ *aus Algorithmus 4.1.1 adaptieren und es wird daher hier bewusst nur allgemein von der* ROTATE*-Funktion gesprochen.*

5.3.2 Laufzeit

Im Kapitel 3.3.2 wurde, wie bei Knuth in [2], der Parameter a untersucht, der die Durchlaufungsanzahl der inneren while-Schleife beschreibt. Für Keller charakterisiert hingegen die Anzahl der Aufrufe der gegebenen Permutation π im Hauptprogrammteil die Laufzeit, also wie oft $\pi(i)$ für ein übergebenes i ausgeführt wird. Dieser Parameter wird im Folgenden, wie in [12], mit κ bezeichnet. Keller wählte den Parameter κ deshalb als Referenz für die Laufzeit, weil er mögliche Einflüsse auf die Laufzeit, die abhängig von der jeweilig betrachteten Permutation auftreten können, verhindern möchte. Er nennt als Beispiel das Verhalten des "Cache", also eines Speicher-Puffers, der Daten für die mehrmalige Verwendung zwischenspeichert. Außerdem spricht er von der plausiblen Annahme, dass das Laufzeitverhalten von der Auswertung von π dominiert wird und dass die zusätzlich entstehenden Kosten für die Implementierung für die Heuristiken "Fastbreak 1" und "Fastbreak 2" ignoriert werden können. Diese letzte Aussage

kann man in Hinblick auf die besprochenen Änderungen von Algorithmus 5.2.1 gegenüber Algorithmus 3.2.1 sofort nachvollziehen.

Zwischen den beiden Parametern a und κ gibt es im klassischen Algorithmus folgenden einfachen Zusammenhang. Da unmittelbar vor der while-Schleife (bei Algorithmus 3.2.1 in Zeile 2) bereits einmal π ausgewertet wird, zählt κ für jedes Element eins mehr als a. Erinnert man sich an die Darstellung (3.1) des Parameters a auf Seite 18, so würde für κ beim klassischen Algorithmus

$$\kappa(\pi) = |\{\,(i,j): \quad 1 \le i \le j \le n \quad \wedge \quad q(i) \le q(i+1),\, ...,\, q(i) \le q(j)\}| \tag{5.1}$$

gelten, wobei π wieder die kanonische Zyklendarstellung $\pi = (q(1), q(2), ..., q(n))$ haben soll. Somit hat man in diesem Fall für jede beliebige Permutation der Länge n: $\kappa = a + n$ und damit aufgrund von $\mathbb{E}(a) = (n+1)H_n - 2n$:

$$\mathbb{E}(\kappa) = (n+1)H_n - n.$$

Interessant ist, dass dieses Resultat nicht mit der von Keller experimentell festgestellten Laufzeit von $\frac{2}{3} n \log_2 n$ zusammenpasst. Da aber $\frac{2}{3 \log 2} \approx 0{,}96$ gilt, erkennt man, dass es offensichtlich rein experimentell nicht möglich war, diesen Wert von 1 zu unterscheiden, um auf die korrekte asymptotische Laufzeit von $n \log n$ zu gelangen. Wie in [12] treffend festgestellt wurde, ist das ein Vorzeigegrund dafür, sich auf mathematischem Wege mit der Analyse von Algorithmen zu beschäftigen.

Da sich κ und a nur um eins pro Iteration unterscheiden, müssen sie dieselbe Varianz haben. Es ist daher die Varianz für κ durch jene von a gegeben, also

$$\mathbb{V}(\kappa) = \mathbb{V}(a) = 2n^2 - (n+1)^2 H_n^{(2)} - (n+1)H_n + 4n.$$

Asymptotisch verhält sich die Varianz also wie $(2 - \frac{\pi^2}{6})n^2$, da $\sum_{k \ge 1} \frac{1}{k^2} = \frac{\pi^2}{6}$ gilt.

Im Folgenden werden beide Parameter $a^{[FB1]}$ und $\kappa^{[FB1]}$ in der Heuristik "Fastbreak 1" untersucht.

worst-case

Ähnlich dem worst-case im klassischen Algorithmus (siehe Abschnitt 3.3.2) nimmt $a^{[FB1]}$, und somit auch $\kappa^{[FB1]}$, im Algorithmus 5.2.1 den Maximalwert bei $\pi = (\pi(1), \pi(2), ..., \pi(n)) = (2, ..., n-1, 1, n)$ an. Das größte Element n als Fixpunkt der Permutation verhindert, dass die Variable *remaining* den Wert 0 vor der Überprüfung des letzten Elements erreicht. Denn nachdem 1 als Zyklenführer erkannt und der $n-1$ lange Zyklus rotiert wurde, gilt $remaining = 1$ bis ans Ende der Überprüfung und unterbindet somit das vorzeitige Abbrechen. Für den Parameter $a^{[FB1]}$ heißt das nun: $a^{[FB1]} = (n-2)+(n-3)+...+1+0+0 = \binom{n-1}{2} = \frac{1}{2}(n^2-3n+2)$. Da die Heuristik in diesem konstruierten Fall keine Verbesserung gegenüber dem klassischen Algorithmus bringt, gilt somit für Keller's Parameter: $\kappa^{[FB1]} = \kappa = a+n = \frac{1}{2}(n^2-3n+2)+n = \frac{1}{2}(n^2-n+2)$.

average-case

Auch wenn die Heuristik keine asymptotische Verbesserung bringt, sind die Ersparnisse, wie sich zeigen wird, nicht unerheblich. Es wird wieder davon ausgegangen, dass jede Eingangspermutation gleich wahrscheinlich ist. Dann hat eine Permutation der Länge n durchschnittlich $\log n$ Zyklen (siehe Satz 3.3.2 auf Seite 18 oder auch [13, Kapitel 6]). Die erwartete Länge eines Zyklus ist daher $\frac{n}{\log n}$. Daraus ergibt sich intuitiv, dass die Ersparnis für die nicht notwendige Überprüfung des letzten Zyklus circa $\frac{n}{\log n} H_{\lfloor n/\log n \rfloor} \sim n$ beträgt. Diese intuitive Vermutung soll nun durch die exakte Berechnung belegt werden.

In der kanonischen Zyklendarstellung $\pi = (q(1), q(2), ..., q(n))$ ist das erste Element $q(1)$ der größte Zyklenführer. Da die Elemente von 1 bis n aufsteigend getestet werden, ist $q(1)$ jener Zyklenführer, der zuletzt durch den Algorithmus bestimmt wird. Daher muss die bereits kennen gelernte Parameterdarstellung (3.1) auf

$$a^{[FB1]} = |\{\,(i,j):\ 1 \le i < j \le n \quad \wedge \quad q(i) < q(i+1),\ ...,\ q(i) < q(j) \textbf{ und } q(i) \le q(1)\}| \tag{5.2}$$

adaptiert werden. Wie im Abschnitt 3.3.2 sollen wieder Indikatorvariablen $y_{ij}^{[FB1]}$ aus der Darstellung (5.2) abgeleitet werden. Denn wir haben gesehen, dass Knuth den erwarteten Wert mit Hilfe von

$$\mathbb{E}\,(a) = \sum_{1 \le i < j \le n} \mathbb{E}\,(y_{ij})$$

berechnen konnte. Analog soll hier vorgegangen werden, da sich die eleganteren Lösungswege leider nicht auf die beiden Heuristiken "Fastbreak 1" und "Fastbreak 2" übertragen lassen, da man bisher keine entsprechenden Funktionalgleichungen finden konnte.
Die direkte Übersetzung von Darstellung (5.2) liefert:

$$y_{ij}^{[FB1]} = \begin{cases} 1, & \text{wenn } q(i) = \min\{q(i), q(i+1), ..., q(j)\} \textbf{ und } q(i) \le q(1), \\ 0, & \text{in allen übrigen Fällen.} \end{cases}$$

Für $i = 1$ ist die zusätzliche Bedingung $q(i) \le q(1)$ für den Fall $y_{ij}^{[FB1]} = 1$ trivialerweise erfüllt und es gilt $y_{1j}^{[FB1]} = y_{1j}$. Wie in Gleichung (3.4) gilt $\mathbb{E}\,(y_{ij}) = \frac{1}{j-i+1}$ und damit $\mathbb{E}\left(y_{1j}^{[FB1]}\right) = \frac{1}{j}$. Nun soll der Fall $i \ge 2$ behandelt werden. Angenommen, es gelte $q(1) = L$. Wir wollen also die Wahrscheinlichkeit, dass $q(i) = \min\{q(i), q(i+1), ..., q(j)\}$ **und** $q(i) < L$ gilt, behandeln. Die scharfe Ungleichung folgt aus der Tatsache, dass $i \ge 2$ und somit $i \ne 1$ sein muss. Nun kann man folgendes äquivalente Problem behandeln: Es sei $(q(2), q(3), ..., q(n))$ als eine zufällige Permutation von $\{1, ..., n-1\}$ gegeben und L sei eine ganze Zahl mit $0 \le L \le n-1$. Damit ist die ursprünglich gesuchte Wahrscheinlichkeit jene Wahrscheinlichkeit, dass

$$q(i) = \min\{q(i), q(i+1), ..., q(j)\} \textbf{ und } q(i) \le L \tag{5.3}$$

gilt. Die Anzahl der Möglichkeiten für eine Permutation von $\{1, ..., n-1\}$ ist $\frac{1}{(n-1)!}$. Ob nun die Bedingung (5.3) erfüllt ist, hängt von der Auswahl der Elemente ab.

Der Ausdruck

$$(j-i)!\binom{n-1-k}{j-i}$$

ist eine Variation von $j-i$ Elementen aus $n-1-k$ Elementen ohne Wiederholung, also die Anzahl der Anordnungsmöglichkeiten von $j-i$ Elementen aus dem Pool der $n-1-k$ Elemente, bei der die Reihenfolge eine Rolle spielt. Soll die Bedingung (5.3) erfüllt sein, ist dies nur bei gewissen Auswahlen von den Elementen $q(i), q(i+1), ..., q(j)$ möglich. Die Anzahl dieser Auswahlen wird im Folgenden zunächst betrachtet:

$$\sum_{k=1}^{L}(j-i)!\binom{n-1-k}{j-i}.$$

Der Laufindex k kann hier als der Wert $q(i)$ interpretiert werden. Wird $q(i)$ größer gewählt, so nimmt die Auswahlmöglichkeit der $j-i$ folgenden Elemente ab, da diese ja alle größer sein müssen. In der Begrenzung der Summe mit L wird der Forderung $q(i) \leq L$ nachgekommen. Man hat mit der Erfüllbarkeitsforderung von Bedingung (5.3) also die dafür relevanten $j-i+1$ Elemente festgelegt. Die restlichen Elemente in $(q(1), q(2), ..., q(n))$ sind von der Forderung nicht betroffen und können die übrigen Werte beliebig annehmen. Dies erklärt den Faktor $(n-1-j+i-1)! = (n-2-j+i)!$ in nachfolgender Formel. Die gesuchte Wahrscheinlichkeit für ein festes L ist also:

$$\begin{aligned}&\frac{1}{(n-1)!}\sum_{k=1}^{L}(j-i)!\binom{n-1-k}{j-i}(n-2-j+i)!\\ &\quad=\sum_{k=1}^{L}\frac{1}{n-1}\cdot\frac{(j-i)!(n-2-j+i)!}{(n-2)!}\binom{n-1-k}{j-i}=\sum_{k=1}^{L}\frac{1}{n-1}\cdot\frac{\binom{n-1-k}{j-i}}{\binom{n-2}{j-i}}.\end{aligned}$$

Bemerkung 5.3.2 *Man hätte wie in [12] gleich direkt auf die letzte Formel schließen können. Dabei bezeichnet* $\frac{1}{n-1}$ *die eindeutige Auswahl von* $q(i)$, $\binom{n-2}{j-i}$ *die Anzahl der Möglichkeiten die weiteren* $j-i$ *Elemente aus den übrigen* $n-2$ *(also allen außer dem* $q(i)$*) auszuwählen und* $\sum_{k=1}^{L}\binom{n-1-k}{j-i}$ *die Anzahl der Auswahlen, die Bedingung* (5.3) *erfüllen.*

Mit der Summenformel $\sum_{k=0}^{n}\binom{k}{m} = \sum_{k=0}^{n}\binom{n-k}{m} = \binom{n+1}{m+1}$ aus Lemma 1.4.1 auf Seite 7 erhält man für $L < n$ unmittelbar:

$$\begin{aligned}\sum_{k=0}^{L}\binom{n-k}{m}&=\sum_{k=0}^{n}\binom{n-k}{m}-\sum_{k=L+1}^{n}\binom{n-k}{m}=\binom{n+1}{m+1}-\sum_{k=0}^{n-L-1}\binom{k}{m}\\ &=\binom{n+1}{m+1}-\binom{n-L}{m+1}.\end{aligned}$$

Damit lässt sich jetzt weiterrechnen und man erhält:

$$\frac{1}{(n-1)\binom{n-2}{j-i}} \sum_{k=1}^{L} \binom{n-1-k}{j-i} = \frac{1}{(n-1)\binom{n-2}{j-i}} \left[\binom{n-1}{j-i+1} - \binom{n-1-L}{j-i+1} \right].$$

Nun soll der Durchschnitt über alle möglichen Werte von L gebildet werden. Nachdem alle Werte gleich wahrscheinlich sind und $L \in \{0, ..., n-1\}$ gilt, ergibt sich wieder mit Hilfe der Summenformel $\sum_{k=0}^{n} \binom{n-k}{m} = \binom{n+1}{m+1}$ für Binomialkoeffizienten:

$$\begin{aligned}
&\frac{1}{n} \sum_{L=0}^{n-1} \frac{1}{(n-1)\binom{n-2}{j-i}} \left[\binom{n-1}{j-i+1} - \binom{n-1-L}{j-i+1} \right] \\
&= \frac{(j-i)!(n-2-j+i)!}{(n-1)(n-2)!} \cdot \frac{(n-1)!}{(j-i+1)!(n-2-j+i)!} - \frac{1}{n(n-1)\binom{n-2}{j-i}} \binom{n}{j-i+2} \\
&= \frac{1}{j-i+1} - \frac{(j-i)!(n-2-j+i)!}{n(n-1)(n-2)!} \cdot \frac{n!}{(j-i+2)!(n-j+i-2)!} \\
&= \frac{1}{j-i+1} - \frac{1}{(j-i+2)(j-i+1)} \\
&= \frac{1}{j-i+2}.
\end{aligned}$$

Fasst man nun die beiden Fälle $i = 1$ und $i > 1$ zusammen, so kann man den Erwartungswert für den Parameter $a^{[FB1]}$ wie folgt berechnen:

$$\begin{aligned}
\mathbb{E}\left(a^{[FB1]}\right) &= \sum_{1 \le i < j \le n} \mathbb{E}\left(y_{ij}^{[FB1]}\right) = \sum_{2 \le j \le n} \frac{1}{j} + \sum_{2 \le i < j \le n} \frac{1}{j-i+2} \\
&= H_n - 1 + \sum_{3 \le r \le n} \frac{n+1-r}{r} \\
&= H_n - 1 + (n+1)\left(H_n - \frac{3}{2}\right) - (n-2) \\
&= (n+2)H_n - \frac{5n}{2} - \frac{1}{2}.
\end{aligned}$$

Dabei wurde ähnlich wie in Gleichung (3.4) eine Variablensubstitution durchgeführt, nämlich $j-i+2 =: r$. Dazu sind $n+1-r$ die Anzahl der Möglichkeiten für einen bestimmten konstanten Wert der Variablen r, wenn $2 \le i < j \le n$ gelten soll.

Für kleine n ist dieses Resultat auch leicht überprüfbar. Für $n = 1$ erhält man durch Anwenden der Formel den Wert 0; hier wird nie die while-Schleife durchlaufen. Auch für $n = 2$ macht der erhaltene Wert $\frac{1}{2}$ Sinn: entweder es werden die Elemente vertauscht und man hat einen Zyklus der Länge 2, für den man einmal die while-Schleife abarbeiten muss, oder die Permutation ist die Identität, wo man von der while-Schleife fern bleibt. Beide Fälle sind gleich wahrscheinlich, also muss der Erwartungswert $\frac{1}{2}$ sein. Folgendes Beispiel soll noch den Fall $n = 3$ behandeln.

Beispiel 5.3.3 *Laut Formel ergibt sich für den Erwartungswert des Parameters $a^{[FB1]}$ bei einem Datenfeld der Länge $n = 3$ der Wert: $(3+2)H_3 - \frac{15}{2} - \frac{1}{2} = 5\frac{11}{6} - 8 = \frac{7}{6}$. Dieses Resultat soll überprüft werden. Folgende Tabelle listet die sechs möglichen Permutationen in (nicht kanonischer) Zyklendarstellung auf und dazu die jeweilige Größe des Parameters $a^{[FB1]}$:*

Index	*Permutationen*	$a^{[FB1]}$
1	(1)(2)(3)	*0*
2	(1 2)(3)	*1*
3	(1 3)(2)	*1*
4	(1)(2 3)	*1*
5	(1 2 3)	*2*
6	(1 3 2)	*2*

Bei den letzten beiden Permutationen wird nur für das Element 1 die Überprüfung durchgeführt und dort wird zweimal (Überprüfung $1 < 2$ und $1 < 3$) die while-Schleife durchlaufen, da dann bereits alle Elemente umgeordnet wurden und somit der vorzeitige Abbruchbefehl dieser Heuristik zum Tragen kommt. Aus dem selben Grund wird bei der dritten Permutation das Element 3 nicht mehr überprüft. Alle Permutationen sind gleich wahrscheinlich. Mittelt man nun die möglichen Werte für $a^{[FB1]}$ ergibt sich $(0+1+1+1+2+2)\frac{1}{6} = \frac{7}{6}$.

Man erkennt, dass es in Bezug auf die erwartete Laufzeit asympotisch keinen Unterschied zwischen dem klassischen Algorithmus und der Heuristik "Fastbreak 1" gibt.

Nun soll Keller's Parameter $\kappa^{[FB1]}$ behandelt werden. Hier gilt

$$\kappa^{[FB1]} = \left|\{\,(i,j):\ 1 \le i \le j \le n \quad \wedge \quad q(i) \le q(i+1),\, ...,\, q(i) \le q(j) \ \textbf{und}\ q(i) \le q(1)\}\right|. \tag{5.4}$$

Mit genau derselben Überlegung bzw. Berechnung erhält man schließlich:

$$\begin{aligned}
\mathbb{E}\left(\kappa^{[FB1]}\right) &= \sum_{1\le j\le n} \frac{1}{j} + \sum_{1\le i\le j\le n} \frac{1}{j-i+2} \\
&= H_n + \sum_{2\le r\le n} \frac{n+1-r}{r} \\
&= H_n + (n+1)\,(H_n - 1) - (n-1) \\
&= (n+2)H_n - 2n.
\end{aligned}$$

Nun sollen die beiden Varianzen $\mathbb{V}\left(a^{[FB1]}\right)$ und $\mathbb{V}\left(\kappa^{[FB1]}\right)$ berechnet werden. Das zweite Moment $\mathbb{E}\left(\left(a^{[FB1]}\right)^2\right)$ für Knuth's Parameter a errechnet sich aus

$$\mathbb{E}\left(\left(a^{[FB1]}\right)^2\right) = \mathbb{E}\left(a^{[FB1]}\right) + \sum_{\substack{1\le i_1<j_1\le n \\ 1\le i_2<j_2\le n \\ (i_1,j_1)\neq(i_2,j_2)}} \mathbb{E}\left(y_{i_1 j_1}^{[FB1]}\, y_{i_2 j_2}^{[FB1]}\right).$$

Die Betrachtung aller möglichen Fälle für die Indexvariablen i_1, j_1, i_2 und j_2 liefert:

$$\begin{aligned}\mathbb{E}\left(\left(a^{[FB1]}\right)^2\right) &= (n+2)H_n - \frac{5n}{2} - \frac{1}{2} \\ &\quad + 2\sum_{2\le i_1<j_1<i_2<j_2\le n} \frac{1}{j_1-i_1+j_2-i_2+3}\left(\frac{1}{j_2-i_2+2}+\frac{1}{j_1-i_1+2}\right) \\ &\quad + 2\sum_{2\le i_1<i_2<j_2\le j_1\le n} \frac{1}{(j_1-i_1+2)(j_2-i_2+2)} \\ &\quad + 2\sum_{2\le i_1<i_2\le j_1<j_2\le n} \frac{1}{(j_2-i_1+2)(j_2-i_2+2)} + 2\sum_{2\le i<j_1<j_2\le n} \frac{1}{j_2-i+2} \\ &\quad + 2\sum_{2\le j_1<i<j_2\le n} \frac{1}{(j_2-i+j_1+1)j_1} + 2\sum_{2\le j_1<j_2\le n} \frac{1}{j_2}.\end{aligned}$$

Die vorvorletzte und die letzte Summe behandeln den Fall, dass $i_1 = i_2 =: i$ ist. Bei der letzten Summe gilt zusätzlich $i_1 = i_2 = 1$. Bei der vorletzten ist $i_1 = 1$ und $i_2 =: i$. Zur Illustration, wie man zu den einzelnen Summanden kommt, soll hier der Fall $2 \le i_1 < i_2 \le j_1 < j_2 \le n$ betrachtet werden:

$$\begin{aligned}&\mathbb{E}\left(y_{i_1j_1}^{[FB1]}\, y_{i_2j_2}^{[FB1]}\right) = \mathbb{P}\left\{y_{i_1j_1}^{[FB1]}\, y_{i_2j_2}^{[FB1]} = 1\right\} \\ &= \frac{1}{n!}\binom{n}{j_2-i_1+2}\binom{j_2-i_1+1}{j_2-i_2+2}(j_2-i_2+1)!(i_2-i_1+1)!(n-(j_2-i_1+2))! \\ &= \frac{1}{(j_2-i_1+2)(j_2-i_2+2)}.\end{aligned}$$

Die zweite Zeile erhält man, wenn man sich vor Augen führt, wie die Forderung (5.3) für zwei Werte zu erfüllen ist. Es geht im Wesentlichen um geschickte Auswahlen (das sind die Binomialkoeffizienten) und darum, dass die Elemente $q(i+1), ..., q(j)$ beim Erfüllen der Bedingung (5.3) beliebig angeordnet werden können – nur das $q(i)$ muss das kleinste Element sein und ist somit fixiert. Daher kommt es, dass die folgenden faktoriellen Ausdrücke um eins kleiner sind, als die auszuwählenden Elemente im Binomialkoeffizient (also das k in $\binom{n}{k}$). Der Grund, warum dieser untere Term im Binomialkoeffizient nicht bloß die Länge des zu betrachtenden Zahlenstrings, also etwa $j_2 - i_1 + 1$ beinhaltet, sondern um eins größer ist, ist der, dass man ja auch die Bedingung $q_i \le q_1$ erfüllen muss und man daher auch das Element q_1 mit auswählt. Konkret sieht das so aus: Von den ausgewählten $j_2 - i_1 + 2$ Elementen kommt das kleinste auf die Position i_1. Von den restlichen $j_2 - i_1 + 1$ ausgewählten Elementen werden weitere $j_2 - i_2 + 2$ Elemente ausgewählt und das kleinste dieser zweiten Auswahl kommt auf die Position i_2. Die restlichen der $j_2 - i_2 + 1$ Elemente der zweiten Auswahl kommen (beliebig angeordnet) auf die Positionen 1 und $i_2, ..., j_2$. Die $i_2 - i_1 - 1$ Elemente der ersten Auswahl, die nicht in die zweite Auswahl aufgenommen wurden, werden beliebig (auf $(i_2 - i_1 - 1)!$ mögliche Arten) auf die Positionen $i_1 + 1, ..., i_2 - 1$ verteilt. Der letzte faktorielle Ausdruck $(n - (j_2 - i_1 + 2))!$ kommt daher, dass es für die restlichen Elemente $q(k)$ mit $k \notin \{1, i_1, i_1 + 1, ..., j_2\}$ unerheblich ist, wie sie angeordnet werden.

Als nächstes müssen die so erhaltenen Summen vereinfacht werden. Exemplarisch soll dies wieder für die eben betrachtete Summe gezeigt werden:

$$
\begin{aligned}
&\sum_{2\le i_1<i_2\le j_1<j_2\le n} \frac{1}{(j_2-i_1+2)(j_2-i_2+2)} \\
&= \sum_{i_1=2}^{n-2}\sum_{i_2=i_1+1}^{n-1}\sum_{j_2=i_2+1}^{n}\sum_{j_1=i_2}^{j_2-1} \frac{1}{(j_2-i_1+2)(j_2-i_2+2)} \\
&= \sum_{i_1=2}^{n-2}\sum_{j_2=i_1+2}^{n}\sum_{i_2=i_1+1}^{j_2-1} \frac{j_2-1-(i_2-1)}{(j_2-i_1+2)(j_2-i_2+2)} \\
&= \sum_{i_1=2}^{n-2}\sum_{j_2=i_1+2}^{n}\sum_{i_2=i_1+1}^{j_2-1} \frac{1}{j_2-i_1+2}\left(1-\frac{2}{j_2-i_2+2}\right) \\
&= \sum_{i_1=2}^{n-2}\sum_{j_2=i_1+2}^{n} \frac{1}{j_2-i_1+2}\left(j_2-i_1-1-2\left(H_{j_2-i_1+1}-\frac{3}{2}\right)\right) \\
&= \sum_{i_1=2}^{n-2}\sum_{j_2=i_1+2}^{n} \frac{1}{j_2-i_1+2}\left(j_2-i_1-1-2\left(H_{j_2-i_1+2}-\frac{1}{j_2-i_1+2}\right)+3\right) \\
&= \sum_{i_1=2}^{n-2}\sum_{j_2=i_1+2}^{n} \left(1-2\frac{H_{j_2-i_1+2}}{j_2-i_1+2}+\frac{2}{(j_2-i_1+2)^2}\right) \quad \text{[wende Lemma 5.3.4 an]} \\
&= \sum_{i_1=2}^{n-2}\left(n-i_1-1-H^2_{n-i_1+2}-H^{(2)}_{n-i_1+2}+H_3^2+H_3^{(2)}+2H^{(2)}_{n-i_1+2}-2H_3^{(2)}\right) \\
&= \sum_{k=1}^{n-3} k-\sum_{k=4}^{n}H_k^2+\sum_{k=4}^{n}H_k^{(2)}+\left(H_3^2-H_3^{(2)}\right)(n-3) \quad \text{[wende Lemmata 5.3.5 \& 5.3.6 an]} \\
&= \frac{(n-3)(n-2)}{2}-(n+1)H_n^2+(2n+1)H_n-2n+4H_3^2-7H_3+6 \\
&\quad +(n+1)H_n^{(2)}-H_n-4H_3^{(2)}+H_3+(n-3)\left(H_3^2-H_3^{(2)}\right).
\end{aligned}
$$

Für die letzten drei Gleichungen wurden die schon bekannten Lemmata 3.4.4 und 3.4.5, sowie die folgenden verwendet.

Lemma 5.3.4 *Für $n\in\mathbb{N}, n\ge 1$ gilt:*

$$\sum_{k=1}^{n}\frac{H_k}{k}=\frac{1}{2}\left(H_n^2+H_n^{(2)}\right).$$

Beweis. Durch Vertauschung der Summationsreihenfolge erhält man:

$$
\begin{aligned}
\sum_{k=1}^{n}\frac{1}{k}H_k &= \sum_{k=1}^{n}\sum_{j=1}^{k}\frac{1}{k}\frac{1}{j}=\sum_{j=1}^{n}\sum_{k=j}^{n}\frac{1}{k}\frac{1}{j}=\sum_{j=1}^{n}\frac{1}{j}\sum_{k=j}^{n}\frac{1}{k}=\sum_{j=1}^{n}\frac{1}{j}\left(H_n-H_{j-1}\right) \\
&= H_n^2-\sum_{j=1}^{n}\frac{1}{j}\left(H_j-\frac{1}{j}\right)=H_n^2-\sum_{j=1}^{n}\frac{1}{j}H_j+\sum_{j=1}^{n}\frac{1}{j^2}=H_n^2-\sum_{j=1}^{n}\frac{1}{j}H_j+H_n^{(2)},
\end{aligned}
$$

$$2\sum_{k=1}^{n}\frac{1}{k}H_k=H_n^2+H_n^{(2)}.$$

□

Lemma 5.3.5 *Für $n \in \mathbb{N}, n \geq 1$ gilt:*

$$\sum_{k=1}^{n} H_k^2 = (n+1)H_n^2 - (2n+1)H_n + 2n$$

Beweis. Hier wird mehrmals das Lemma 3.4.4 auf Seite 27 verwendet.

$$\begin{aligned}
\sum_{k=1}^{n} H_k\, H_k &= \sum_{k=1}^{n}\sum_{j=1}^{k} H_k \frac{1}{j} = \sum_{j=1}^{n}\sum_{k=j}^{n} H_k \frac{1}{j} = \sum_{j=1}^{n}\frac{1}{j}\sum_{k=j}^{n} H_k = \sum_{j=1}^{n}\frac{1}{j}\left(\sum_{k=1}^{n} H_k - \sum_{k=1}^{j-1} H_k\right) \\
&= \sum_{j=1}^{n}\frac{1}{j}\left((n+1)H_n - n - jH_{j-1} + j + 1\right) \\
&= (n+1)H_n^2 - (n-1)H_n - \sum_{j=1}^{n}\left(H_j - \frac{1}{j}\right) + n \\
&= (n+1)H_n^2 - (n-1)H_n - (n+1)H_n + n + H_n + n \\
&= (n+1)H_n^2 - (2n+1)H_n + 2n.
\end{aligned}$$

□

Lemma 5.3.6 *Für $n \in \mathbb{N}, n \geq 1$ gilt:*

$$\sum_{k=1}^{n} H_k^{(2)} = (n+1)H_n^{(2)} - H_n.$$

Beweis.

$$\sum_{k=1}^{n} H_k^{(2)} = \sum_{k=1}^{n}\sum_{j=1}^{k}\frac{1}{j^2} = \sum_{j=1}^{n}\sum_{k=j}^{n}\frac{1}{j^2} = \sum_{j=1}^{n}\frac{1}{j^2}\sum_{k=j}^{n} 1 = \sum_{j=1}^{n}\frac{1}{j^2}(n-j+1) = (n+1)H_n^{(2)} - H_n.$$

□

Bemerkung 5.3.7 *In [12] wird darauf verwiesen, dass man die benötigten Formeln auch mit Hilfe von erzeugenden Funktionen beweisen kann. Dazu kann man beispielsweise von*

$$\sum_{n\geq 1} H_n z^n = \frac{1}{1-z}\log\frac{1}{1-z}$$

ausgehen und durch Integration, Differentiation, usw. auf die Aussagen der Lemmata schließen.

Wenn man alle auftretenden Summen auf diese Weise behandelt, kann man schließlich vom dadurch erhaltenen zweiten Moment das Quadrat des Erwartungswertes abziehen und erhält so die Varianz:

$$\begin{aligned}
\mathbb{V}\left(a^{[FB1]}\right) &= \mathbb{E}\left(\left(a^{[FB1]}\right)^2\right) - \left(\mathbb{E}\left(a^{[FB1]}\right)\right)^2 \\
&= -(2n+3)H_n^2 - (n+1)^2 H_n^{(2)} + (3n-2)H_n + \frac{31}{12}n^2 + \frac{11}{2}n - \frac{1}{12}.
\end{aligned}$$

Asymptotisch verhält sich $\mathbb{V}\left(a^{[FB1]}\right)$ also wie folgt:

$$\mathbb{V}\left(a^{[FB1]}\right) = \left(\frac{31}{12} - \frac{\pi^2}{6}\right) n^2 + O(n \log^2 n).$$

Für Keller's Parameter $\kappa^{[FB1]}$ erhält man durch analoge Rechnung:

$$\begin{aligned}\mathbb{V}\left(\kappa^{[FB1]}\right) &= -(2n+3)H_n^2 - (n+1)^2 H_n^{(2)} + (n-4)H_n + 3n(n+3) \\ &= \left(3 - \frac{\pi^2}{6}\right) n^2 + O(n \log^2 n).\end{aligned}$$

Kapitel 6

Zweites Abbruchkriterium: "Fastbreak 2"

Die Heuristik "Fastbreak 2" baut auf der bereits kennen gelernten Heuristik "Fastbreak 1" auf und versucht diese noch weiter zu verbessern.

6.1 Kernidee der Heuristik "Fastbreak 2"

Nachdem bei der Heuristik "Fastbreak 1" bereits eine Variable eingeführt wurde (*remaining*), welche die im Verlauf des Algorithmus gefundenen Zyklenlängen aufaddiert, kann man diesen Variablenwert auch bei der Überprüfung verwenden, ob es sich bei einem Element um einen Zyklenführer handelt. Denn wurden nur r Elemente noch nicht umgespeichert bzw. als Fixpunkte erkannt, ist es sinnlos, wenn man bei der Überprüfung auf die Zyklenführereigenschaft für ein Element i länger als r Schritte im jeweiligen Zyklus voranschreitet. Denn dann beinhaltet der Zyklus sicherlich ein Element, das bereits umgespeichert wurde und somit wurde bereits der gesamte Zyklus rotiert. Es wurde also bereits ein anderes Element als Zyklenführer ausgemacht und das betrachtete i kommt nicht mehr als Zyklenführer in Frage – man kann daher vorzeitig die Überprüfung beenden und die Schleife verlassen.
Da "Fastbreak 2" eine Erweiterung von "Fastbreak 1" ist, endet das Programm auch hier, sobald die aufaddierten Zyklenlängen als Summe n, also die Länge des Datenfelds A, ergeben. Folgender Algorithmus versucht, ausgehend von der Heuristik "Fastbreak 1", diese Kernidee umzusetzen und ist wieder an den Algorithmus von [11, Seiten 120-122], und an die Überarbeitung von [12, Seite 8], Algorithmus 2, angelehnt.

6.2 Algorithmus nach J. Keller

Algorithmus 6.2.1 (Fastbreak 2)

```
1  function ROTATE_π(leader)
2  i := leader
3  length := 1
4  while π(i) ≠ leader do
5     interchange the values in A[i] and A[π(i)]
6     length := length + 1
7     i := π(i)
8  end while.
9  return length.
```

```
10  function IS_LEADER(j)
11  k := j
12  leader := true
13  steps := 0
14  repeat
15    k := π(k)
16    steps := steps + 1
17    if k < j or steps > remaining then
18      leader := false
19      break
20    end if
21  until k = j
22  return leader
23
24  remaining := n
25  for j := 1 to n do
26    if IS_LEADER(j) = true then
27      remaining := remaining − ROTATE_π(k)
28    end if
29    if remaining = 0 then
30      break
31    end if
32  end for
```

6.3 Verbesserung des Algorithmus "Fastbreak 2"

Hier wird die modifizierte Variante der Funktion IS_LEADER(j) von Panholzer, Prodinger und Riedel (siehe [12, Seite 8, Algorithmus 3]) vorgestellt. In der folgenden Analyse wird auf beide Varianten eingegangen und erklärt, warum die folgende Funktion effizienter arbeitet als die ursprünglich von Keller vorgestellte.

Algorithmus 6.3.1 (Verbesserung der Überprüfungsfunktion)

```
1   function IS_LEADER(j)
2   leader := true
3   k := π(j)
4   steps := 1
5   while k > j and steps < remaining do
6     k := π(k)                                  ▷ a^[FB2]
7     steps := steps + 1
8   end while
9   if k ≠ j then
10    leader := false
11  end if
12  return leader
```

6.4 Analyse

6.4.1 Korrektheit

Gegenüber der Heuristik "Fastbreak 1" wurde hier die Überprüfung auf Zyklenführereigenschaft in die Funktion IS_LEADER(j) ausgelagert und modifiziert. Der restliche Programmablauf ist derselbe. In den beiden Funktionen IS_LEADER(j) taucht jetzt die neue Variable *steps* auf, deren Aufgabe es ist, die Anzahl der Aufrufe von π im jeweiligen Zyklus mitzuzählen; also quasi die "Schritte" entlang π, die man voranschreiten muss. In der ersten Variante wird mit diesem Wert in Zeile 17 überprüft, ob man schon mehr Nachfolger von i entlang π betrachtet hat, als es noch zu permutierende Elemente gibt. In diesem Fall wird in Zeile 19 die Überprüfung von i abgebrochen und aufgrund der Zuweisung in Zeile 18 schließlich am Ende (Zeile 22) der Negativwert $false$ zurückgegeben. In der zweiten Variante gewährleistet die zweite Bedingung in Zeile 5, dass die Schrittanzahl kleiner als die übrigen Elemente ist; ansonsten wird die while-Schleife verlassen. In Zeile 9 wird untersucht, ob der letzte Zyklenführer gefunden wurde (also die Bedingung nicht erfüllt ist), oder ob sich i in einem Zyklus befindet, der schon rotiert wurde.

Bemerkung 6.4.1 *Auch hier gilt wieder, dass statt der ursprünglichen Prozedur* ROTATE_π *auch die Prozedur* $\text{ROTATE}_{\pi^{-1}}$ *aus Algorithmus 4.1.1 modifiziert werden kann.*

6.4.2 Laufzeit

Bei der Heuristik "Fastbreak 2" sollen wieder die beiden Parameter a und κ untersucht werden. Zuerst werden aber die beiden Varianten der Funktion IS_LEADER(j) miteindander verglichen. Auch wenn, wie in [12] bemerkt und später auch hier ersichtlich wird, asymptotisch bei beiden gleich viele Funktionsaufrufe benötigt werden (gemeint sind Aufrufe von π), ist die Anzahl dieser Aufrufe bei der zweiten Variante kleiner und somit auch die Parameter a und κ, was sich speziell bei kleineren Datenfeldern deutlich zeigt. Als Beispiel wird in [12] die kleinste Permutation angeführt, bei der bereits ein Unterschied vorliegt. Dieses Beispiel soll hier gezeigt und erklärt werden.

Beispiel 6.4.2 *Ein Datenfeld der Länge* 4 *habe folgende kanonische Zyklendarstellung:* $(4)(1\ 2\ 3)$. *Die Überprüfung beginnt in jedem Fall mit dem Element* $i = 1$, *da ja der Hauptprogrammteil immer der gleiche ist. Beide Varianten benötigen je drei Aufrufe von* π *– dann ist bei Keller's Variante die "until"-Bedingung in Zeile 21, nämlich* $k = j$, *erfüllt und in der zweiten Variante beendet das Nicht-Erfüllen der Bedingung* $k < j$ *die while-Schleife in Zeile 5. In beiden Fällen wird also mit Zyklenführer* 1 *der Zyklus* $(1\ 2\ 3)$ *rotiert und die Variable remaining auf den Wert* $remaining = 4 - 3 = 1$ *gesetzt. Beim nächsten zu überprüfenden Element* $i = 2$ *kann man ja schon nach einem Schritt stoppen, denn mit* $\pi(2) = 3 \neq 2$ *weiß man sofort, dass man sich in einem nichttrivialen Zyklus (Länge größer 1) befindet, während man aber aufgrund der remaining-Variable nur nach einen einelementigen Zyklus sucht. Die erste Variante benötigt aber zwei Aufrufe von* π *um* $steps = 2$ *zu erreichen (gestartet wird immer mit* $steps = 0$*) und somit die Abbruchbedingung in Zeile 17 zu erfüllen (da nach dem ersten Durchlauf auch die erste Abbruchbedingung wegen* $\pi(2) = 3 > 2$ *nicht zutrifft). Bei der zweiten Variante hingegen reicht schon der Wert* $steps = 1$ *um die zweite Bedingung in Zeile 5 zu falsifizieren und somit die while-Schleife zu verlassen (da* π *gleich zu Beginn aufgerufen wird und daher die Variable steps mit dem Wert 1 startet, wird die Schleife hier nie durchlaufen). Also benötigt Algorithmus 6.2.1 in der* IS_LEADER*-Funktion einen zusätzlichen Aufruf von* π *im Vergleich zu der abgeänderten Funktion in Algorithmus 6.3.1. Für die restlichen Elemente* 3 *und* 4 *ist bei beiden Varianten je ein Aufruf von* π *erfoderlich. Somit hat der Parameter* κ *im Algorithmus 6.2.1 den Wert 7 während er mit der Modifikation von Algorithmus 6.3.1 lediglich den Wert 6 aufweist.*

Nachdem sich also die zweite Variante als effizienter herausgestellt hat, wird im folgenden $a^{[FB2]}$ und $\kappa^{[FB2]}$ in Bezug auf Algorithmus 6.3.1 behandelt.

worst-case

Anders als bei den bisherigen Algorithmen ist der worst-case hier nicht leicht einzusehen. Es ist auch in der Literatur kein Hinweis zu finden, wie dieser worst-case bei der Heuristk Fastbreak 2 aussehen könnte. Nach einer Vermutung von A. Panholzer ([17]) nimmt der Parameter a bei jener Permutation den größten Wert an, die aus zwei Zyklen mit dem (auf ganze Zahlen gerundeten) Längenverhältnis $\phi : (1-\phi)$ besteht, wobei

$$\phi := \frac{\sqrt{5}-1}{2} = 0{,}618... \ , \qquad \phi : (1-\phi) = \frac{\sqrt{5}-1}{2} : \frac{3-\sqrt{5}}{2},$$

das Verhältnis des goldenen Schnitts (englisch: "golden ratio") sein muss. Bei einer Permutation der Länge n sollte der größere Zyklus die Länge $\lceil n \cdot \phi \rceil$ haben, wenn $\lceil n \cdot \phi \rceil - n \cdot \phi \leq n \cdot \phi - \lfloor n \cdot \phi \rfloor$ gilt und $\lfloor n \cdot \phi \rfloor$ sonst (das entspricht einer Länge von $\lfloor n \cdot \phi + 0{,}5 \rfloor = \lceil n \cdot \phi - 0{,}5 \rceil$; es wird also auf die nächstgelgene ganze Zahl auf- oder abgerundet). Für den kleineren Zyklus bleibt dann die Restlänge. Diese Vermutung wurde experimentell für Permutationen bis zu einer Länge von $n = 10$ getestet (siehe Anhang). Dabei sind die Fälle ab $n = 5$ interessant, da zuvor die Permutationsgröße noch so klein ist, dass a maximal den Wert $n-1$ annehmen kann. Es ist also bis zur Permutationslänge $n = 4$ der worst-case eine Permutation mit einem Zyklus.

Im Anhang wurde auch die Vermutung ([17]) für das worst-case Szenario bezüglich Parameter κ für Permutationen bis zur Länge $n = 10$ verifiziert. Es wird behauptet, dass der worst-case für Permutationen mit Länge n genau in dem Fall angenommen wird, wenn die Zyklen folgende Länge aufweisen: Der größte Zyklus hat die Länge $\lceil n \cdot \phi \rceil =: h_1$. Damit bleibt für die übrigen Zyklen eine Gesamtlänge von $n_2 := n - h_1$ übrig. Der nächstgrößere Zyklus hat jetzt die Länge $\lceil n_2 \cdot \phi \rceil =: h_2$. Es bleibt nun eine Länge von $n_3 := n_2 - h_2 = n - h_1 - h_2$ übrig. Der drittgrößte Zyklus hat die Länge $\lceil n_3 \cdot \phi \rceil =: h_3$. Auf diese Art und Weise wird die verbleibende Zyklenlänge in immer kleiner werdende Zyklen aufgeteilt, bis am Ende $n_j := n_{j-1} - h_{j-1} = n - \sum_{i=1}^{j-1} h_i = 0$ gilt, also keine Restlänge mehr übrig bleibt. Im Experiment erkennt man, dass es für alle $n = 1, ..., 10$ zumindest einen solchen worst-case gibt, der dieser Behauptung entspricht.

average-case

Hier soll nun die Berechnung für $\mathbb{E}\left(\kappa^{[FB2]}\right)$ durchgeführt werden und im Anschluss das auf die gleiche Weise ermittelbare Ergebnis für $\mathbb{E}\left(a^{[FB2]}\right)$ angegeben werden. Der Parameter $\kappa^{[FB2]}$ hat folgende Interpretation:

$$\begin{aligned}\kappa^{[FB2]} = \Bigg| \Bigg\{ (i,j): \quad & 1 \leq i \leq j \leq n \quad \wedge \quad \Big[q(i) \leq q(i+1), ..., q(i) \leq q(j) \Big] \quad \textbf{und} \\ & \Big[\Big(j - i + 2 \leq \min_{k<i} \big\{ k: \ q(k) < q(i) \ \ \textbf{und} \ \ q(1) > q(i), ..., q(k-1) > q(i) \big\} \Big) \\ & \qquad \textbf{oder} \quad \Big(q(1) > q(i), ..., q(i-1) > q(i) \Big) \Big] \Bigg\} \Bigg|. \quad (6.1)\end{aligned}$$

Selbst wenn $i = j$ gilt, ist mit $2 \leq \min_{k<i} \{k : \ q(k) < q(i) \ \textbf{und} \ q(1) > q(i), ..., q(k-1) > q(i)\}$ sichergestellt, dass $q(1) > q(i)$ und somit die Abbruchbedingung von Heuristik "Fastbreak 1" auch hier realisiert ist. Das Element $q(1)$ ist der größte und somit der zuletzt gefundene Zyklenführer. Die Anzahl der Elemente, die diesem $q(1)$ in der kanonischen Zyklendarstellung folgen und größer sind, gehören zum letzten Zyklus und sind – sollte $q(1)$ noch nicht gefunden worden sein – auch noch nicht durch den Algorithmus umgespeichert worden; die *remaining*-Variable ist also zumindest so groß wie die Anzahl dieser Elemente des letzten Zyklus. Die nächst kleineren Zyklenführer – sofern es diese gibt – können in der kanonischen Zyklendartellung ab der zweiten Position auftauchen, für deren Zyklenelemente dasselbe gilt. Somit sind die ersten Elemente in der kanonischen Zyklendartellung die letzten, die umgespeichert werden. Die Variable *remaining* hat als Wert $remaining = k - 1$, wenn $q(k)$ der zuletzt gefundene Zyklenführer ist. Sollte jetzt für das zu betrachtende Paar (i, j) gelten, dass $q(i)$ größer als einer der Werte $q(m)$ ist, mit $1 \leq m \leq j - i + 1$ (der erste Index $m = k$ bei dem dies auftritt muss ein Zyklenführer sein), dann würde der Algorithmus sofort enden, da die Variable *remaining* kleiner als die Schrittlänge, also der Wert der Varialben *steps*, ist. Diesen Sachverhalt spiegelt die zweite Zeile der Gleichung (6.1) wider: Abhängig vom der Länge des Intervalls von i bis j, die der Anzahl der Durchläufe der Schleife entspricht, darf der erste Wert in der Zyklendarstellung, der kleiner als $q(i)$ ist, erst frühestens an der $(j - i + 2)$-ten Stelle sein.
Die letzte Zeile berücksichtigt den Fall, dass man für das Element $q(i)$ immer die gesamte Schleife bis zum Ende abarbeiten muss, da $q(i)$ kleiner als sämtliche Vorgänger in der kanonischen Zyklendarstellung ist und somit ein gefundener Zyklenführer im Algorithmus ist.

Nun soll der Ewartungswert von $\kappa^{[FB2]}$ berechnet werden. Dazu erhält man mit nachfolgenden Überlegungen aus der Gleichung (6.1):

$$\mathbb{E}\left(\kappa^{[FB2]}\right) = \sum_{1 \leq i \leq j \leq n} \frac{1}{n!} \sum_{L=1}^{n} \left[\binom{n-L}{j-1} (j-1)!(n-j)! + \sum_{k=j-i+2}^{i-1} \binom{n-L}{j-i+k-1} (j-i+k-1)!(L-1)(n-k-(j-i)-1)! \right]. \quad (6.2)$$

In obiger Gleichung werden für $1 \leq i \leq j \leq n$ alle möglichen Werte für die Variable $L := q(i)$ erfasst. Dabei kennzeichnet

$$\binom{n-L}{j-1} (j-1)!(n-j)!$$

die Umsetzung der letzten Zeile von der Gleichung (6.1): Es werden im Binomialkoeffizient jene $j - 1$ Elemente ausgewählt, die größer als $q(i)$ sind, also $q(1)$ bis $q(i-1)$ und $q(i+1)$ bis $q(j)$. Diese kann man auf $(j-1)!$ verschiedene Arten anordnen (es handelt sich also um eine Variation von $j - 1$ aus $n - L$ Elementen). Die übrigen Elemente können beliebig angeordnet werden, was auf $(n - j)!$ verschiedene Möglichkeiten gemacht werden kann. Die Summe in der zweiten Zeile von Gleichung (6.2) entspricht der anderen Möglichkeit für ein (i, j) aus Gleichung (6.1) in der Menge zu sein. Diese Summe kann aufgrund der Bedingung $q(k) < q(i)$ nur bis zum Index $i - 1$ laufen und darf wegen der Ungleichung $j - i + 2 \leq \min\{...\}$ erst bei $k = j - i + 2$ starten. Es brauchen hier nur $j - i + k - 1$ Elemente ausgewählt werden, die größer als $q(i)$ sind. Das sind wie vorhin die Elemente von $q(i+1)$ bis $q(j)$ und dazu noch die Elemente von $q(1)$ bis $q(k-1)$.

Mit dem Faktor $(j-i+k-1)!$ erhält man wieder die gewünschte Variation. Um nun noch die zusätzliche Bedingung $q(k) < q(i) = L$ zu erfüllen, muss $q(k)$ einen der $(L-1)$ kleineren Werte annehmen – so erklärt sich der Faktor $(L-1)$. Der letzte Faktor $(n-k-(j-i)-1)!$ wird den Möglichkeiten für die Anordnung der restlichen Elemente gerecht.

Nun soll eine geschlossene Formel für die Gleichung (6.2) gefunden werden. Dazu wird zuerst der einfachere Teil behandelt:

$$\sum_{i=1}^{n}\sum_{j=i}^{n}\frac{1}{n!}\sum_{L=1}^{n}\binom{n-L}{j-1}(j-1)!(n-j)! = \frac{1}{n}\sum_{i=1}^{n}\sum_{j=i}^{n}\frac{1}{\binom{n-1}{j-1}}\sum_{L=1}^{n}\binom{n-L}{j-1}$$
$$= \frac{1}{n}\sum_{i=1}^{n}\sum_{j=i}^{n}\frac{\binom{n}{j}}{\binom{n-1}{j-1}} = \sum_{i=1}^{n}\sum_{j=i}^{n}\frac{1}{j} = \sum_{j=1}^{n}\frac{1}{j}\sum_{i=1}^{j}1 = n.$$

Die erste verwendete Summenformel folgt aus der schon bekannten Formel $\sum_{k=0}^{n}\binom{n-k}{m} = \binom{n+1}{m+1}$ und der Grundformel für Binomialkoeffizienten $\binom{n+1}{m+1} - \binom{n}{m} = \binom{n}{m+1}$.

Der andere Teil aus Gleichung (6.2) ist etwas aufwendiger zu handhaben. Der erste Schritt ist:

$$\sum_{i=1}^{n}\sum_{j=i}^{n}\frac{1}{n!}\sum_{L=1}^{n}\sum_{k=j-i+2}^{i-1}\binom{n-L}{j-i+k-1}(j-i+k-1)!(L-1)(n-k-(j-i)-1)!$$
$$= \frac{1}{n(n-1)}\sum_{i=1}^{n}\sum_{j=i}^{n}\sum_{k=j-i+2}^{i-1}\frac{1}{\binom{n-2}{j-i+k-1}}\sum_{L=1}^{n}\binom{n-L}{j-i+k-1}(L-1)$$
$$= \frac{1}{n-1}\sum_{i=1}^{n}\sum_{j=i}^{n}\sum_{k=j-i+2}^{i-1}\frac{1}{\binom{n-2}{j-i+k-1}}\sum_{L=1}^{n}\binom{n-L}{j-i+k-1}$$
$$- \frac{1}{n}\sum_{i=1}^{n}\sum_{j=i}^{n}\sum_{k=j-i+2}^{i-1}\frac{1}{\binom{n-1}{j-i+k}}\sum_{L=1}^{n}\binom{n-L+1}{j-i+k}$$
$$= \frac{1}{n-1}\sum_{i=1}^{n}\sum_{j=i}^{n}\sum_{k=j-i+2}^{i-1}\frac{\binom{n}{j-i+k}}{\binom{n-2}{j-i+k-1}} - \frac{1}{n}\sum_{i=1}^{n}\sum_{j=i}^{n}\sum_{k=j-i+2}^{i-1}\frac{\binom{n+1}{j-i+k+1}}{\binom{n-1}{j-i+k}}$$
$$= \sum_{i=1}^{n}\sum_{j=i}^{n}\sum_{k=j-i+2}^{i-1}\left(\frac{n}{(j-i+k)(n-(j-i)-k)} - \frac{n+1}{(j-i+k+1)(n-(j-i)-k)}\right).$$

Um die vorletzte Zeile zu erhalten wurde wieder die Formel $\sum_{k=0}^{n}\binom{n-k}{m} = \binom{n+1}{m+1}$ bemüht.

Jetzt wird die Variablensubstitution $\ell := j - i$ durchgeführt und es ergibt sich:

$$\begin{aligned}
&\sum_{i=1}^{n}\sum_{\ell=0}^{n-i}\sum_{k=\ell+2}^{i-1}\left(\frac{n}{(\ell+k)(n-\ell-k)} - \frac{n+1}{(\ell+k+1)(n-\ell-k)}\right)\\
&= \sum_{i=1}^{n}\sum_{\ell=0}^{n-i}\sum_{k=\ell+2}^{i-1}\left(\frac{n(\ell+k+1)-(n+1)(\ell+k)}{(\ell+k)(n-\ell-k)(\ell+k+1)}\right)\\
&= \sum_{i=1}^{n}\sum_{\ell=0}^{n-i}\sum_{k=\ell+2}^{i-1}\left(\frac{[n(\ell+k)-n(\ell+k)]+(n-\ell-k)}{(\ell+k)(n-\ell-k)(\ell+k+1)}\right)\\
&= \sum_{i=1}^{n}\sum_{\ell=0}^{n-i}\sum_{k=\ell+2}^{i-1}\left(\frac{1}{(\ell+k)(\ell+k+1)}\right) = \sum_{i=1}^{n}\sum_{\ell=0}^{n-i}\sum_{k=\ell+2}^{i-1}\left(\frac{1}{\ell+k} - \frac{1}{\ell+k+1}\right).
\end{aligned}$$

Hier erkennt man, dass die letzte, durch Partialbruchzerlegung erhaltene Summe für $1 \leq i \leq 2$ aufgrund der oberenen Grenze $i-1$ und dem Startindex $k = 2$ keinen Beitrag liefert. Daher kann man die erste Summe auch erst mit $i = 3$ starten lassen. Wegen der oberen Grenze der letzten Summe reicht es auch, dass ℓ maximal bis $i-3$ läuft. Somit erhält man:

$$\begin{aligned}
&\sum_{i=1}^{n}\sum_{\ell=0}^{n-i}\sum_{k=\ell+2}^{i-1}\left(\frac{1}{\ell+k} - \frac{1}{\ell+k+1}\right) = \sum_{i=3}^{n}\sum_{\ell=0}^{\min(i-3,n-i)}\sum_{k=\ell+2}^{i-1}\left(\frac{1}{\ell+k} - \frac{1}{\ell+k+1}\right)\\
&= \sum_{i=3}^{n}\sum_{\ell=0}^{\min(i-3,n-i)}\left(\frac{1}{2\ell+2} - \frac{1}{\ell+i}\right).
\end{aligned}$$

Die letzte Umformung war das Auflösen der Teleskopsumme. Hier bleiben nur der erste und letzte Summand übrig, die restlichen Summanden heben sich gegenseitig auf.

Der nächste Schritt ist die Vertauschung der Summationsreihenfolge. Dabei muss man sich die obere Grenze für ℓ neu überlegen. Würde $0 \leq \ell \leq \min(i, n-i)$ mit $0 \leq i \leq n$ gelten, dann hätte man für ℓ, aufgrund der Symmetrieeigenschaft der beiden Ausdrücke in der min-Überprüfung, den Bereich von 0 bis $\lfloor \frac{n}{2} \rfloor$. Im vorliegenden Fall ($0 \leq \ell \leq \min(i-3, n-i)$ mit $3 \leq i \leq n$) muss sich daher der Laufindex ℓ im Wertebereich von 0 bis $\lfloor \frac{n-3}{2} \rfloor$ bewegen. Daher ergibt sich:

$$\begin{aligned}
&\sum_{i=3}^{n}\sum_{\ell=0}^{\min(i-3,n-i)}\left(\frac{1}{2\ell+2} - \frac{1}{\ell+i}\right) = \sum_{\ell=0}^{\lfloor\frac{n-3}{2}\rfloor}\sum_{i=\ell+3}^{n-\ell}\left(\frac{1}{2\ell+2} - \frac{1}{\ell+i}\right)\\
&= \sum_{\ell=0}^{\lfloor\frac{n-3}{2}\rfloor}\left(\frac{n}{2\ell+2} - 1 - H_n + H_{2\ell+2}\right).
\end{aligned}$$

Um die letzte Summe auflösen zu können, wird das folgende Lemma herangezogen.

Lemma 6.4.3 *Für $n \in \mathbb{N}, n \geq 1$ gilt:*

$$\sum_{k=1}^{n} H_{2k} = \left(n + \frac{1}{2}\right) H_{2n} + \frac{1}{4} H_n - n.$$

Beweis. Für $n = 1$ erhält man $(1+\frac{1}{2})H_2+\frac{1}{4}-1 = \left(\frac{3}{2}\right)^2-\frac{3}{4} = \frac{3}{2}$, was sich mit der linken Seite $H_2 = \frac{3}{2}$ deckt. Induktiv soll die Aussage nun für alle $n > 1$ gezeigt werden. Als Induktionsannahme darf von der Richtigkeit der Gleichung für n ausgegangen werden. Im Induktionsschritt wird nun auch die Gültigkeit für $n + 1$ gezeigt:

$$\begin{aligned}
&\sum_{k=1}^{n+1} H_{2k} = \sum_{k=1}^{n} H_{2k} + H_{2n+2} = \left(n + \frac{1}{2}\right) H_{2n} + \frac{1}{4} H_n - n + H_{2n+2} \\
&= \left(n + \frac{1}{2}\right)\left(H_{2n+2} - \frac{1}{2n+2} - \frac{1}{2n+1}\right) + \frac{1}{4}\left(H_{n+1} - \frac{1}{n+1}\right) - n + H_{2n+2} \\
&= \left(n + \frac{3}{2}\right) H_{2n+2} - \left(n + \frac{1}{2}\right)\left(\frac{1}{2n+2} + \frac{1}{2n+1}\right) + \frac{1}{4} H_{n+1} - \frac{1}{4n+4} - n \\
&= \left(n + \frac{3}{2}\right) H_{2n+2} - \frac{2n+1}{2} \cdot \frac{4n+3}{(2n+2)(2n+1)} - \frac{1}{4n+4} + \frac{1}{4} H_{n+1} - n \\
&= \left(n + \frac{3}{2}\right) H_{2n+2} - \frac{4n+3}{4n+4} - \frac{1}{4n+4} + \frac{1}{4} H_{n+1} - n \\
&= \left((n+1) + \frac{1}{2}\right) H_{2(n+1)} + \frac{1}{4} H_{n+1} - (n+1).
\end{aligned}$$

□

Mit diesem Lemma kann man nun nach der Indexverschiebung

$$\sum_{\ell=0}^{\lfloor \frac{n-3}{2} \rfloor} \left(\frac{n}{2(\ell+1)} - 1 - H_n + H_{2(\ell+1)}\right) = \sum_{\ell=1}^{\lfloor \frac{n-1}{2} \rfloor} \left(\frac{n}{2\ell} - 1 - H_n + H_{2\ell}\right)$$

folgende geschlossene Formel finden:

$$\left(\left\lfloor \frac{n-1}{2} \right\rfloor + \frac{1}{2}\right) H_{2\lfloor \frac{n-1}{2} \rfloor} + \left(\frac{n}{2} + \frac{1}{4}\right) H_{\lfloor \frac{n-1}{2} \rfloor} - \left\lfloor \frac{n-1}{2} \right\rfloor H_n - 2 \left\lfloor \frac{n-1}{2} \right\rfloor.$$

Da man jetzt beide Teile aus der Darstellung (6.2) berechnet hat, kann man diese zusammenfassen und erhält daher den gesuchten Erwartungswert:

$$\begin{aligned}
\mathbb{E}\left(\kappa^{[FB2]}\right) = &\left(\left\lfloor \frac{n-1}{2} \right\rfloor + \frac{1}{2}\right) H_{2\lfloor \frac{n-1}{2} \rfloor} + \left(\frac{n}{2} + \frac{1}{4}\right) H_{\lfloor \frac{n-1}{2} \rfloor} - \left\lfloor \frac{n-1}{2} \right\rfloor H_n \\
&- 2 \left\lfloor \frac{n-1}{2} \right\rfloor + n.
\end{aligned}$$

Diese Formel soll nun mit Hilfe einer Fallunterscheidung umgeformt werden. Dazu wird zuerst angenommen, dass n gerade ist. Somit gilt also $\left\lfloor \frac{n-1}{2} \right\rfloor = \left\lfloor \frac{n-2}{2} \right\rfloor = \frac{n-2}{2} = \frac{n}{2} - 1$. Dies führt zu:

$$\begin{aligned}
\mathbb{E}\left(\kappa^{[FB2]}\right) &= \left(\frac{n-2}{2} + \frac{1}{2}\right) H_{n-2} + \left(\frac{n}{2} + \frac{1}{4}\right) H_{\frac{n}{2}-1} - \frac{n-2}{2} H_n - 2 \cdot \frac{n-2}{2} + n \\
&= \frac{n-1}{2}\left(H_n - \frac{1}{n} - \frac{1}{n-1}\right) + \left(\frac{n}{2} + \frac{1}{4}\right)\left(H_{\frac{n}{2}} - \frac{1}{\frac{n}{2}}\right) - \frac{n-2}{2} H_n + 2 \\
&= \left(\frac{n-1}{2} - \frac{n-2}{2}\right) H_n + \left(\frac{n}{2} + \frac{1}{4}\right) H_{\frac{n}{2}} - \left(\frac{n-1}{2n} + \frac{1}{2}\right) - \left(1 + \frac{1}{2n}\right) + 2 \\
&= \frac{1}{2} H_n + \left(\frac{n}{2} + \frac{1}{4}\right) H_{\frac{n}{2}} - \frac{2n-1}{2n} - \frac{2n+1}{2n} + 2 \\
&= \frac{1}{2} H_n + \left(\frac{n}{2} + \frac{1}{4}\right) H_{\frac{n}{2}}.
\end{aligned}$$

Als nächstes soll der Fall, dass n ungerade ist, untersucht werden. Hier gilt $\left\lfloor \frac{n-1}{2} \right\rfloor = \frac{n-1}{2}$ und man erhält:

$$\begin{aligned}
\mathbb{E}\left(\kappa^{[FB2]}\right) &= \left(\frac{n-1}{2} + \frac{1}{2}\right) H_{n-1} + \left(\frac{n}{2} + \frac{1}{4}\right) H_{\frac{n-1}{2}} - \frac{n-1}{2} H_n - 2 \cdot \frac{n-1}{2} + n \\
&= \frac{n}{2}\left(H_n - \frac{1}{n}\right) + \left(\frac{n}{2} + \frac{1}{4}\right) H_{\frac{n-1}{2}} - \frac{n}{2} H_n + \frac{1}{2} H_n + 1 \\
&= -\frac{1}{2} + \left(\frac{n}{2} + \frac{1}{4}\right) H_{\frac{n-1}{2}} + \frac{1}{2} H_n + 1 = \frac{1}{2} H_n + \left(\frac{n}{2} + \frac{1}{4}\right) H_{\frac{n-1}{2}} + \frac{1}{2} \\
&= \frac{1}{2} H_n + \left(\frac{n-1}{2} + \frac{1}{4}\right) H_{\frac{n-1}{2}} + \frac{1}{2}\left(H_{\frac{n-1}{2}} + 1\right).
\end{aligned}$$

Da $\frac{n-1}{2} = \left\lfloor \frac{n}{2} \right\rfloor$ bei einem ungeraden n gilt, erkennt man, dass in beiden Fällen der Term

$$\frac{1}{2} H_n + \left(\left\lfloor \frac{n}{2} \right\rfloor + \frac{1}{4}\right) H_{\left\lfloor \frac{n}{2} \right\rfloor}$$

vorkommt. Ist n ungerade, kommt noch der letzte Summand hinzu. Daher wird die Konstruktion

$$\left\lceil \frac{n}{2} \right\rceil - \left\lfloor \frac{n}{2} \right\rfloor = \begin{cases} 0, & \text{wenn } n \text{ gerade ist,} \\ 1, & \text{wenn } n \text{ ungerade ist,} \end{cases}$$

als zusätzlicher Faktor hinzugefügt und man erhält in Hinblick auf die vorangehenden Überlegungen somit folgende Formel für den Erwartungswert von $\kappa^{[FB2]}$:

$$\mathbb{E}\left(\kappa^{[FB2]}\right) = \left(\left\lfloor \frac{n}{2} \right\rfloor + \frac{1}{4}\right) H_{\left\lfloor \frac{n}{2} \right\rfloor} + \frac{1}{2} H_n + \frac{1}{2}\left(\left\lceil \frac{n}{2} \right\rceil - \left\lfloor \frac{n}{2} \right\rfloor\right)\left(H_{\left\lfloor \frac{n}{2} \right\rfloor} + 1\right).$$

Asymptotisch verhält sich der Parameter $\kappa^{[FB2]}$ daher folgendermaßen:

$$\mathbb{E}\left(\kappa^{[FB2]}\right) = \frac{1}{2} n \log n + O(n).$$

Die gleiche Rechnung kann auch für den Parameter $a^{[FB2]}$ duchgeführt werden. Startet man also mit

$$\begin{aligned} a^{[FB2]} = \Bigg| \Bigg\{ (i,j): \quad 1 \leq i < j \leq n \quad \wedge \quad \Big[q(i) < q(i+1),\, ..., q(i) < q(j) \Big] \quad \textbf{und} \\ \Big[\Big(j - i + 2 \leq \min_{k<i} \big\{ k: \; q(k) < q(i) \;\; \textbf{und} \;\; q(1) > q(i), ..., q(k-1) > q(i) \big\} \Big) \\ \textbf{oder} \quad \Big(q(1) > q(i), ..., q(i-1) > q(i) \Big) \Big] \Bigg\} \Bigg|, \end{aligned} \tag{6.3}$$

so erhält man schließlich (siehe [12]):

$$\mathbb{E}\left(a^{[FB2]}\right) = \frac{2n+1}{4} H_{\lfloor \frac{n+1}{2} \rfloor} + \frac{1}{2} H_{2\lfloor \frac{n+1}{2} \rfloor} - \frac{1}{2} \left(\left\lfloor \frac{n+1}{2} \right\rfloor - \left\lfloor \frac{n}{2} \right\rfloor \right) - \frac{n+1}{2}.$$

Abschließend soll noch der Erwartungswert des κ-Werts in Keller's ursprünglicher Variante berechnet werden. Für diesen Parameter soll der Ausdruck $\tilde{\kappa}^{[FB2]}$ verwendet werden. Wieder startet man mit der kanonischen Zyklendarstellung $(q(1), ..., q(n))$ und erhält für $\tilde{\kappa}^{[FB2]}$:

$$\begin{aligned} \tilde{\kappa}^{[FB2]} = \Bigg| \Bigg\{ (i,j): \quad 1 \leq i < j \leq n \quad \wedge \quad \Big[q(i) < q(i+1),\, ..., q(i) < q(j) \Big] \quad \textbf{und} \\ \Big[\Big(j - i + 1 \leq \min_{k<i} \big\{ k: \; q(k) < q(i) \;\; \textbf{und} \;\; q(1) > q(i), ..., q(k-1) > q(i) \big\} \Big) \\ \textbf{oder} \quad \Big(q(1) > q(i), ..., q(i-1) > q(i) \Big) \Big] \Bigg\} \Bigg| + \Bigg| \Big\{ i : q(i) \leq q(1) \Big\} \Bigg|. \end{aligned} \tag{6.4}$$

Im Gegensatz zu der Darstellung von $\kappa^{[FB2]}$ muss hier $i \neq j$ gelten, da die Schleife in Algorithmus 6.2.1 (Zeilen 14 bis 21) sofort verlassen wird, wenn $i = j$ gilt. Die Mächtigkeit der Menge $\{i : q(i) \leq q(1)\}$ als Zusatz zum Parameterwert erklärt sich aus der zusätzlichen einmaligen Ausführung von π bei der Überprüfung eines jeden Wertes (wie bereits im Vergleich von Algorithmus 6.2.1 und 6.3.1 untersucht wurde, siehe auch Beispiel 6.4.2) bis die Abbruchbedingung der Heuristik "Fastbreak 1" das Programm vorzeitig beendet. Dies geschieht nach dem Auffinden und Umspeichern des letzten Zyklenführers $q(1)$.

Aus Gleichung (6.4) folgt somit für den Erwartungswert die Darstellung:

$$\mathbb{E}\left(\kappa^{[FB2]}\right) = \sum_{1\le i<j\le n} \frac{1}{n!} \sum_{L=1}^{n} \left[\binom{n-L}{j-1}(j-1)!(n-j)! + \sum_{k=j-i+1}^{i-1} \binom{n-L}{j-i+k-1}(j-i+k-1)!(L-1)(n-k-(j-i)-1)! \right] + \frac{1}{2}(n-1)+1. \tag{6.5}$$

Die letzte Zeile stellt die erwartete Anzahl der $n-1$ übrigen Elemente dar, die kleiner als $q(1)$ sind. Bei einer Gleichverteilung ist die Wahrscheinlichkeit, dass ein $q(i)$ mit $i \neq 1$ kleiner als $q(1)$ ist, gleich der (Gegen-)Wahrscheinlichkeit, dass $q(i) > q(1)$ gilt, also $\frac{1}{2}$. Nachdem in der Bedingung $q(i) \le q(1)$ auch Gleichheit erlaubt ist, erfüllt $i = 1$ diese Bedingung und fügt den letzten Summanden 1 der erwarteten Mächtigkeit der Menge $\{i : q(i) \le q(1)\}$ hinzu.

Nun sollen wieder geschlossene Formeln für die beiden auftretenden Summen gefunden werden. Es ergibt sich also für den ersten Term:

$$\sum_{i=1}^{n}\sum_{j=i+1}^{n}\frac{1}{n!}\sum_{L=1}^{n}\binom{n-L}{j-1}(j-1)!(n-j)! = \frac{1}{n}\sum_{i=1}^{n}\sum_{j=i+1}^{n}\frac{1}{\binom{n-1}{j-1}}\sum_{L=1}^{n}\binom{n-L}{j-1}$$
$$= \frac{1}{n}\sum_{i=1}^{n}\sum_{j=i+1}^{n}\frac{\binom{n}{j}}{\binom{n-1}{j-1}} = \sum_{i=1}^{n}\sum_{j=i+1}^{n}\frac{1}{j} = \sum_{j=2}^{n}\frac{1}{j}\sum_{i=1}^{j-1}1 = \sum_{j=2(1)}^{n}\frac{j-1}{j} = \sum_{j=1}^{n}1-\frac{1}{j} = n - H_n.$$

Man erkennt, dass die leicht unterschiedlichen Grenzen zu der Berechnung für den Parameter $\kappa^{[FB2]}$ erst am Ende zum Tragen kommen. Daher wird die folgende Berechnung für die zweite Zeile in Gleichung (6.5) gleich ausgeführt wie jene für $\kappa^{[FB2]}$ und man erhält sofort von der schon zuvor ausgeführten Rechnung die folgende erste Umformung (siehe zweite Zeile in nachfolgender Berechnung), in der lediglich die Grenzen angepasst wurden. Es kann nun analog mit Rücksicht auf die veränderten Grenzen forgefahren werden:

$$\sum_{i=1}^{n}\sum_{j=i+1}^{n}\frac{1}{n!}\sum_{L=1}^{n}\sum_{k=j-i+1}^{i-1}\binom{n-L}{j-i+k-1}(j-i+k-1)!(L-1)(n-k-(j-i)-1)!$$
$$= \sum_{i=1}^{n}\sum_{j=i+1}^{n}\sum_{k=j-i+1}^{i-1}\left(\frac{n}{(j-i+k)(n-(j-i)-k)} - \frac{n+1}{(j-i+k+1)(n-(j-i)-k)}\right)$$
$$= \sum_{i=1}^{n}\sum_{\ell=1}^{n-i}\sum_{k=\ell+1}^{i-1}\left(\frac{1}{\ell+k}-\frac{1}{\ell+k+1}\right) = \sum_{i=2}^{n}\sum_{\ell=1}^{\min(i-2,n-i)}\sum_{k=\ell+1}^{i-1}\left(\frac{1}{\ell+k}-\frac{1}{\ell+k+1}\right)$$
$$= \sum_{i=2}^{n}\sum_{\ell=1}^{\min(i-2,n-i)}\left(\frac{1}{2\ell+1}-\frac{1}{\ell+i}\right) = \sum_{\ell=1}^{\lfloor\frac{n-2}{2}\rfloor}\sum_{i=\ell+2}^{n-\ell}\left(\frac{1}{2\ell+1}-\frac{1}{\ell+i}\right)$$
$$= \sum_{\ell=1}^{\lfloor\frac{n-2}{2}\rfloor}\left(\frac{n-2\ell-1}{2\ell+1} - H_n + H_{2\ell+1}\right) = \sum_{\ell=1}^{\lfloor\frac{n-2}{2}\rfloor}\left(\frac{n}{2\ell+1} - 1 - H_n + H_{2\ell+1}\right).$$

Für die letzte Summe wird das folgende Lemma benötigt.

Lemma 6.4.4 *Für $n \in \mathbb{N}$ gilt:*

$$\sum_{k=0}^{n} H_{2k+1} = \left(n + \frac{3}{2}\right) H_{2n+1} - n - \frac{1}{2} - \frac{1}{4} H_n.$$

Beweis. Für $n = 0$ erhält man $(0+\frac{3}{2})H_1 - 0 - \frac{1}{2} - \frac{1}{4}H_0 = \frac{3}{2} - \frac{1}{2} = 1$, was sich mit der linken Seite $H_1 = 1$ deckt. Induktiv soll die Aussage nun für alle $n \geq 1$ gezeigt werden. Als Induktionsannahme darf von der Richtigkeit der Gleichung für n ausgegangen werden. Im Induktionsschritt wird nun auch die Gültigkeit für $n+1$ gezeigt:

$$\begin{aligned}
&\sum_{k=1}^{n+1} H_{2k+1} = \sum_{k=1}^{n} H_{2k+1} + H_{2n+3} = \left(n + \frac{3}{2}\right) H_{2n+1} - n - \frac{1}{2} - \frac{1}{4} H_n + H_{2n+3} \\
&= \left(n + \frac{3}{2}\right) \left(H_{2n+3} - \frac{1}{2n+3} - \frac{1}{2n+2}\right) - n - \frac{1}{2} - \frac{1}{4}\left(H_{n+1} - \frac{1}{n+1}\right) + H_{2n+3} \\
&= \left(n + \frac{5}{2}\right) H_{2n+2} - n - \frac{1}{2} - \left(n + \frac{3}{2}\right)\left(\frac{1}{2n+3} + \frac{1}{2n+2}\right) - \frac{1}{4} H_{n+1} + \frac{1}{4n+4} \\
&= \left(n + \frac{5}{2}\right) H_{2n+2} - n - \frac{1}{2} - \frac{2n+3}{2} \cdot \frac{4n+5}{(2n+3)(2n+2)} + \frac{1}{4n+4} - \frac{1}{4} H_{n+1} \\
&= \left(n + \frac{5}{2}\right) H_{2n+2} - n - \frac{1}{2} - \frac{4n+5}{4n+4} + \frac{1}{4n+4} - \frac{1}{4} H_{n+1} \\
&= \left((n+1) + \frac{3}{2}\right) H_{2(n+1)} - (n+1) - \frac{1}{2} - \frac{1}{4} H_{n+1}.
\end{aligned}$$

□

Mit Lemma 6.4.4 erhält man unter Berücksichtigung von $\sum_{k=1}^{n} H_{2k+1} = \sum_{k=0}^{n} H_{2k+1} - \underbrace{H_1}_{=1}$:

$$\begin{aligned}
&\sum_{\ell=1}^{\lfloor \frac{n-2}{2} \rfloor} \left(\frac{n}{2\ell+1} - 1 - H_n + H_{2\ell+1}\right) \\
&= n\left(H_{2\lfloor \frac{n-2}{2} \rfloor + 1} - \frac{3}{2}\right) - \left\lfloor \frac{n-2}{2} \right\rfloor - \left\lfloor \frac{n-2}{2} \right\rfloor H_n \\
&\quad + \left(\left\lfloor \frac{n-2}{2} \right\rfloor + \frac{3}{2}\right) H_{2\lfloor \frac{n-2}{2} \rfloor + 1} - \left\lfloor \frac{n-2}{2} \right\rfloor - \frac{1}{2} - \frac{1}{4} H_{\lfloor \frac{n-2}{2} \rfloor} - 1 \\
&= \left(n + \left\lfloor \frac{n}{2} \right\rfloor + \frac{1}{2}\right) H_{2\lfloor \frac{n}{2} \rfloor - 1} - \frac{1}{4} H_{\lfloor \frac{n}{2} \rfloor - 1} - \left(\left\lfloor \frac{n}{2} \right\rfloor - 1\right) H_n - 2\left\lfloor \frac{n}{2} \right\rfloor - \frac{3n}{2} + \frac{1}{2}.
\end{aligned}$$

Somit ist Darstellung (6.5) aufgelöst und man erhält:

$$\begin{aligned}
\mathbb{E}\left(\tilde{\kappa}^{[FB2]}\right) &= \left(n + \left\lfloor \frac{n}{2} \right\rfloor + \frac{1}{2}\right) H_{2\lfloor \frac{n}{2} \rfloor - 1} - \frac{1}{4} H_{\lfloor \frac{n}{2} \rfloor - 1} - \left(\left\lfloor \frac{n}{2} \right\rfloor - 1\right) H_n - 2\left\lfloor \frac{n}{2} \right\rfloor - \frac{3n}{2} + \frac{1}{2} \\
&\quad + n - H_n + \frac{1}{2}(n-1) + 1 \\
&= \left(n + \left\lfloor \frac{n}{2} \right\rfloor + \frac{1}{2}\right) H_{2\lfloor \frac{n}{2} \rfloor - 1} - \left\lfloor \frac{n}{2} \right\rfloor H_n - \frac{1}{4} H_{\lfloor \frac{n}{2} \rfloor - 1} - 2\left\lfloor \frac{n}{2} \right\rfloor + 1.
\end{aligned}$$

Asymptotisch verhält sich der Erwartungswert des Parameters $\tilde{\kappa}^{[FB2]}$ also wie jener von $\kappa^{[FB2]}$:

$$\mathbb{E}\left(\tilde{\kappa}^{[FB2]}\right) = \frac{1}{2} n \log n + O(n).$$

Damit ist der theoretische Teil dieser Arbeit abgeschlossen. Im folgenden Kapitel werden nun die wichtigsten Resultate gesammelt, um sie anschließend experimentell zu verifizieren.

Kapitel 7

Experimentelles Testen

In diesem Kapitel sollen zuerst die Ergebnisse der Arbeit zusammengefasst werden. Anschließend wird auf die Implementierung der Algorithmen in Matlab eingegangen und eine daraus resultierende graphische Darstellung des asymptotischen Verhaltens präsentiert.

7.1 Zusammenfassung der theoretischen Ergebnisse

Die wichtigsten berechneten Größen wurden in der Arbeit von A. Panholzer, H. Prodinger und M. Riedel in einer Tabelle zusammengefasst (siehe [12, Abbildung 4]), die hier im Wesentlichen übernommen wird:

Größe	Bez.	Exakte Formel	Asympt.
Knuth's Parameter a	a	$(n+1)H_n - 2n$	$n \log n$
Keller's Parameter κ	κ	$(n+1)H_n - n$	$n \log n$
a mit Fastbreak 1	$a^{[FB1]}$	$(n+2)H_n - \frac{5n}{2} - \frac{1}{2}$	$n \log n$
κ mit Fastbreak 1	$\kappa^{[FB1]}$	$(n+2)H_n - 2n$	$n \log n$
a mit Fastbreak 2	$a^{[FB2]}$	$\frac{2n+1}{4} H_{\lfloor \frac{n+1}{2} \rfloor} + \frac{1}{2} H_{2\lfloor \frac{n+1}{2} \rfloor} - \frac{1}{2}\left(\lfloor \frac{n+1}{2} \rfloor - \lfloor \frac{n}{2} \rfloor\right) - \frac{n+1}{2}$	$\frac{1}{2} n \log n$
κ mit Fastbreak 2	$\kappa^{[FB2]}$	$\left(\lfloor \frac{n}{2} \rfloor + \frac{1}{4}\right) H_{\lfloor \frac{n}{2} \rfloor} + \frac{1}{2} H_n + \frac{1}{2}\left(\lceil \frac{n}{2} \rceil - \lfloor \frac{n}{2} \rfloor\right)\left(H_{\lfloor \frac{n}{2} \rfloor} + 1\right)$	$\frac{1}{2} n \log n$
κ mit Fastbreak 2, Keller's Implement.	$\tilde{\kappa}^{[FB2]}$	$\left(n + \lfloor \frac{n}{2} \rfloor + \frac{1}{2}\right) H_{2\lfloor \frac{n}{2} \rfloor - 1} - \lfloor \frac{n}{2} \rfloor H_n$ $-\frac{1}{4} H_{\lfloor \frac{n}{2} \rfloor - 1} - 2\lfloor \frac{n}{2} \rfloor + 1$	$\frac{1}{2} n \log n$

In obiger Tabelle ist mit "Keller's Implement." jene Implementierung der Heuristik "Fastbreak 2" gemeint, die von J. Keller in [11] entwickelt wurde (Algorithmus 6.2.1). Hier macht es keinen Sinn, den zugehörigen Parameter " $\tilde{a}^{[FB2]}$ " zu betrachten, da bei jedem Aufruf der inneren Schleife genau einmal π berechnet wird und somit $\tilde{a}^{[FB2]} = \tilde{\kappa}^{[FB2]}$ gilt. Dem wird die Implementierung der Heuristik "Fastbreak 2" von A. Panholzer, H. Prodinger und M. Riedel in [12] gegenübergestellt (Algorithmus 6.2.1), bei der auf beide Parameter a und κ eingegangen wird.

7.2 Praktische Ergebnisse

Mithilfe der Computersoftware Matlab wurden die Algorithmen GOWER (der klassische Algorithmus 3.2.1), FASTBREAK1 (Algorithmus 5.2.1), die ursprüngliche Variante von Keller's "Fastbreak 2", nämlich FASTBREAK2_V1 (Algorithmus 6.2.1) und die Überarbeitung aus [12], FASTBREAK2_V2 (Algorithmus 6.3.1), derart implementiert (siehe Anhang), dass sie mit einer übergebenen Permutation die Parameterwerte a und κ ermitteln und zurückgeben – ohne ein dazugehöriges Datenfeld umzuspeichern. Die Permutationsfunktion π ist so realisiert, dass es sich um einen Vektor $p \in \mathbb{N}^n$ handelt, sodass $(\pi(1), ..., \pi(n)) = (p(1), ..., p(n))$ gilt. Außerdem gibt es beim Algorithmus FASTBREAK2_V2 zu den Parameterwerten a und κ drei zusätzliche Rückgabewerte, nämlich die Anzahl der Zyklen, die maximale Zyklenlänge und ein Vektor mit den Zyklenlängen aller vorkommenden Zyklen. Diese Werte werden bei der experimentellen Bestimmung des worst-case der Heuristik "Fastbreak 2" benötigt.

7.2.1 Experimenteller Vergleich der Algorithmen

Um mit den eben genannten Implementierungen die Laufzeiteigenschaft vergleichen zu können, wurden im Programm START_COMPARE (siehe Anhang) für jede der zwanzig Permutationsgrößen $n = 2^5, 2^6, ..., 2^{24}$ je 100 zufällige Permutationen mit dem Matlab-internen Befehl "randperm" erzeugt. Hinter diesem Befehl steht das Festlegen von n gleichmäßig verteilten Zufallszahlen im Intervall [0,1] mithilfe des Befehls "rand" und anschließender Sortierung dieser Werte – ein Vergleich der Platznummern vor und nach dem Sortieren (der zweite Rückgabeparameter der "sort"-Funktion ist ein derartiger Vektor) liefert die Permutation (siehe Anhang). Die 100 Permutationen pro n werden nun jeweils den vorhin genannten Implementierungen übergeben, um anschließend die prozentuelle Ersparnis der Heuristiken "Fastbreak 1" bzw. "Fastbreak 2" gegenüber dem klassischen Algorithmus bezüglich der Parameter a und κ berechnen zu können. Diese 50 Prozentwerte werden auf der Ordinate über der entsprechenden Permutationsgröße n aufgetragen. Zusätzlich wird von diesen 50 Prozentwerten für jeden Wert n auch der Mittelwert berechnet, um anschließend alle diese Mittelwerte verbinden zu können. Dabei kann man (siehe Abbildung 7.1) sehr deutlich die große Ersparnis der Heuristik "Fastbreak 1" für kleine Permutationsgrößen n erkennen, während die Kurve für größere n abflacht und gegen Null strebt. Dahingegen liegt die Vermutung nahe, dass die Ersparnis der Heuristik "Fastbreak 2" für $n \to \infty$ gegen 50 Prozent strebt. Beide Beobachtungen entsprechen den theoretischen Berechnungen und spiegeln somit einen wahren Sachverhalt wider. Außerdem erkennt man, nachdem dieselben 100 zufälligen Permutationen allen Implementierungen übergeben wurden, dass der verbesserte "Fastbreak 2"-Algorithmus speziell für kleine Permutationsgrößen n weniger Aufrufe von π benötigt (was ja dem Parameterwert κ entspricht) als Keller's ursprüngliche Variante.

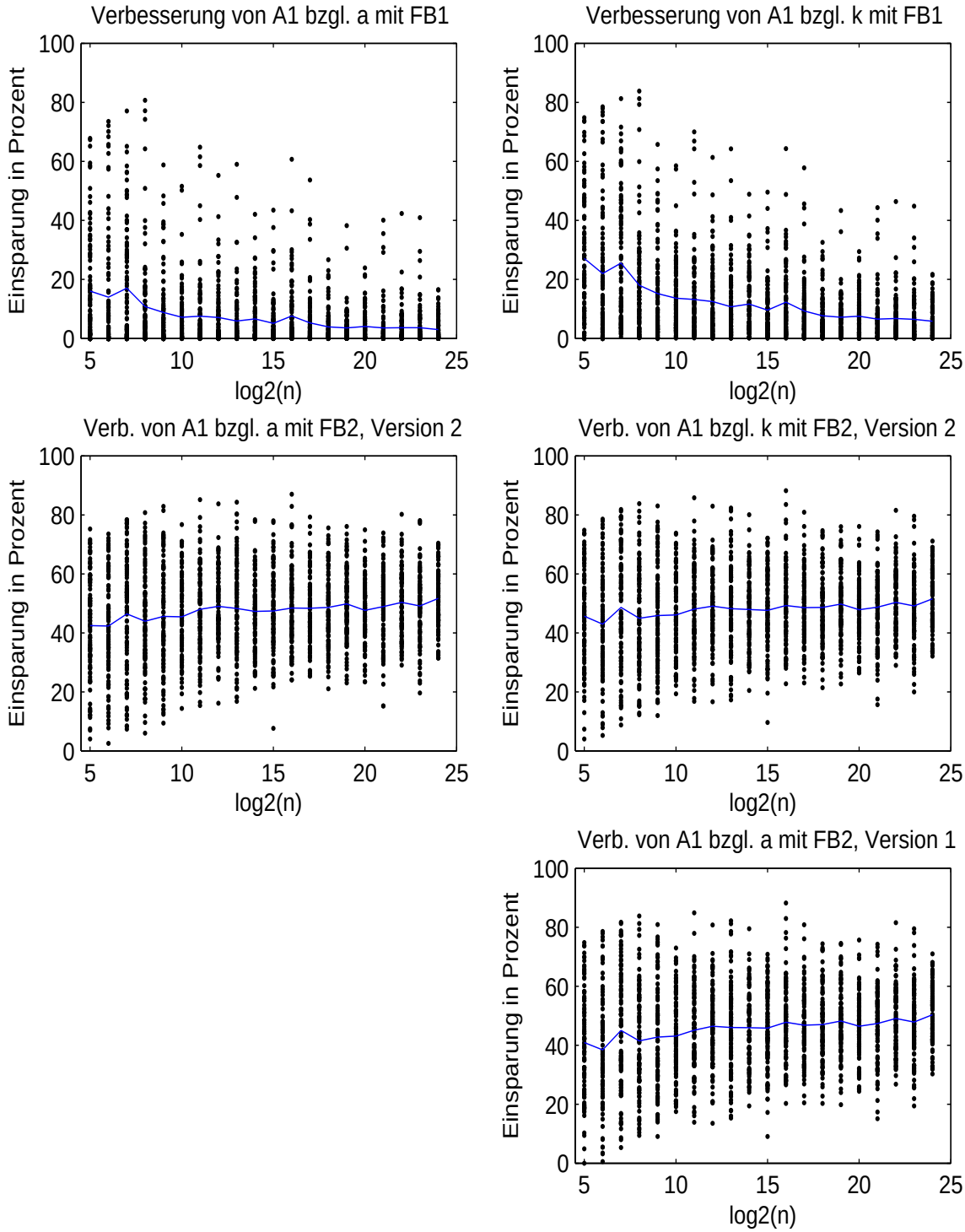


Abbildung 7.1: Vergleich der Laufzeiteigenschaften des klassischen Algorithmus (hier: A1) und der Heuristiken “Fastbreak 1” (hier: FB1) und “Fastbreak 2” (hier: FB2) bezüglich der Parameter a (linke Spalte) und κ (rechte Spalte)

7.2.2 Experimentelle Ermittlung des worst-case für "Fastbreak 2"

These 1: Struktur des worst-case Szenarios

Um experimentell den schlechtesten Fall für die Implementierung FASTBREAK2_V2 ermitteln zu können, muss man sich zunächst im Programm START_WORSTCASE alle möglichen Permutationen für eine feste Länge n mit der Matlab-Funktion "perms" berechnen lassen (siehe Anhang). Die Matrix, die alle diese möglichen Permutationen aufnimmt, hat $n!$ Zeilen und n Spalten und ist vollbesetzt (alle Einträge sind ungleich null). Aufgrund der enormen Größe dieser Matrix wurde die Experimentierreihe bis zur Permutationsgröße $n = 10$ angesetzt. Es wird für jedes n für alle $n!$ Permutationen die zugehörigen Parameter a und κ bezüglich der Funktion FASTBREAK2_V2 ermittelt. Als nächstes werden die maximalen Werte "a_max" und "k_max" für die Parameter a und κ berechnet und die dazugehörigen Indizes, wo diese Maxima angenommen werden. Damit kann man sich die Permutationen mitsamt der Zyklenanzahl und der Länge des größten Zyklus ausgegeben lassen (so kann man etwa erkennen, dass der größte Zyklus – sofern es mindestens zwei gibt – immer streng monoton steigend ist). Dies wurde zwar implementiert (siehe Anhang) aber aufgrund der Unübersichtlichkeit deaktiviert. Vielmehr wurde die These (siehe 6.4.2) überprüft, ob der worst-case bezüglich des Parameters a aus zwei Zyklen besteht und der größere die auf ganze Zahlen gerundete Länge von $n \cdot \phi$ mit

$$\phi = \frac{\sqrt{5}-1}{2} = 0{,}618...$$

hat. Dazu wurde ein Diagramm erstellt (siehe Abbildung 7.2), indem die Anzahl der worst-case Szenarien mit diesem Verhalten (blaue Balken) jenen gegenübergestellt wird, die davon abweichen (rote Balken). Die beiden Balken zusammengezählt ergeben die Anzahl der worst-case Szenarien. Die Abweichungen für $n < 5$ sind dadurch zu erklären, dass, wie schon im Unterkapitel 6.4.2 erwähnt, hier der worst-case nur einen Zyklus hat, da a maximal den Wert $n-1$ annehmen kann. Auch bei $n = 5$ gibt es viele Fälle, wo der Parameterwert a bei Permutationen maximal wird, die nur aus einem Zyklus (der Länge 5) bestehen. Bei $n = 7$ und $n = 10$ hat man im worst-case stets zwei Zyklen. Lediglich die Zyklenlänge des größeren Zyklus ist bei den Abweichungsfällen um eins größer als angenommen, also $\lceil n \cdot \phi \rceil$ statt dem richtig gerundeten Wert $\lfloor n \cdot \phi \rfloor$ (siehe These 3). Würde man also die Codezeile

```
if c == 2 && gcs == round(expected_greatest_cycle_size)
```

in der Implementierung START_WORSTCASE, Zeile 62, durch

```
if c == 2 && ( gcs == ceil(expected_greatest_cycle_size) ||
                          gcs == floor(expected_greatest_cycle_size) )
```

ersetzen, würden für $n = 6, ..., 10$ alle worst-case Szenarien das erwartete Verhalten aufweisen.

Das zweite Diagramm (siehe Abbildung 7.3) zeigt den Sachverhalt bezüglich des Parameters κ und ist analog zu interpretieren. Lediglich die Überprüfung auf zwei Zyklen wurde hier modifiziert, da hier in der Regel mehr Zyklen auftauchen, deren Längen einem bestimmten Größenverhältnis entsprechen. Der blaue Balken stellt jene Fälle dar, bei denen der größte Zyklus die Länge $\lceil n \cdot \phi \rceil$ aufweist, der zweitgrößte Zyklus die Länge $\lceil (n - \lceil n \cdot \phi \rceil) \cdot \phi \rceil$ hat, der drittgrößte Zyklus $\lceil (n - \lceil n \cdot \phi \rceil - \lceil (n - \lceil n \cdot \phi \rceil) \cdot \phi \rceil) \cdot \phi \rceil$ lang ist und so fort (siehe Unterkapitel 6.4.2).

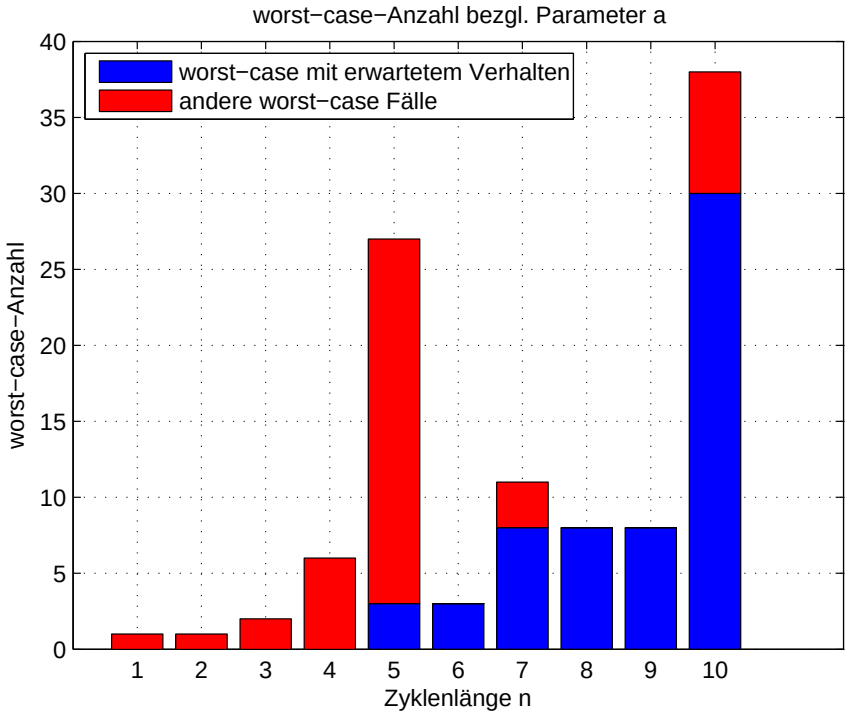


Abbildung 7.2: Vergleich der erwarteten zweizyklischen worst-case Szenarien (größerer Zyklus hat die Länge $n \cdot \phi$ auf ganze Zahlen gerundet) bzgl. des Parameters a bei der verbesserten Implementierung von "Fastbreak 2" (hier: FB2)

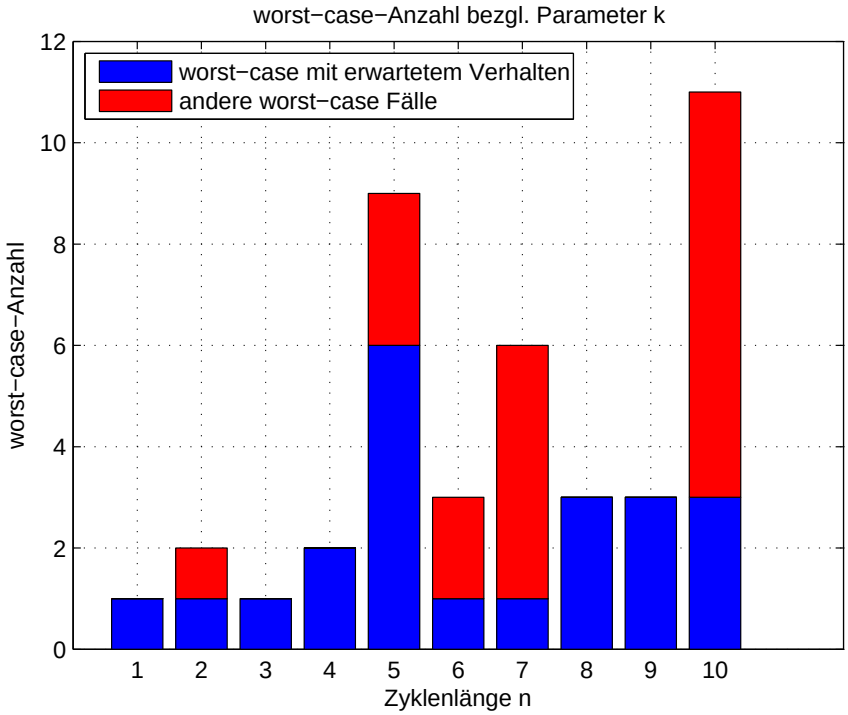


Abbildung 7.3: Vergleich der erwarteten mehrzyklischen worst-case Szenarien (größter Zyklus hat die Länge $\lceil n \cdot \phi \rceil$, zweitgrößter Zyklus hat die Länge: $\lceil (n - \lceil n \cdot \phi \rceil) \cdot \phi \rceil$, usw.) bzgl. des Parameters κ bei der verbesserten Implementierung von "Fastbreak 2" (hier: FB2)

These 2: Formel für "a_max" und "k_max"

In der Implementierung START_WORSTCASE werden auch die zwei Formeln ([17])

$$\max_{a_n} = \left(\sum_{j=1}^{n} \lfloor (1-\phi)j \rfloor \right) + 1, \quad \max_{\kappa_n} = \sum_{j=1}^{n} \lceil (1-\phi)j \rceil$$

für die Berechnung der maximalen Parameterwerte experimentell für den Fall $n \leq 10$ verifiziert (dabei bezeichnet a_n bzw. κ_n einen berechneten a- bzw. κ-Wert einer beliebigen Permutation der Größe n). Die Realisierung der Berechnungen anhand der Formeln findet man in den Code-Zeilen 105-115. Diese Berechnungen werden dann in Form der Variablen "a_max_formel" und "k_max_formel" in der Ausgabematrix, die anschließend an These 3 angeführt ist, den realen Werten "a_max" und "k_max" gegenübergestellt und die Gleichheit erkannt.

These 3: Größte Zyklenlänge im worst-case bzgl. Parameter κ

Die letzte These betrifft die größte Zyklenlänge c_n des größeren Zyklus unter allen worst-case Szenarien bzgl. Parameter κ bei einer fixierten Permutationslänge n. Diese sollte den Wert

$$\max_{c_n} = \lceil n \cdot \phi \rceil$$

haben. Im Programm FASTBREAK2_V2 wurde die Größe des größten Zyklus aller worst-case Szenarien in den Variabelen "gcs_a_max" und "gcs_k_max" berechnet und in der Ausgabematrix ausgegeben. Anschließend wurde ein Vergleichsvektor generiert, dessen n Einträge die Werte $(\phi, 2 \cdot \phi, ..., n \cdot \phi)$ sind. Man erkennt somit, dass, wenn man diese Werte auf die nächstgrößere ganze Zahl aufrundet, man die letzte Zeile der Matrix M erhält.
Folgende Matlab-Ausgabe wurde unverfälscht übernommen, lediglich die zur Erklärung dienenden Variablenbezeichnungen in der letzten Spalte wurden nachträglich hinzugefügt.

```
Ausgabematrix =

     1     2     3     4     5     6     7     8     9    10     (Index)
     0     1     2     3     4     6     8    11    14    17     (a_max)
     1     2     4     6     8    11    14    18    22    26     (k_max)
     1     1     2     3     4     6     8    11    14    17     (a_max_formel)
     1     2     4     6     8    11    14    18    22    26     (k_max_formel)
     1     2     3     4     5     4     5     5     6     7     (gcs_a_max)
     1     2     2     3     4     4     5     5     6     7     (gcs_k_max)

Vergleichsvektor =

    0.618  1.236  1.854  2.472  3.090  3.708  4.326  4.944  5.562  6.180
```

Anhang A

Matlab-Quellcode

```
function p = randperm(n)
%RANDPERM Random permutation.
%   RANDPERM(n) is a random permutation of the integers from 1 to n.
%   For example, RANDPERM(6) might be [2 4 5 6 1 3].
%
%   Note that RANDPERM calls RAND and therefore changes RAND's state.
%
%   See also PERMUTE.

%   Copyright 1984-2004 The MathWorks, Inc.
%   $Revision: 5.10.4.1 $  $Date: 2004/03/02 21:48:27 $

[ignore,p] = sort(rand(1,n));
```

```
function [ a, kappa ] = gower( p )
%GOWER Evaluation time of the classical In situ-permutationalgorithm
%(invented by J. C. Gower and analysed by D. E. Knuth)
%   gower(p), where p is a vector of length n representing a permutation,
%   computes Knuth's parameter a, which counts the runs of the inner
%   while-loop and Keller's parameter kappa, which counts the calls of the
%   permutationvektor.

a = 0;
kappa = 0;
n = size(p,2);

for j = 1:n
    k = p(j);
    kappa = kappa + 1;
    while k > j
        k = p(k);
        kappa = kappa + 1;
        a = a + 1;
    end
end
```

```
function [ a, kappa ] = fastbreak1( p )
%FASTBREAK1 Evaluation time of the heuristic Fastbreak 1 according to Keller
%   fastbreak1(p), where p is a vector of length n representing a
%   permutation, computes Knuth's parameter a, which counts the runs of the
%   inner while-loop and Keller's parameter kappa, which counts the calls
%   of the permutationvektor.
```

```
a = 0;
kappa = 0;
n = size(p,2);
remaining = n;

for j = 1:n
    k = p(j);
    kappa = kappa + 1;
    cycle_size = 1;
    while k > j
        k = p(k);
        kappa = kappa + 1;
        a = a + 1;
        cycle_size = cycle_size + 1;
    end
    if k == j
        remaining = remaining - cycle_size;
    end
    if remaining == 0
        break
    end
end
```

```
function [ a, kappa ] = fastbreak2_v1( p )
%FASTBREAK2_V1 Evaluation time of the heuristic Fastbreak 2 according to Keller
%   fastbreak2_v1(p), where p is a vector of length n representing a
%   permutation, computes Keller's parameter kappa, which counts the calls
%   of the permutationvektor. Knuth's parameter a, which counts the runs of
%   the inner while-loop, has the same value as kappa.

a = 0;
kappa = 0;
n = size(p,2);
remaining = n;

for j = 1:n
    k = j;
    leader = true;
    steps = 0;
    while 1
        k = p(k);
        kappa = kappa + 1;
        a = a + 1;
        steps = steps + 1;
        if k < j || steps > remaining
            leader = false;
            break
        end
        if k == j
            break
        end
    end

    if leader == true
        remaining = remaining - steps;
    end
    if remaining == 0
        break
    end
```

```
end
```

```
function [ a, kappa, cycle_number, greatest_cycle_size, cycle_sizes ] = fastbreak2_v2( p )
%FASTBREAK2_V2 Evaluation time of the heuristic Fastbreak 2 according to
%Panholzer, Prodinger, Riedel
%   fastbreak2_v2(p), where p is a vector of length n representing a
%   permutation, computes Knuth's parameter a, which counts the runs of the
%   inner while-loop and Keller's parameter kappa, which counts the calls
%   of the permutationvektor.

a = 0;
kappa = 0;
cycle_number = 0;
greatest_cycle_size = 0;
n = size(p,2);
remaining = n;

for j = 1:n
    leader = true;
    k = p(j);
    kappa = kappa + 1;
    steps = 1;
    while k > j && steps < remaining
        k = p(k);
        kappa = kappa + 1;
        a = a + 1;
        steps = steps + 1;
    end
    if k ~= j
        leader = false;
    end

    if leader == true
        remaining = remaining - steps;
        cycle_number = cycle_number + 1;
        if steps > greatest_cycle_size
            greatest_cycle_size = steps; %update the greatest cycle size
        end
        cycle_sizes(cycle_number) = steps; %store the cycle sizes
    end
    if remaining == 0
        break
    end
end
```

```
%START_COMPARE Visualisation of the improvement from GOWER to:
%FASTBREAK1, FASTBREAK2_V1 and FASTBREAK2_V2
%   Gernerates a graphic of the percentage of improvement according to
%   Knuth's parameter a and Keller's parameter kappa from GOWER to:
%   FASTBREAK1, FASTBREAK2_V1 and FASTBREAK2_V2
%
%   See also GOWER, FASTBREAK1, FASTBREAK2_V1 and FASTBREAK2_V2.

clear
clc

% for permutationsize up to 2^24 (starting with 2^5)
largest_exponent = 20;
%for every permutationsize handling 100 random samples
```

```
15 samples = 100;

17 for n = 1:largest_exponent

19     clear for_mean_value;
20     for_mean_value(6) = 0;

22     for j = 1:samples

24         %generating a random permutation
25         p = randperm(2^(n+4));

27         [a(n*2-1,j),k(n*2-1,j)] = gower(p);

29         [a(n*2,j),k(n*2,j)] = fastbreak1(p);

31         improvement = 100-(a(n*2,j)/a(n*2-1,j))*100;
32         for_mean_value(1) = for_mean_value(1) + improvement;

34         %Plotting one point of the left figure in the first row
35         subplot(3, 2, 1);
36         plot(n+4,improvement,'k.');
37         axis([4.5 25 0 100])
38         xlabel('log2(n)');
39         ylabel('Einsparung in Prozent');
40         title('Verbesserung von A1 bzgl. des Parameters a mit FB1')
41         hold on

43         improvement = 100-(k(n*2,j)/k(n*2-1,j))*100;
44         for_mean_value(2) = for_mean_value(2) + improvement;

46         %Plotting one point of the right figure in the first row
47         subplot(3, 2, 2);
48         plot(n+4,improvement,'k.');
49         axis([4.5 25 0 100])
50         xlabel('log2(n)');
51         ylabel('Einsparung in Prozent');
52         title('Verbesserung von A1 bzgl. des Parameters k mit FB1')
53         hold on

55         [a(n*2,j),k(n*2,j)] = fastbreak2_v2(p); %faster version

57         improvement = 100-(a(n*2,j)/a(n*2-1,j))*100;
58         for_mean_value(3) = for_mean_value(3) + improvement;

60         %Plotting one point of the left figure in the second row
61         subplot(3, 2, 3);
62         plot(n+4,improvement,'k.');
63         axis([4.5 25 0 100])
64         xlabel('log2(n)');
65         ylabel('Einsparung in Prozent');
66         title('Verbesserung von A1 bzgl. des Parameters a mit FB2, Version 2')
67         hold on

69         improvement = 100-(k(n*2,j)/k(n*2-1,j))*100;
70         for_mean_value(4) = for_mean_value(4) + improvement;

72         %Plotting one point of the right figure in the second row
73         subplot(3, 2, 4);
74         plot(n+4,improvement,'k.');
75         axis([4.5 25 0 100])
76         xlabel('log2(n)');
```

```
77          ylabel('Einsparung in Prozent');
78          title('Verbesserung von A1 bzgl. des Parameters k mit FB2, Version 2')
79          hold on

81          [a(n*2,j),k(n*2,j)] = fastbreak2_v1(p); %Keller's version

83          improvement = 100-(k(n*2,j)/k(n*2-1,j))*100;
84          for_mean_value(6) = for_mean_value(6) + improvement;

86          %Plotting one point of the left figure in the third row
87          subplot(3, 2, 6);
88          plot(n+4,improvement,'k.');
89          axis([4.5 25 0 100])
90          xlabel('log2(n)');
91          ylabel('Einsparung in Prozent');
92          title('Verbesserung von A1 bzgl. des Parameters a mit FB2, Version 1')
93          hold on

95      end

97      for j = 1:6
98          M(j,n) = for_mean_value(j)/samples; %calculating the mean
99      end

101 end

103 %the n-values
104 X(1,1:largest_exponent)= 5:largest_exponent+4;

106 %plotting a line connecting the calculated means
107 for j = 1:4
108     subplot(3, 2, j);
109     plot(X,M(j,:));
110     hold off
111 end

113 subplot(3, 2, 6);
114 plot(X,M(6,:));
115 hold off
```

```
1  function P = perms(V)
2  %PERMS  All possible permutations.
3  %   PERMS(1:N), or PERMS(V) where V is a vector of length N, creates a
4  %   matrix with N! rows and N columns containing all possible
5  %   permutations of the N elements.
6  %
7  %   This function is only practical for situations where N is less
8  %   than about 10 (for N=11, the output takes over 3 giga-bytes).
9  %
10 %   Class support for input V:
11 %      float: double, single
12 %
13 %   See also NCHOOSEK, RANDPERM, PERMUTE.

15 %   Copyright 1984-2004 The MathWorks, Inc.
16 %   $Revision: 1.12.4.1 $  $Date: 2004/07/05 17:02:07 $

18 V = V(:).'; % Make sure V is a row vector
19 n = length(V);
20 if n <= 1, P = V; return; end
```

```
q = perms(1:n-1);  % recursive calls
m = size(q,1);
P = zeros(n*m,n);
P(1:m,:) = [n * ones(m,1) q];

for i = n-1:-1:1,
    t = q;
    t(t == i) = n;
    P((n-i)*m+1:(n-i+1)*m,:) = [i*ones(m,1) t]; % assign the next m
                                                % rows in P.
end

P = V(P);
```

```
%START_WORSTCASE Computes the worst-case for the algorithm fastbreak2
%   For all possible permutations up to the size n = 10 the cases with the
%   greatest parameter a (Knuth's parameter) and k (Keller's parameter) are
%   calculated. Here the improvement of the heuristic fastbreak2 developed
%   by Panholzer, Prodinger and Riedel is used.
%
%   See also FASTBREAK2_V2.

clear
clc

gs = (sqrt(5)-1)/2; %golden ratio

for n = 1:10    %one could start with 5 to compute the interesting cases

    %give out the current permutation size n
    %n,   %activate to create output

    for j = 1:n
        p(j) = n - j + 1;
    end

    %constructing all n! permutations
    p_all = perms(p);

    %calculating parameter a
    a = zeros(1, factorial(n)); %preallocating array for speed-up
    k = zeros(1, factorial(n)); %preallocating array for speed-up
    for j = 1:factorial(n)
        [a(j),k(j)] = fastbreak2_v2(p_all(j,:));
    end

    %getting the maximum value of a and k
    a_max(n) = max(a); k_max(n)= max(k);

    %seeking all indices according to that maximum
    all_index_a_max = find(a >= a_max(n)); all_index_k_max = find(k >= k_max(n));
    %calculating the number of these indices (change ; to , for giving out)
    number_of_worstcases_a = size(all_index_a_max,2);
    number_of_worstcases_k = size(all_index_k_max,2);
    expected_worstcases_a = 0; expected_worstcases_k = 0;

    %theoretically expected biggest cycle size before rounding
    expected_greatest_cycle_size = gs * n;  %(change ; to , for giving out)
```

```
46      gcs_a_max(n) = 0;
47      for j = 1:length(all_index_a_max)

49          %permutation according to a_max
50          %p_max = p_all(all_index_a_max(j),:)   %activate to create output

52          %the number of cycles and the biggest cycle size for comparison
53          [d1, d2, c, gcs] = fastbreak2_v2(p_all(all_index_a_max(j),:));
54          %cycle_number = c, greatest_cycle_size = gcs,   %activate to create output

56          %calculating greatest cycle size of all permutations with length n
57          if gcs >= gcs_a_max(n)
58              gcs_a_max(n) = gcs;
59          end

61          %checking theoretically expected biggest cycle size and cycle number equal two
62          if c == 2 && gcs == round(expected_greatest_cycle_size)
63              expected_worstcases_a = expected_worstcases_a + 1;
64          end

66      end

68      gcs_k_max(n) = 0;
69      for j = 1:length(all_index_k_max)

71          %permutation according to k_max
72          %p_max = p_all(all_index_k_max(j),:)   %activate to create output

74          %the number of cycles and the the cycle sizes for comparison
75          [d1, d2, c, gcs, cs] = fastbreak2_v2(p_all(all_index_k_max(j),:));
76          %cycle_number = c, cycle_sizes = cs,   %activate to create output

78          expected_worstcases_k = expected_worstcases_k + 1;
79          remainder = n;
80          for h = 1:length(cs)
81              %checking expected cyclesize with real cyclesize
82              if ceil(remainder*gs) ~= cs(h)
83                  expected_worstcases_k = expected_worstcases_k - 1;
84                  break
85              end
86              remainder = remainder - ceil(remainder*gs);
87          end


90      end

92      clear p_all;
93      clear a; clear k;
94      clear all_index_a_max; clear all_index_k_max;

96      %store the number of the worst cases, with expected behavior...
97      number_of_expected_worstcases_a(n) = expected_worstcases_a;
98      number_of_expected_worstcases_k(n) = expected_worstcases_k;

100     %...and the number of other worst cases
101     number_of_other_worstcases_a(n) = number_of_worstcases_a - expected_worstcases_a;
102     number_of_other_worstcases_k(n) = number_of_worstcases_k - expected_worstcases_k;
103     %(change ; to , for giving out)

105     %calculating a_max and k_max with the unproofed formula
106     a_max_formel(n) = 0;
107     k_max_formel(n) = 0;
```

```
108     for j = 1:n
109         a_max_formel(n) = a_max_formel(n) + floor((1-gs)*j);
110         k_max_formel(n) = k_max_formel(n) + ceil((1-gs)*j);
111     end

113 end

115 a_max_formel = a_max_formel + 1;

117 %collecting and giving out calculated data
118 M = (1:n);
119 M(2,:) = a_max(:);
120 M(3,:) = k_max(:);
121 M(4,:) = a_max_formel(:);
122 M(5,:) = k_max_formel(:);
123 M(6,:) = gcs_a_max(:);
124 M(7,:) = gcs_k_max(:);
125 Ausgabematrix = M,

127 for j = 1:n
128     H(j) = gs * j;
129 end
130 Vergleichsvektor = H,    %give out compare vector

132 %plotting diagrams
133 graphic(:,1) = number_of_expected_worstcases_a(:);
134 graphic(:,2) = number_of_other_worstcases_a(:);
135 figure
136 h = bar(graphic,'stacked');
137 set(h(1),'FaceColor','b');
138 set(h(2),'FaceColor','r');
139 xlabel('Zyklenlänge n');
140 ylabel('worst-case-Anzahl');
141 title('worst-case-Anzahl bezgl. Parameter a');
142 legend('worst-case mit erwartetem Verhalten','andere worst-case Fälle');
143 legend('Location','NorthWest');
144 grid on

146 graphic(:,1) = number_of_expected_worstcases_k(:);
147 graphic(:,2) = number_of_other_worstcases_k(:);
148 figure
149 h = bar(graphic,'stacked');
150 set(h(1),'FaceColor','b');
151 set(h(2),'FaceColor','r');
152 xlabel('Zyklenlänge n');
153 ylabel('worst-case-Anzahl');
154 title('worst-case-Anzahl bezgl. Parameter k');
155 legend('worst-case mit erwartetem Verhalten','andere worst-case Fälle');
156 legend('Location','NorthWest');
157 grid on
```

Literaturverzeichnis

[1] F. E. Fich, J. I. Munro und P. V. Poblete, *Permuting in place,* SIAM Journal on Computing 24 (1995), 266-278.

[2] D. E. Knuth, *Mathematical analysis of algorithms,* Information Processing 71, North Holland Publishing Company, 1972, Proceedings of IFIP Congress, Ljubljana, 1971, 19-27. Reprinted in: *Selected papaers on analysis of algorithms,* CSLI Publications Stanford, CA, 2000.

[3] P. Kirschenhofer, H.Prodinger und R. F. Tichy, *A contribution to the analysis of in situ permutation,* Glasnik Mathematicki 22(42) (1987), 269-278.

[4] D. H. Greene und D. E. Knuth, *Mathematics for the analysis of algorithms,* 2^{nd} edition, Birkhäuser, Boston-Basel-Stuttgart, 1982.

[5] D. A. Zave, *A Series Expansion Involving the Harmonic Numbers,* Information Processing Letters 5(1) (1976), 75-77.

[6] P. Flajolet und A. Odlyzko, *Permuting in place,* SIAM Journal on Discret Mathematic 3 (1990), 216-240.

[7] D. S. Hirschberg und J. B. Sinclair, *Decentralized extrema-finding in circular configurations of processes,* Communication of the Association for Computing Machinery, 23 (1980), 627-628.

[8] D. Dolev, M. Klawe und M. Rodeth, $O(n \log n)$ *unidirectional distributed algorithm for extrema finding in a circle,* J. Algorithms, 3 (1982), 245-260.

[9] G. L. Peterson, *An* $O(n \log n)$ *Unidirectional algorithm for the circular extrema problem,* J. Algorithms, 3 (1982), 245-260.

[10] J. L. Bentley, *Programming Pearls,* Addision-Wesley Publishing Company, Reading, Mass., 1986.

[11] J. Keller, *A heuristic to accelerate in-situ permutation algorithms,* Information Processing Letters 81 (2002), 119-125.

[12] A. Panholzer, H. Prodinger und M. Riedel, *Permuting in place: analysis of two stopping rules,* J. Algorithms 51 (2004), 170–184.

[13] R. Sedgewick und P. Flajolet, *An Introduction to the Analysis of Algorithms,* Addision-Wesley Publishing Company, Reading, Mass., 1996.

[14] G. Raidl, *Algorithmen und Datenstrukturen 1,* Skriptum zur gleichnamigen Vorlesung von G. Raid, 2005.

[15] K. Felsenstein, *Einführung in die Wahrscheinlichkeitsrechnung und Statistik,* Skriptum zur gleichnamigen Vorlesung von K. Felsenstein, 2004.

[16] H.-K. Hwang und R. Neininger, *Phase change of limit laws in the quicksort recurrence under varying toll functions,* SIAM Journal on Computing 31 (2002), 1687–1722.

[17] A. Panholzer, Persönliche Konversation.

[18] A. Duschek, *Vorlesungen über höhere Mathematik. Dritter Band. Gewöhnliche und partielle Differentialgleichungen. Variationsrechnung. Funktionen einer komplexen Veränderlichen.,* Springer-Verlag, Wien, 1953.

[19] F. Harary, *Graph Theory,* Addision-Wesley Publishing Company, Reading, Mass., 1969.