Optimierungsansätze für das Quicksort-Verfahren

Zur Erlangung des akademischen Grades
Bachelor of Science
im Studiengang Informatik

Vorgelegt von:
Jonas Stübbe
Matrikelnummer: 451280
j_stue18@uni-muenster.de

Betreuer:
Prof. Dr. Jan Vahrenhold
jan.vahrenhold@wwu.de

Erstgutachter: Prof. Dr. Jan Vahrenhold, Institut für Informatik
Zweitgutachter: M.Ed. Phil Steinhorst, Institut für Informatik

Münster, 31. Oktober 2020

Vorwort

In Zeiten stetig steigender Datenmengen haben Sortierverfahren eine immer größere Bedeutung. Besonders die Optimierung dieser ist ein zentraler Aspekt der aktuellen Forschung. So wird sortiert, um Platz zu sparen, eine logische Reihenfolge zu erhalten, einfacher zu suchen oder einfach um das nächste Element zu finden, welches es zu bearbeiten gilt. Nach Ottmann et. al. [38] entfallen mehr als ein Viertel der kommerziell verbrauchten Rechenzeit auf Sortiervorgänge. Dabei kann nach mehreren Aspekten sortiert werden, z.B. nach Eigenschaft, Relevanz oder Größe. Damit war das Sortierproblem die letzten 50 Jahre eines der zentralen Gesichtspunkte der Informatik und wurde seitdem gut erforscht. Trotz allem ist Quicksort, ein im Jahre 1962 gefundener Ansatz zum Sortieren, in abgewandelter Form immer noch das meist benutzte Verfahren für den allgemeinen Gebrauch.

Ein besonderer Dank gilt den Autoren Prof. Dr. Stefan Edelkamp und Dr. Armin Weiß, sowie Dr. Martin Aumüller und Nikolaj Hass, welche die Daten ihrer Arbeit: „BlockQuicksort: Avoiding Branch Mispredictions in Quicksort“ [10] und „Simple and Fast BlockQuicksort using Lomuto's Partitioning Scheme“ [4] zur Verfügung stellten.

Inhaltsverzeichnis

1. Einleitung

Das Sortierproblem ist in der Informatik ein gut erforschtes Thema, welches stark mit unserem Alltag verbunden ist. Dabei sollen nicht nur Zahlen in eine Reihenfolge gebracht, sondern vielmehr eine Ordnung zwischen den Elementen hergestellt werden. Dies lässt sich jedoch auf den Anwendungsfall abbilden, Zahlen in eine Reihenfolge zu bringen. Dabei können die Sortierverfahren nach Ottmann et. al. [38] z.B. benutzt werden, um die Ordnung in Artikelnummern, Hausnummern, Personalnummern und Scheckkartennummern herzustellen. Dort wird dargestellt, dass das Sortierproblem von vielen Parametern abhängt: Ob Eingabe in schriftlicher Form, dem Magnetband, der Platte, Diskette oder CD-Rom. Weitere Parameter sind, ob der Eingabebereich bekannt ist, die Daten bewegt werden können und noch vieles mehr. In dieser Arbeit werden wir nur den Fall betrachten, ein Array $A[1,...,N]$ von N Elementen nach der totalen Ordnung ihrer Schlüssel zu sortieren. Dies der allgemeine Fall, für den Quicksort optimiert ist.

1.1. Motivation

Das Quicksort-Verfahren wurde 1962 von Hoare vorgestellt, seitdem wurden zahlreiche Optimierungen entwickelt. In dieser Arbeit wollen wir uns unter dem Aspekt der Sprungfehler die besten dieser Optimierungen anschauen und vergleichen. Es gibt viele Variablen von denen abhängt, welcher der vielen heute zur Verfügung stehenden Algorithmen der beste für das Anwendungsgebiet ist. Quicksort ist zusammen mit Mergesort einer der meist benutzten Algorithmen. Einer der großen Vorteile von Quicksort ist, dass nur Vergleichsoperationen benutzt werden, während andere Verfahren wie zum Beispiel Radixsort auch arithmetische Eigenschaften der zu sortierenden Schlüssel benutzen. Damit kann Quicksort für beliebige Datentypen eingesetzt werden. Ein weiterer Vorteil ist die Eigenschaft (fast) ohne zusätzlichen Speicherplatz auszukommen.

Wir wissen mittlerweile, dass das Sortierproblem eine Schranke von $N \log N$ hat[1] [40]. Einen Algorithmus zu finden, der nah an diese herankommt ist mittlerweile nicht mehr schwer. Daher fokussiert man sich auf andere Kriterien wie das Cache-Verhalten oder die Speicherkomplexität. Sprungfehler sind dabei ein neuer Aspekt, der erst seit ein paar Jahren erforscht wird und welchen es sich lohnt zu betrachten. Quicksort findet man in den meisten Standard-Bibliotheken moderner Programmiersprachen. C++ benutzt Introsort, eine Variante von Quicksort mit Absicherung gegen den schlechtesten Fall, Java benutzt ein Dual-Pivot Quicksort-Verfahren und C# benutzt Quicksort mit der *Median − Of − Three* Pivot-Wahl Strategie. Damit eine dieser Implementierungen ersetzt werden kann, muss sie in jedem Aspekt, das heißt nicht nur in der Laufzeit, sondern gegebenenfalls auch im Cache-Verhalten und der Speicherkomplexität, übertroffen werden.

[1] In dieser Arbeit ist mit log falls nicht explizit anders erwähnt $\log_2$ gemeint.

1.2. Zielsetzung

Ziel dieser Arbeit soll es sein, Sortierverfahren vorzustellen, welche die Implementierungen in Standard-Bibliotheken ersetzen können. Dafür sollen die Verfahren eine theoretische Laufzeit der gleichen Klasse vorweisen, gleich gute Speicherkomplexität erreichen und mithilfe des neuen Ansatzes der Sprungfehler eine deutliche Laufzeitverbesserung in der Praxis bewirken. Dafür sollen dem Leser vor der Vorstellung dieser Verfahren sowohl das Konzept der Sprungfehler näher gebracht werden, als auch ein tiefgründiges Verständnis für den Standard Quicksort-Algorithmus hergestellt, sowie erste Varianten und Optimierungen erarbeitet werden.

1.3. Aufbau der Arbeit

In den nächsten beiden Kapiteln werden die Grundlagen für diese Arbeit gelegt. Dafür wird in Kapitel 2 das Konzept des Pipelinings eingeführt, um dem Leser das Auftreten von Sprungfehlern deutlich zu machen und zu erklären, wie Sprungvorhersagen funktionieren. Anschließend folgt Kapitel 3: Hier wird zum ersten Mal das Sortierproblem definiert, danach werden wichtige Eigenschaften eingeführt, mit denen Sortierverfahren weiter charakterisiert werden können. Nachfolgend können wir in Kapitel 4 Quicksort einführen und die ersten Optimierungen vorstellen. Dafür werden wir zusätzlich eine neue Idee der Partitionierung einführen. In Kapitel 5 und 6 werden wir dann das erste Mal Sprungfehler freie Sortierverfahren betrachten. Dabei wird in Kapitel 5 die Idee eingeführt, Puffer mit konstanter Größe zu benutzen, mithilfe der wir die Verfahren BlockQuicksort, BlockLomuto und Dual-Pivot BlockLomuto erhalten. In Kapitel 6 wird hingegen auf den Ansatz gesetzt, viele Pivot-Elemente gleichzeitig zu benutzen. In diesem Kapitel werden wir drei Verfahren vorstellen, wobei diese jeweils aufeinander aufbauen und erhalten schließlich IPS^{4}o. Darauf folgt in Kapitel 7 ein Vergleich der vier Sprungfehler freien Sortierverfahren mit der Quicksort-Implementierung std::sort[2], sowie die Untersuchung der Verfahren für verschiedene Parameterwerte. Danach wechselt der Fokus, von der Laufzeit der verschiedenen Sortierverfahren, zu der Anwendungsbeschreibung in Kapitel 8. Dort wird das parallel zu dieser Arbeit erstellte Programm vorgestellt, sowie verschiedene Implementations und Erweiterungsaspekte dessen betrachtet und erklärt. Abschließend folgen die Kapitel Ausblick und Fazit in denen sowohl die gewonnenen Erkenntnisse zusammengefasst als auch zukünftig mögliche Entwicklungen besprochen werden.

[2] In dieser Arbeit ist mit std::sort die entsprechende GCC Implementation gemeint.

2. Einführung in Pipelining

Fast alle Prozessoren, welche heutzutage im Einsatz sind, ermöglichen es innerhalb einer *Pipeline* mehrere Instruktionen parallel auszuführen. Das gängigste Beispiel um Pipelining zu erklären ist, das Wäsche Beispiel von David Patterson [30] welches hier etwas vereinfacht dargestellt wird.

Die Ausgangssituation ist Folgende: Definiert wird ein Wäschevorgang durch die Abfolge von Waschmaschine, Trockner und Falten. Dabei braucht jeder dieser drei Abläufe jeweils 20 Minuten. Wenn man nun zwei Kleidungsstück waschen möchte, dies jedoch nicht zusammen machen kann, ist die einfachste Herangehensweise die, zwei Waschvorgänge hintereinander durchzuführen. Wie in Abbildung 2.1[1] zu sehen, wird mit dieser sequentiellen Herangehensweise eine Waschzeit von 120 Minuten erreicht. Nun wird jedoch mindestens jeder, der schon einmal eine Waschmaschine und einen Trockner bedient hat, auf eine weitere Idee kommen: Sobald die Waschmaschine fertig ist, kann man bereits einen neuen Waschvorgang starten. Damit würde die Waschmaschine parallel zum Trockner[2] und danach der Trockner parallel zum Faltvorgang laufen. In Abbildung 2.2 sieht man, dass mit der parallelen Herangehensweise in den zuvor benötigten 120 Minuten zwei weitere Waschvorgänge abgeschlossen werden können.

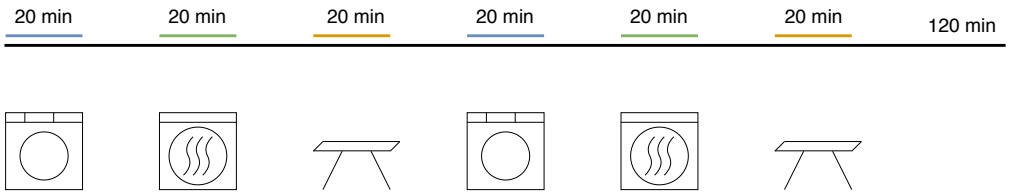


Abbildung 2.1.: Wäschebeispiel in sequentieller Ausführung

2.1. Pipeline Konflikte

Während man die sequentielle Ausführung jederzeit ohne Risiken benutzen kann, können bei der Ausführung in einer Pipeline verschiedene Konflikte auftreten. Diese sorgen für ein Stocken in der Pipeline, denn die nächste Instruktion kann nicht ohne Weiteres ausgeführt werden. Dies ist ein Leistungsabfall und sollte daher vermieden werden.

[1] Alle Abbildungen, Algorithmen und Beweise dieser Arbeit ohne Kennzeichnung eines Autors sind aus eigener Arbeit entstanden.

[2] In diesem vereinfachten Beispiel wird davon ausgegangen, dass Waschmaschine und Trockner zwei separate Maschinen sind.

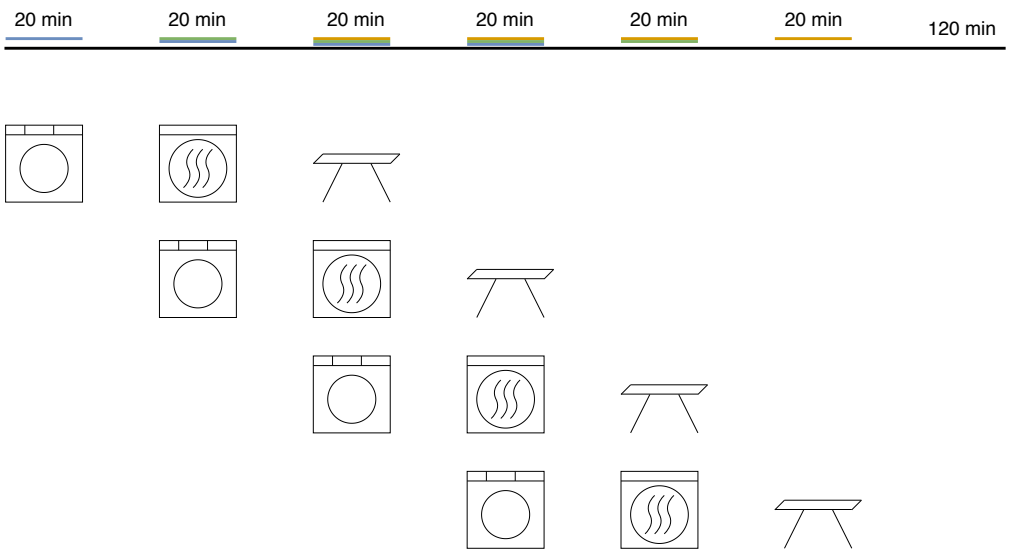


Abbildung 2.2.: Wäschebeispiel in einer Pipeline (parallele Ausführung)

2.1.1. Datenkonflikte

Datenkonflikte treten auf, wenn eine Instruktionen von einer anderen Instruktion abhängig ist. In dem vorgestellten Wäschebeispiel würde dies zum Beispiel bedeuten, dass beim Faltvorgang eine einzelne Socke vorliegt, jedoch zum Zusammenfalten das ganze Paar gebraucht wird. Die zweite Socke befindet sich noch in der Waschmaschine, es muss auf die Vollendung von Waschmaschine und Trockner gewartet werden. Prozessoren können die Wartezeit verringern, dies nennt sich *Forwarding*[3]. Beim Forwarding werden Ergebnisse mittels zusätzlicher logischer Elemente weitergegeben, ohne dass die eigentliche Pipeline abgeschlossen ist. Um wieder zum Beispiel zurückzukehren: Man könnte die Socke nach dem Waschvorgang föhnen, damit spart man 10 Minuten und kann die Socke schneller zu dem wartenden Faltvorgang weitergeben. In diesem Fall benötigt man jedoch zusätzlich einen Föhn.

2.1.2. Strukturkonflikte

Strukturkonflikte können auftreten, falls eine limitierte Ressource von zwei Instruktionen gleichzeitig gebraucht wird. Daher spricht man hier auch von einem Ressourcenkonflikt. Dieser wird meistens einfach durch Hinzufügen oder Optimierung von Hardware behoben. In dem begleitenden Beispiel würde dieser Konflikt auftreten, falls eine Kombi-Maschine für Waschen und Trocknen benutzt wird.

[3]Eine mögliche Übersetzung lautet: „Weiterleitung“

2.1.3. Kontrollkonflikte

Bei Kontrollkonflikten basiert eine Entscheidung auf dem Ergebnis einer anderen Instruktion. Das Beheben von Kontrollkonflikten ist in der Regel nicht so einfach wie bei Daten- oder Strukturkonflikten, Optimierungen wie das Forwarding sind hier nicht möglich. Im Zuge dieser Arbeit spielen Kontrollkonflikte eine zentrale Rolle und werden daher im Folgenden etwas genauer vorgestellt. Auch für diesen Konflikt gibt es wieder ein entsprechendes Beispiel in der Wäsche-Analogie. Angenommen es wurden neue Hemden gekauft, welche einzeln gewaschen werden müssen, es ist jedoch noch unklar, mit welchem Programm diese Hemden gewaschen werden sollten. Dann hängen die Wascheinstellungen des zweiten Hemdes von dem Ergebnis des ersten Hemdes nach dem Faltvorgang ab. Man muss die Pipeline anhalten und warten, ein direkter Leistungsabfall. Wie in Abbildung 2.2 zu sehen, können weitere Wäschen mit einer Erhöhung der Waschzeit von 20 Minuten hinzugefügt werden, ein neuer Waschvorgang mit einer leeren Pipeline dauert jedoch 60 Minuten. Das Füllen einer leeren Pipeline ist also sehr viel kostenintensiver als das Füllen einer bereits laufenden Pipeline. Eine Möglichkeit das Anhalten der Pipeline bei Kontrollkonflikten zu umgehen, ist die Pipeline, solange das Ergebnis unbekannt ist, mit von dem Ergebnis unabhängiger Wäsche zu füllen. Eine andere Möglichkeit ist es, das Ergebnis vorherzusagen und die Pipeline somit im Weiteren spekulativ zu füllen.

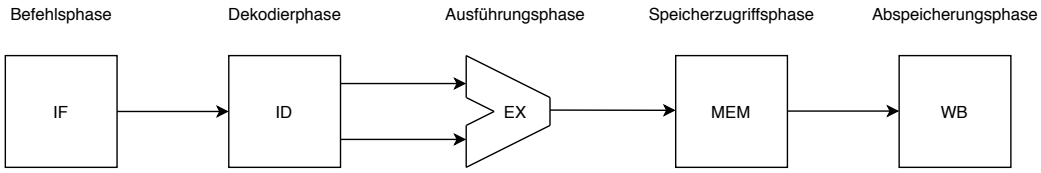


Abbildung 2.3.: Beispiel-Instruktionen einer Pipeline. Die Abbildung stammt aus der Rechnerstrukturen Vorlesung von Herber [15]
IF = Instruction Fetch, ID = Instruction decode and register file read, EX = Execution or address calculation, MEM = Data Memory Access, WB = Write Back.

In Abbildung 2.4 sehen wir ein Beispiel für einen Kontrollkonflikt. Es wird geprüft, ob die durch $t1$ und $t2$ referenzierten Werte gleich sind um den Sprung zu entscheiden. Abhängig davon werden Instruktionen 2-4 oder Instruktion 5 ausgeführt. Das Ergebnis ist jedoch erst nach der Speicherzugriffsphase *MEM* (siehe Abbildung 2.3) bekannt, bis dahin könnten drei weitere mögliche Instruktionen in der Pipeline ausgeführt werden. Für den hier benutzten Beispielprozessor mit den Instruktionen aus Abbildung 2.3 kann dies zwar auf nur eine weitere mögliche Instruktion optimiert werden, spätestens bei aktuellen Prozessoren wie dem Intel Core i7 mit 16 Pipeline Stufen, fallen dabei jedoch zu hohe Kosten an [15]. Die ersten beiden Möglichkeiten fallen daher weg: Die

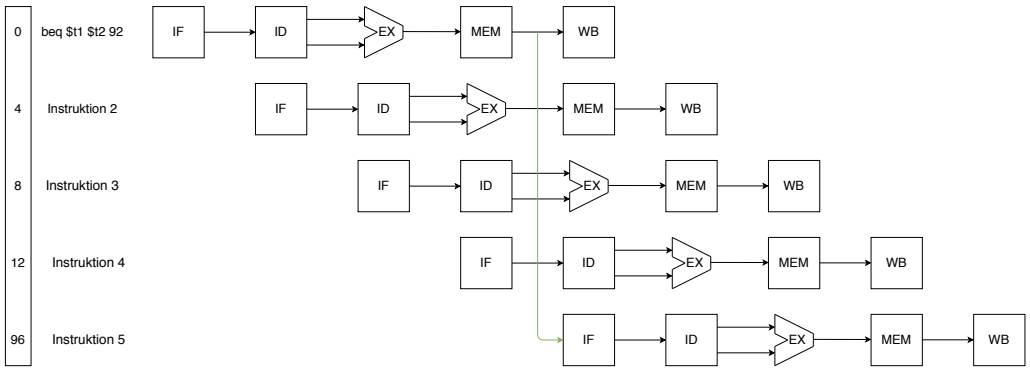


Abbildung 2.4.: Pipeline mit Entscheidungssprung. Die Abbildung wurde abgeändert der Rechnerstrukturen Vorlesung von Herber [15] entnommen.

Verschwendung beim Warten auf das Ergebnis ist zu hoch und so viele unabhängige Instruktionen zu finden meist nicht möglich. Es bleibt die Möglichkeit, wie in Abbildung 2.4 abgebildet, die Pipeline spekulativ zu füllen. Hier wird angenommen, dass der Sprung nicht durchgeführt wird. Nach der Speicherzugriffsphase MEM ist jedoch zu sehen, dass der Sprung doch durchgeführt wird. Bei solch einer Falschvorhersage muss die Pipeline kostenintensiv *gespült* werden. So nennt man das Vorgehen, wenn die Pipeline geleert und ausgeführte Instruktionen rückgängig gemacht werden.

2.2. Sprungvorhersage

Das Zitat von Mahlke et. al. „Kürzliche Studien haben gezeigt, dass mangelhafte Sprungvorhersage die Leistung um einen Faktor von zwei bis mehr als zehn verringern kann.“ [23] deckt sich mit unseren bisherigen Ergebnissen aus Unterabschnitt 2.1.3. Dort wird dargestellt, dass Fehlvorhersagen von Sprüngen sehr kostenintensiv sind, da die Pipeline gespült werden muss. Im Folgenden werden mehrere Techniken der Sprungvorhersage dargestellt, mit dem Ziel, Fehlvorhersagen zu minimieren.

Im Intel Handbuch [16] wird erläutert, dass das Konzept der Sprungvorhersage global verfügbar ist, da alle Sprünge die *Sprungvorhersageeinheit* (BPU - Branch Prediction Unit) benutzen. Die genauen Techniken, mit der ein Prozessor Vorhersagen bestimmt, sind für die meisten neueren Prozessoren nicht veröffentlicht. Im Folgenden werden jedoch die gängigsten Vorhersagemethoden vorgestellt, welche hinreichend sind um im späteren Verlauf der Arbeit auf die jeweiligen Fehlvorhersagen der vorgestellten Verfahren einzugehen. Generell können wir die Vorhersage in zwei Bereiche teilen: *statische Sprungvorhersage* und *dynamische Sprungvorhersage.*

2.2.1. Statische Sprungvorhersage

Statische Sprungvorhersagetechniken werden ihre Ergebnisse zur Laufzeit nicht ändern.Um ihre Entscheidungen zu treffen, benutzen sie die Informationen, welche

zur Übersetzungszeit bereits verfügbar sind. Damit sind sie einfacher zu benutzen und können teilweise als Backup für die dynamische Sprungvorhersage dienen. Das häufigste Beispiel ist „backward taken, forward not taken“[4] welches im Folgenden dargestellt ist.

```
1: if x < y then
2:     DoSomething
3: Continue
```

Listing 2.1.: Beispiel „forward not taken“

Sprünge, die an eine höhere Adresse springen, werden nicht gewählt, d.h. es werden immer die Instruktionen direkt nach dem Sprung bearbeitet, bis der Sprung ausgewertet wird und genommen werden soll. Im Falle von Listing 2.1 würde also immer zuerst die Instruktion *DoSomething* in die Pipeline geladen werden. Die Pipeline muss dann, falls $x \geq y$ gilt, vor dem Ausführen der Instruktion *Continue* gespühlt werden.

```
1: for int i = 0;i < 100;i + + do
2:     DoSomething
3: Continue
```

Listing 2.2.: Beispiel „backward taken“

Sprünge, die an eine niedrigere Adresse springen, werden gewählt. Schleifen werden meistens noch einmal genommen. Im Falle von Listing 2.2 würde also nach dem Ausführen der Instruktion *DoSomething* erneut die Instruktion *DoSomething* in die Pipeline geladen werden. Die Pipeline muss dann, falls $i \geq 100$ gilt, vor dem Ausführen der Instruktion *Continue* gespühlt werden. Hier wird daher bei 100 Iterationen nur in der letzten der Sprung falsch vorhergesagt.

2.2.2. Dynamische Sprungvorhersage

Im Gegensatz zur statischen Sprungvorhersage kann sich das Ergebnis bei der dynamischen Sprungvorhersage zur Laufzeit ändern, da das Ergebnis abhängig vom Verhalten des Sprunges ist. Mithilfe von moderner dynamischer Sprungvorhersage kann eine Trefferquote von ca. 90%[5][31] erreicht werden. Ein einfache Vorhersagemethode ist hier die *Sprungverlaufstabelle* (BHT - Branch History Table). Diese Methode basiert auf der Annahme, dass die Entscheidung meistens gleich bleibt. Wird ein Sprung genommen, dann wird er beim nächsten Mal sehr wahrscheinlich wieder genommen. In der Sprungverlaufstabelle wird die Entscheidung „genommen“ oder „nicht genommen“ des letzten Sprunges gespeichert und beim nächsten Mal benutzt um eine Vorhersage zu treffen und fortzufahren ohne die Pipeline anzuhalten. Nachdem das Ergebnis

[4] Eine mögliche Übersetzung lautet: „Rückwärts gewählt, vorwärts nicht gewählt.“
[5] Wir werden später sehen, dass dies beim Sortieren nicht der Fall ist.

dann ausgewertet wurde, wird es mit dem Wert der Sprungverlaufstabelle verglichen. Falls die Werte nicht übereinstimmen, wurde eine falsche Vorhersage getroffen. Die Tabelle muss aktualisiert und die Pipeline gespült werden.

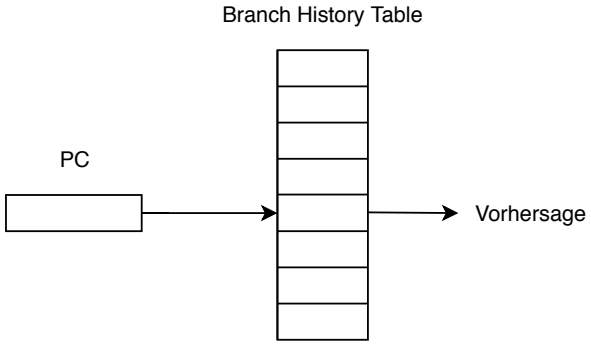


Abbildung 2.5.: Sprungverlaufstabelle (BHT - Branch History Table) mit dem Befehlszähler (PC - Program counter)

Mit der Sprungverlaufstabelle muss das Sprungziel bei jeder Sprungausführung neu berechnet werden. Da wir jedoch sowieso pro Sprung Informationen speichern müssen, kann die Tabelle optimiert werden, um auch das Sprungziel mitzuliefern. Diese Optimierung nennt sich *Sprungzielpuffer* (BTB - Branch Target Buffer) und funktioniert als eine Art Cache für Sprungvorhersage und Sprungzieladresse (vgl. Abbildung 2.6).

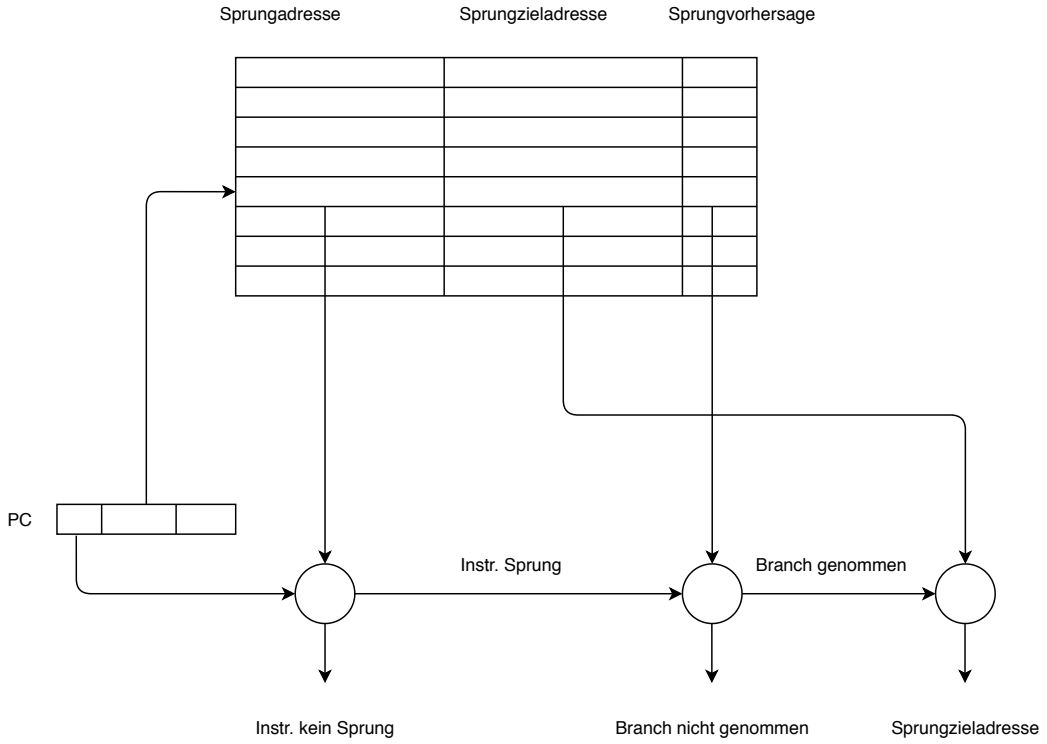


Abbildung 2.6.: Sprungzielpuffer (BTB - Brach Target Buffer)

2.2.3. 2-Bit Sprungvorhersager

Schaut man bei den Sprungvorhersagern mit Verlauf auf das Konzept „backward taken, forward not taken“, welches in Unterabschnitt 2.2.1 dargestellt wurde, so stellt man fest, dass pro Schleife immer zwei Fehlvorhersagen auftreten: eine beim Eintritt und eine beim Austritt. Beim Eintritt wird das Bit auf „genommen“ geändert und beim Austritt wird das Bit auf „nicht genommen“ geändert. Wenn das Bit auf „nicht genommen“ geändert wird, wird beim nächsten Durchgang beim Eintritt wieder eine Fehlvorhersage entsteht. Aus diesem Grund wird oft ein 2-Bit Sprungvorhersager benutzt (Abbildung 2.7). Dieser ändert das gespeicherte Bit nur, falls zwei Fehlvorhersagen aufeinander folgen. Dadurch hat die Schleife im Regelfall nur noch eine Fehlvorhersage.

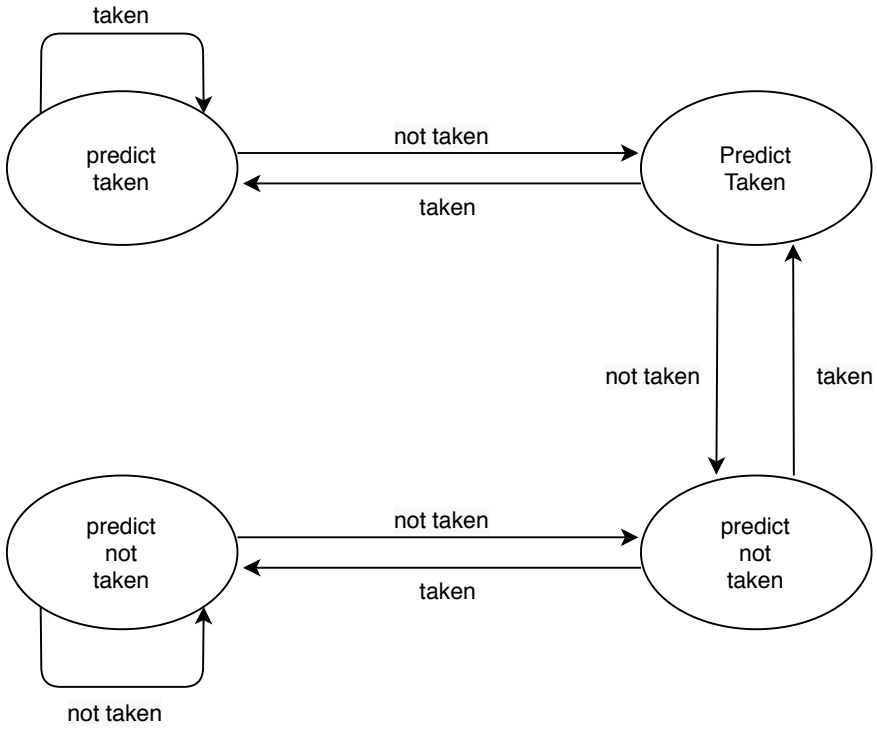


Abbildung 2.7.: 2-Bit Vorhersage

Weitere Sprungvorhersager

Weitere Sprungvorhersager sind im Zuge dieser Arbeit nur wenig relevant und werden daher nur kurz vorgestellt:

- Korrelierende Sprungvorhersager benutzen das Verhalten anderer Sprünge. Sie benutzen einen globalen und einen lokalen Sprungvorhersager und kombinieren das Ergebnis.

- Turnier Sprungvorhersager bestehen aus mehreren Sprungvorhersagern und halten für jeden Sprung fest, welcher der momentan beste Sprungvorhersager ist, dieser wird benutzt.

2.3. Optimierung der Sprungvorhersage

Mit besseren Methoden für die Sprungvorhersage und somit weniger Kosten für die Spülung langer Pipelines kann man die Leistung weiter steigern, indem das Maß an Parallelität erhöht wird. Nach Patterson et.al. [31] gilt:

Definition 2.1. (*Instruction-level parallelism*)
Der potentielle Parallelismus zwischen Instruktionen wird Instruction-level parallelism[6] genannt.

Je höher der Instruction-level parallelism ist, desto mehr Instruktionen können vom Prozessor zeitgleich (parallel) ausgeführt werden. Es ist offensichtlich, dass ein hoher IInstruction-level parallelism sowohl die Laufzeit für größere Programme enorm senken kann, als auch Sprung Vorhersage umso wichtiger macht. Im Folgenden werden daher weitere Optimierungen für die Sprungvorhersage vorgestellt.

Eine bereits angesprochene Optimierung (siehe: Listing 2.1 und Listing 2.2) ist die Anpassung des Codes an die entsprechende Sprungvorhersage. Außerdem sollten statische Sprungvorhersagen als Backup für dynamische erstellt werden, da in der ersten Iteration noch kein Verlaufswert z.B. in der Sprungverlaufstabelle enthalten ist.

2.3.1. Loop Unrolling

Beim *Loop Unrolling*[7] werden Schleifen optimiert um Iterationen in dieser zu sparen oder diese ganz zu entfernen. Dies ist eine Optimierungsmaßnahme, welche den Zusatzspeicher, wie Zählervariablen für Sprünge, verkleinert, der Pipeline weitere verfügbare Instruktionen bereitstellt und dem Compiler weitere Codeoptimierungen ermöglicht. Dabei benötigt Loop Unrolling zusätzliche Kosten falls der Code nicht mehr in den Cache passt und erhöht den benötigten Platz im Sprungzielpuffer. Eine der wichtigsten Optimierungen welche mit Loop Unrolling einhergeht wird beschreibt Patterson et. al. wie folgt: „Nach dem Loop Unrolling ist mehr Instruction-level parallelism verfügbar, in dem die Instruktionen verschiedener Iterationen überlappt werden können.“

Das Verfahren

Beim Loop Unrolling werden kleinere Schleifen aufgelöst oder (auch größere) Schleifen verkleinert.

[6] Eine mögliche Übersetzung lautet: „Parallelität auf Instruktionenebene“
[7] Eine mögliche Übersetzung lautet: „Schleifenausrollung“

```
1: for i = 1; i < 100; i++ do
2:     if i%2 = 0 then
3:         a(i) = x
4:     a(i) = y
```

Listing 2.3.: Vor dem Loop Unrolling

```
1: for i = 1; i < 100; i = i + 2 do
2:     a(i) = y
3:     a(i + 1) = x
```

Listing 2.4.: Nach dem Loop Unrolling

Die Optimierung von Listing 2.3 zu Listing 2.4 wurde dem Referenzhandbuch zur Optimierung der Intel-Architektur [16] entnommen. Mittels Zusammenfassen von jeweils zwei Schleifen Iterationen zu einer, werden 50 Schleifen Iterationen erspart.

```
1: for i = 1; i < 5; i++ do
2:     a(i) = y
```

Listing 2.5.: Vor dem Unrolling

Durch vollständiges Auflösen der Schleife werden in der Optimierung von Listing 2.5 zu Listing 2.6 insgesamt 5 Schleifen Iterationen erspart.

Richtlinien für Optimales Loop Unrolling

Bei der Optimierung mit Loop Unrolling kommt man schnell auf Ergebnisse, welche die Laufzeit erhöhen anstatt sie zu verringern. Da Loop Unrolling jedoch auch sehr effektiv sein kann, sollte man sich an folgende Richtlinien aus bereits erwähntem Referenzhandbuch [16] halten.

- Man sollte die Schleifen nicht immer vollständig auflösen, am besten bis der Zusatzspeicher für die Schleife weniger als 10% der Ausführungszeit der Schleife beträgt
- Nicht exzessiv Loop Unrolling betreiben
- Loop Unrolling vor allem bei viel benutzten Schleifen benutzen

2.3.2. Vermeidung von bedingten Sprüngen

Im Zentrum dieser Arbeit stehen die fehlerhaften Sprungvorhersagen. Diese werden in Sortieralgorithmen meistens dann hervorgerufen, wenn zwei Elemente verglichen und dann entweder vertauscht werden, oder nicht. Dieser bedingte Sprung kann vermieden werden, sodass er nicht länger Fehlvorhersagen verursacht, die zur Pipeline Spülung führen.

```
1: a(1) = y
2: a(2) = y
3: a(3) = y
4: a(4) = y
5: a(5) = y
```

Listing 2.6.: Nach dem Unrolling

Das Ziel ist, aus einem Kontrollkonflikt Datenkonflikte zu machen, welche mittels Forwarding optimiert werden können und keine Fehlvorhersagen hervorrufen. Dafür wird eine *vorherbestimmte Instruktion* verwendet, welche nach Mahlke et. al. [23] definiert wird als:

Definition 2.2. (Vorherbestimmte Instruktion)
Die *vorherbestimmte Instruktion* speichert zusätzlich einen Wahrheitswert in einem Register. Falls der Wahrheitswert im Register zu '1' ausgewertet wird, wird die Instruktion normal ausgeführt, sonst wird sie für nichtig erklärt.

Dabei ist die *bedingte Bewegung*[8] ($CMOV$) die einfachste Form einer vorherbestimmten Instruktion, die Kompilierung zu einer bedingten Bewegung auf den meisten Prozessoren ist nach Edelkamp und Weiß [10] definiert durch $i = (x < y)?j : i$. Möglich ist jedoch auch eine Umwandlung von einer Booleschen Variable zu einer Zahl ($SETcc$) welche dort definiert ist durch: int $i = (x < y)$.

2.4. Zusammenfassung

Nach der Einführung in Pipelining und falschen Sprungvorhersagen durch dieses Kapitel ist klar: Die Sprungvorhersage ist bereits weit entwickelt und bringt viel Leistung, vor allem wenn sie gut optimiert wurde, jedoch sind falsche Vorhersagen teuer. Zur Sprungvorhersage gibt es gute Alternativen wie die bedingten Sprünge, jedoch sollten solche Alternativen laut Edelkamp und Weiß [10] nur benutzt werden, um unvorhersehbare Sprünge zu vermeiden, da korrekt vorhergesagte Sprünge schneller seien.

[8] englisch: conditional move

3. Einführung in Sortieren

Im Verlaufe dieser Arbeit werden viele verschiedene Sortieransätze vorgestellt. Diese Ansätze reichen von Sortieren durch Einfügen (Unterabschnitt 3.5.1) und Sortieren mit binärem Heap (Unterabschnitt 3.5.2), über Sortieren durch rekursives Teilen (Kapitel 4), bis hin zu Sortieren mit binärem Suchbaum (Kapitel 6). Trotz verschiedener Ansätze sind die Begrifflichkeiten einer sortierten Menge oder einer Eingabesequenz immer gleich und werden im Folgenden eingeführt. Ziel dieses Kapitels ist es, ein Verständnis für die Laufzeitaspekte in Sortierverfahren zu entwickeln und wie wir diese charakterisieren können, um mit diesem Fundament in den nächsten Kapiteln Optimierungen des Quicksort-Verfahrens zu erarbeiten.

3.1. Das Sortierproblem

Jeder Sortieralgorithmus benötigt eine Eingabesequenz welche es zu sortieren gilt. Dabei besteht jede Eingabesequenz aus einer Menge von *Elementen*, wobei jedes Element einen *Schlüssel* besitzt. Bei den Sortierverfahren, welchen wir uns in Unterabschnitt 3.3.1 widmen werden, existiert zwischen den Schlüsseln eine Ordnungsrelation $<$ oder $\leq$. In dieser Arbeit wird angenommen, dass Elemente keine weiteren Daten abseits vom Schlüssel enthalten und die Schlüssel ganzzahlig sind. Dies ist im Normalfall jedoch nicht unbedingt gegeben.

Damit wir wissen, wann wir ein funktionierendes Sortierverfahren gefunden haben, führen wir das Sortierproblem nach Ottmann et. al. [38] ein:

Definition 3.1. (Das *Sortierproblem*)
Gegeben sei eine Folge von Elementen $s_1, ..., s_N$: jedes Element s_i hat einen Schlüssel k_i. Das Sortierproblem ist die Suche nach einer Permutation π der Zahlen von 1 bis N derart, dass die Umordnung der Elemente gemäß π die Schlüssel in aufsteigende Reihenfolge bringt: $k_{\pi(1)} \leq k_{\pi(2)} \leq ... \leq k_{\pi(N)}$.

Im Verlauf der Arbeit werden wir einfachheitshalber nicht zwischen Elementen und Schlüsseln unterscheiden.

3.2. Analyse von Sortierverfahren

Auch wenn wir im Verlauf der Arbeit im Normalfall von zufällig permutierten Eingaben ausgehen, wollen wir die Verfahren teilweise abhängig von der Eingabesequenz analysieren. Dafür führen wir Inversionen ein, welche angeben, wie vorsortiert die jeweilige Eingabe bereits ist. Nach Knuths „Die Kunst der Computerprogrammierung“ [20] seien Inversionen wie folgt definiert:

Definition 3.2. (*Inversionen*)
Sei $a_1 a_2 ... a_n$ eine Permutation der Menge $\{1, 2, ..., n\}$. Wenn $i < j$ und $a_i > a_j$ gilt, wird das Paar (a_i, a_j) eine Inversion der Permutation genannt.

Mit der vorherigen Definition der Inversionen können wir nach Brodal et. al. [6] adaptive Algorithmen einführen:

Definition 3.3. (*adaptive* Algorithmen)
Ein Sortieralgorithmus wird als adaptiv bezeichnet, falls die Laufzeitkomplexität sowohl von der Eingabegröße, als auch der Vorsortiertheit (Anzahl der Inversionen) abhängt.

Die Laufzeit der vorgestellten rekursiven Verfahren soll teilweise mithilfe eines Konzeptes namens: „Master-Therorem“ klassifiziert werden, welches nach der Definition von Vahrenhold [41] eingeführt wird als:

Definition 3.4. (*Master-Theorem*)
Sei $f : \mathbb{N} \to \mathbb{R}$ eine asymptotisch positive[1] Funktion, die für alle echten Potenzen einer Zahl $b \in \mathbb{N}$, $b > 1$, definiert sei, und sei $a \geq 1$. Dann gelten für die Rekursionsgleichung $T(N) = a \cdot T(\frac{N}{b}) + f(N)$
die folgenden Aussagen:

1. $Ist\, f(N) \in O(N^{\log_b a - \epsilon})$ für konstantes $\epsilon > 0$, so gilt: $T(N) \in \Theta(N^{\log_b a})$.

2. $Ist\, f(N) \in \Theta(N^{\log_b a})$, so gilt: $T(N) \in \Theta(N^{\log_b a} \log N)$.

3. $Ist\, f(n) \in \Omega(n^{\log_b(a)+\epsilon})$ für konstantes $\epsilon > 0$, und gilt $a \cdot f(N/b) \leq c \cdot f(N)$ für konstantes $c < 1$ und alle hinreichend großen Werte von N, so gilt: $T(N) \in \Theta(f(N))$.

Harmonische Zahl

Im Laufe der Arbeit wird die Approximation der n-ten Harmonischen-Zahl $\mathcal{H}_N$ benutzt, um die Laufzeit von Quicksort abzuschätzen. Die Auffassung der Harmonischen Nummer im Bezug zu den Ausgleichstermen variiert je nach Autor (vgl. mit den Definitionen von Sedgewick [45] oder Widmayer [36]); die hier verwendete Abschätzung entspringt einem Artikel des „American Math Monthly“ [39] und lautet wie folgt:

Definition 3.5. (*Harmonische-Zahl*)
$\mathcal{H}_N = 1 + \frac{1}{2} + .. + \frac{1}{N} = \ln N + \gamma + \frac{1}{2N} - \frac{1}{12N^2} + \frac{\epsilon_N}{120N^4}$ Für: $0 < \epsilon_N < 1$ und $\gamma = 0.57721...$ (Euler-Mascheroni Konstante)

3.3. Charakterisierung von Sortierverfahren

Im Folgenden werden verschiedene Eigenschaften vorgestellt, welche die Sortieralgorithmen besitzen und diese attraktiv machen können. In manchen Fällen lohnt es

[1] Funktion, die für alle bis auf endlich viele Werte positiv ist.

sich, einen Einschnitt in der Laufzeit eines Sortierverfahrens in Kauf zu nehmen, um dafür im Gegenzug diese Eigenschaften zu erhalten. Es werden außerdem mehrere Eigenschaften eingeführt, welche in der Analyse der vorgestellten Sortierverfahren nicht explizit erwähnt werden. Diese Eigenschaften sind in erster Linie dazu da, das Verständnis zu erweitern, auf welche Art und Weise Algorithmen optimiert werden können oder welche Eigenschaften sie zu erreichen vermögen.

3.3.1. Vergleichsbasierte Sortierverfahren

Als *vergleichsbasierte Sortierverfahren* bezeichnet man jene Algorithmen, welche die Relation „größer“ und „kleiner“ zwischen Elementen ausnutzen. Wir nehmen dabei an, dass für die vergleichsbasierten Sortierverfahren, welche nicht auf Sprungfehler-optimiert sind, allen Vergleichen ein Entscheidungssprung folgt, welcher auf genau diesem Vergleich basiert. Im Gegensatz dazu gibt es die Nicht-vergleichsbasierten Sortierverfahren, welche (fast[2]) nur auf numerischen Eingaben funktionieren, aber dafür eine in N lineare Laufzeit haben können. Es existiert also ein Tradeoff zwischen der Anzahl möglicher zu bearbeitender Eingabesequenzen und der Laufzeit. Im Verlaufe dieser Arbeit werden wir uns ausschließlich vergleichsbasierter Sortierverfahren widmen, da die Grundvorraussetzungen sonst bereits zu eingeschränkt für den generellen Gebrauch sind und zwei der größten Stärken von Quicksort die Flexibilität in der Eingabe und die Effizienz im allgemeinen Gebrauch sind.

Die Besonderheit von vergleichsbasierten Sortierverfahren ist, dass die Komplexität häufig durch die Anzahl der Vergleiche angegeben wird. Dies sollte im ersten Moment überraschen, da sich die Vergleiche „größer“ und „kleiner“ meistens durch Subtraktion ergeben, eine Operation, welche im Vergleich zu Speicherzugriffen zweitrangig erscheint. Wir wissen jedoch, dass die Vergleiche mit Sprüngen geknüpft sind und seit Kapitel 2 auch, dass diese teuer sind und somit die Angabe der Komplexität in der Anzahl der Vergleiche rechtfertigen. Man kann zeigen, dass die untere Schranke an Vergleichen bei $\log(N!) \approx N \log N$ liegt [9]. In Kapitel 4 werden wir sehen, dass wir mit optimierten Quicksort-Verfahren sehr nahe an diese Schranke herankommen können.

3.3.2. Zusätzlich benötigter Speicherplatz

Der zusätzlich benötigte Speicherplatz in Sortieralgorithmen spielt eine wichtige Rolle bei der Erreichbarkeit von guten Laufzeiten. Man kann sich leicht vorstellen, dass es bei größerem verfügbaren Speicherplatz einfacher wird, gute Laufzeiten zustande zu bringen. Im Laufe der Arbeit werden wir den zusätzlichen Speicher anhand folgender Definitionen nach Axtmann et. al. [5] charakterisieren:

[2] Können auch auf Eingaben angewandt werden, welche gut auf numerische Werte abgebildet werden können.

Definition 3.6. (strikt In-Place)
In der Theorie der Algorithmen arbeitet ein Algorithmus *In-Place*, falls er zusätzlich zur Eingabe nur konstant viel Speicherplatz benutzt. Wir benutzen für diesen Fall die Notation: *strikt In-Place*

Definition 3.7. (fast In-Place)
In der Theorie der Algorithmen arbeitet ein Algorithmus fast *In-Place*, falls der zur Eingabe zusätzlich benötigte Speicherplatz sublinear in der Eingabegröße liegt. Wir benutzen für diesen Fall die Notation: *In-Place*

Wenn man die Definition des in Sortieralgorithmen zusätzliche benötigten Speicherplatzes in unterschiedlichen Texten und Quellen betrachtet, wird man schnell Inkonsistenzen bei der Begrifflichkeit feststellen. Dies liegt daran, dass die Auffassungen von „In-Place“ stark auseinander gehen. Für uns soll in dieser Arbeit der Fokus auf *fast In-Place* liegen, weswegen die dafür durchaus gängige Notation *In-Place* benutzt wird. Der Fall, dass ein Verfahren nur konstant viel Speicherplatz benutzt, wird bei uns eher selten vorkommen, weswegen wir dafür weiterhin die Notation *strikt In-Place* benutzen werden.

Wie wir in Kapitel 4 sehen werden, arbeitet Quicksort nur mit einem kleinen Hilfsstack, auf dem die unerledigten Rekursionsaufrufe gespeichert werden und ist daher nach unserer Definition *In-Place* aber nicht *strikt In-Place*.

3.3.3. Weitere Eigenschaften von Sortierverfahren

In diesem Abschnitt werden drei weitere Eigenschaften betrachtet, welche durch Ottmann et. al. [38] für das Sortieren vorgestellt wurden. Diese Eigenschaften werden bei den meisten unserer Optimierungsversuche nicht im Vordergrund stehen, jedoch kann man sie im Laufe der Arbeit immer mal wieder bei Verfahren sehen.

Definition 3.8. (*interne* Sortierverfahren)
Sortierverfahren die annehmen, dass die Menge der zu sortierenden Datensätze vollständig im Hauptspeicher Platz findet, heißen *interne* Sortierverfahren.

In dieser Arbeit wird angenommen, dass alle zu sortierenden Datensätze vollständig im Hauptspeicher Platz finden. Es werden daher nur interne Sortierverfahren betrachtet.

Definition 3.9. (*stabile* Sortierverfahren)
Ein Sortierverfahren heiß *stabil*, falls die Reihenfolge von Elementen mit gleichem Sortierschlüssel während des Sortierverfahrens nicht vertauscht wird.

Das klassische Quicksort-Verfahren ist nicht stabil. Es gibt zwar stabile Varianten, jedoch benutzen die Optimierungen dieser Arbeit größtenteils Heapsort als Worst-Case Absicherung und werden dadurch sowieso nicht stabil (vgl. Unterabschnitt 3.5.2).

Definition 3.10. (*glatte* Sortierverfahren)
Ein Sortierverfahren wird *glatt* genannt, wenn es N verschiedene Schlüssel im Mittel in $O(N \log N)$ und N gleiche Schlüssel in $O(N)$ Schritten zu sortieren vermag, mit einem „glatten“ Übergang zwischen diesen Werten.

Ein Beispiel für ein solches Sortierverfahren sehen wir in Unterabschnitt 3.5.1.

3.4. Fehler bei Sprungvorhersagen in Sortierverfahren

Folgendes Zitat aus dem BlockQuicksort-Paper von Aumüller und Hass [4] beschreibt eines der Zentralen Probleme, welches in dieser Arbeit angegangen werden soll: „Wie [...] beschrieben, sind die größten Probleme beim Ausnutzen von Parallelität die falsch vorhergesagten Sprünge oder Cache Fehler“. Falls man in der angegebenen Quelle nachschaut, findet man eine Grafik eines ARM Cortex-A8 Prozessors, welcher die Zyklen pro Instruktion für verschiedene Programme angibt (Abbildung 3.1).

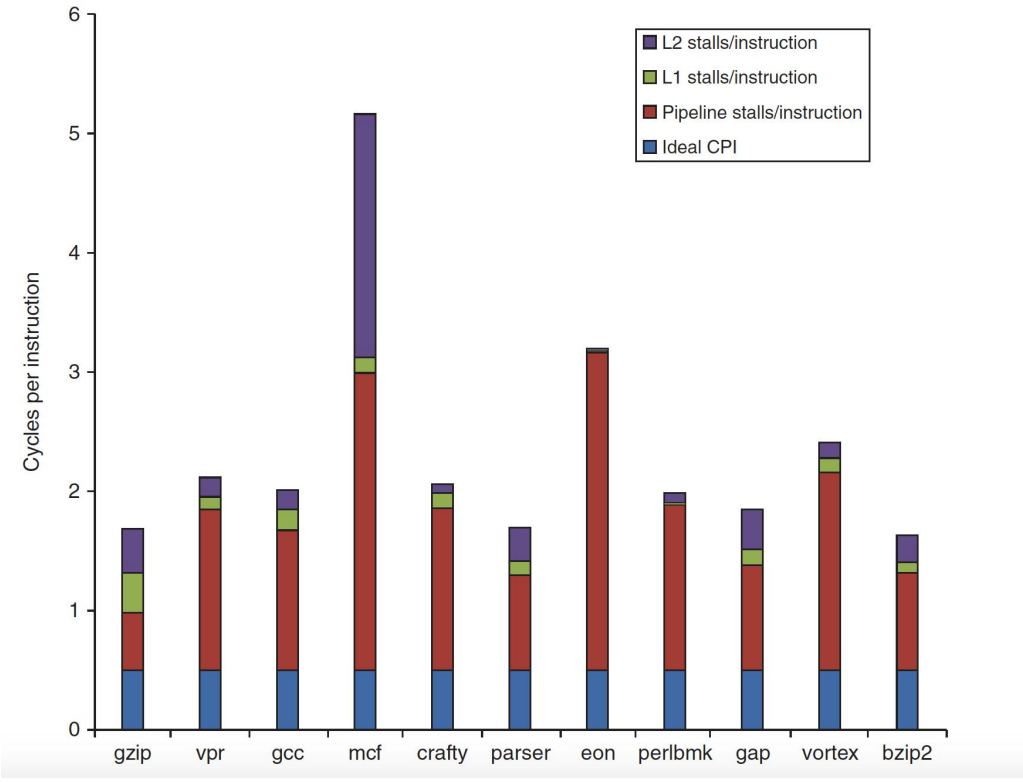


Abbildung 3.1.: Zyklen pro Instruktion für den ARM Cortex-A8 Prozessor auf verschiedenen Programmen. Die Grafik stammt aus dem Buch „Computer-Architektur“ von Hennessy et. al. [14].

In Abbildung 3.1 sieht man, dass die benötigten Zyklen pro Instruktion bei Pipeline-Verzögerungen, zu denen falsche Sprungvorhersagen gehören, deutlich ansteigen. Brodal et. al. zeigen in „Kompromisse zwischen Sprungfehlern und Vergleichen für

Sortieralgorithmen“ [6], vergleichsbasierte Sortierverfahren welche für ein $d \in \mathbb{R}$, $d \geq 1$ $O(d \cdot N \log N)$ Vergleiche benutzen, sorgen für $\Omega(N \log_d N)$ falsche Sprungvorhersagen. Wenn wir dieses Ergebnis auf die Grafik aus Abbildung 3.1 anwenden, wird klar, warum die Anzahl der Vergleiche in Sortierverfahren ein besseres Maß für die Laufzeit ist, als die Anzahl der Speicherzugriffe. Dies wird mit einer Beschreibung aus dem Buch „Computer-Architektur“ von Hennessy et. al. [14], die meisten Verluste kämen entweder von falschen Sprungvorhersagen oder Cache-Fehlern, noch einmal verdeutlicht.

3.4.1. Vermeidung falscher Sprungvorhersagen in Sortierverfahren

Im vorherigen Kapitel 2 wurden mit statischer sowie dynamischer Sprungvorhersage mehrere Verfahren vorgestellt, die Sprungvorhersage zu verbessern. Für zufällige Permutationen von Eingabesequenzen ist es bei vergleichsbasierten Sortierverfahren mit $O(N \log N)$ Vergleichen jedoch nicht möglich, vorher zu entscheiden, ob der Sprung genommen wird oder nicht, da es für diese Eingabesequenzen bei Vergleichen von Elementen nicht möglich ist vorherzusehen, ob das Element größer oder kleiner ist. Daher sind im Regelfall Fehlvorhersagen bei Sprüngen unvermeidbar.

Vorherbestimmte Instruktionen

Um also falsche Sprungvorhersagen zu vermeiden, wird im Folgenden der Ansatz vorgestellt, erst gar keine Sprungvorhersage zu treffen. Dafür wenden wir das Konzept der vorherbestimmten Instruktion aus Unterabschnitt 2.3.2 an. Dadurch wird aus einem Kontrollkonflikt ein Datenkonflikt und wir haben die Sprungvorhersage entfernt.

Im Folgenden wird der Teilbereich eines Codes dargestellt, welcher sich zum Beispiel in einer Schleife befinden könnte. $RANDOM\_NUMBER$ sei dabei eine Zufallszahl zwischen 0 und 9. Da $RANDOM\_NUMBER$ zu 50% kleiner als 5 und zu 50% größer gleich 5 ist, kann der Compiler den Sprung nicht vorhersagen, in jedem Schleifendurchlauf tritt mit 50% Wahrscheinlichkeit ein Sprungfehler auf.

```
1: if RANDOM_NUMBER < 5 then
2:     array[counter] = RANDOM_NUMBER;
3:     counter++
```

Falls der Sprung im Code-Abschnitt jedoch entfernt wird, vermeiden wir das Auftreten von Sprungfehlern, da wir die Instruktion *SETcc* benutzen, also eine Umwandlung aus einem Boolean in einen Integer durchführen. Dadurch wird aus dem Kontrollkonflikt ein Datenkonflikt gemacht. Dies ist ein zentraler Optimierungsaspekt in dieser Arbeit.

```
1: array[counter] = RANDOM_NUMBER;
2: counter+ = (RANDOM_NUMBER < 5)
```

Sprung Antizipation

Ein anderer Versuch ist, die Sprungvorhersage nach der Idee von Farcy et. al. [11] im Voraus zu berechnen. Dafür wollen wir kurz den *Sprung Fluss* einführen.

Definition 3.11. (*Sprung Fluss*)
Der Sprung Fluss ist die Reihenfolge, in der Instruktionen und Funktionsaufrufe in einem Programm durchgeführt werden.

Die Idee dabei ist Folgende: Die Instruktionen eines Entscheidungssprunges werden dupliziert und bilden den Sprung Fluss. Danach wird versucht, den Sprung Fluss vor den originalen Fluss zu bringen. Einen Fluss vor einen anderen zu bringen ist der Schlüssel dieser Technik und funktioniert nur, falls die Entscheidung des Sprunges *regulär* und *vorhersehbar* ist. Ein regulärer und vorhersehbarer Entscheidungssprung wird von Farcy et. al. [11] definiert als:

Definition 3.12. (reguläre und vorhersehbare Entscheidungssprünge)
Ein Entscheidungssprung wird als regulär und vorhersehbar klassifiziert sobald gilt:

- Der Entscheidungssprung tritt innerhalb einer lokalen Schleife auf.
- Es gilt: in der i-ten Iteration der Schleife hängen die Werte der Schleife und damit die Entscheidung linear von der Iteration $i-1$ und einer Konstanten ab.

Damit können wir diesen Ansatz als Optimierungsmethode bereits sehr schnell ausschließen, da die Vergleiche bei zufällig permutierten Eingabesequenzen nicht voneinander abhängen können. Diese Technik baut eher auf Entscheidungssprünge auf, welche lineare oder arithmetische Ausdrücke sowie Tabellendurchläufe beschreiben und ingesamt 20% aller statischen Sprünge ausmachen. Sortierverfahren können leider nicht davon profitieren.

3.4.2. Schlanke Algorithmen

Katajainen [19] definiert Schlanke[3] Algorithmen wie folgt:

Definition 3.13. (*Schlanker* Algorithmus)
Ein Algorithmus wird als schlank bezeichnet, wenn er nur eine konstante Anzahl an Entscheidungssprüngen besitzt.

Da falsche Sprungvorhersagen direkt abhängig von Entscheidungssprüngen sind, liegt die Anzahl der falschen Sprungvorhersagen für Algorithmen welche schlank sind in $O(1)$. Jeder Algorithmus kann zu einem gemacht werden, dessen Sprungfehler in $O(1)$ liegt und dessen Laufzeit linear in der ursprünglichen liegt. Jedoch entstehen dadurch sehr viele zusätzliche Kosten, wie von Katajainen [19] gezeigt wird. Außerdem sind optimierte Varianten vergleichsbasierter Sortierverfahren in der praktischen Anwendung (fast) immer schneller als die Varianten, welche schlank sind.

[3] englisch: Lean

3.5. Standard Sortierverfahren

3.5.1. Insertionsort

Insertionsort ist ein grundlegendes Sortierverfahren und basiert auf dem Ansatz Sortieren durch Einfügen. Jeder, der bereits ein Kartenspiel gespielt hat, hat wohl unbewusst schon dieses Sortierverfahren angewandt, da es dem Einsortieren neuer Handkarten beim Aufnehmen gleicht. Bekommt man eine neue Karte, so sortiert man diese in die bereits vorhandenen Karten ein: Zuerst sucht man den Platz, an den diese Karte gehört und dann steckt man die Karte dort hin.

Algorithmus 3.14 Insertionsort

```
1: function INSERTIONSORT(A[l,..,r])
2:     integer i, j, temp;
3:     for i = l,...,r do
4:         temp = A[i]
5:         j = i
6:         while (j > 0) and (A[j − 1] > temp) do
7:             A[j] = A[j-1]
8:             j = j - 1
9:         A[j] = temp
```

Ein Großteil der vorgestellten Algorithmen wird mit einer Ablaufdarstellung unterstützt. Diese stammt aus dem praktischen Teil dieser Arbeit, welcher parallel zum theoretischen entstanden ist und auf GitHub gefunden werden kann: `https://github.com/jstuebbe/QuicksortOptimization`.

Abbildung 3.2 zeigt eine Beispielsequenz in Insertionsort. Die Zeiger l und r geben die aktuellen Bereiche an. Das Verfahren ist für diese Arbeit spannend, da es eines der effizientesten vergleichsbasierten Sortierverfahren ist, falls die Eingabesequenz hinreichend klein ist. Außerdem arbeitet es *stabil* und *strikt In-Place*. Im Regelfall ist Insertionsort jedoch langsamer als Quicksort. Nach Ottmann et. al. [38] gilt für die Laufzeit: $\sum_{=1}^{N} \frac{i}{2} = \Theta(N^2)$. Dieser Algorithmus ist ein Standard-Werkzeug in der Informatik, die Korrektheit wird hier als gegeben angenommen.

3.5.2. Heapsort

Das Heapsort-Sortierverfahren basiert auf dem Ansatz Sortieren mit binärem Heap bzw. Sortieren durch Auswahl, wobei die Auswahl nicht trivial ist. Vahrenhold [42] definiert ein Max-Heap wie folgt:

Definition 3.15. (*Max-Heap*)
Ein *Max-Heap* ist ein fast vollständiger binärer Baum, in dem folgende Heap-Bedingung gilt: Der bei einem Knoten v gespeicherte Wert x ist mindestens so groß wie der Wert eines beliebigen anderen Knotens im Teilbaum mit Wurzel v.

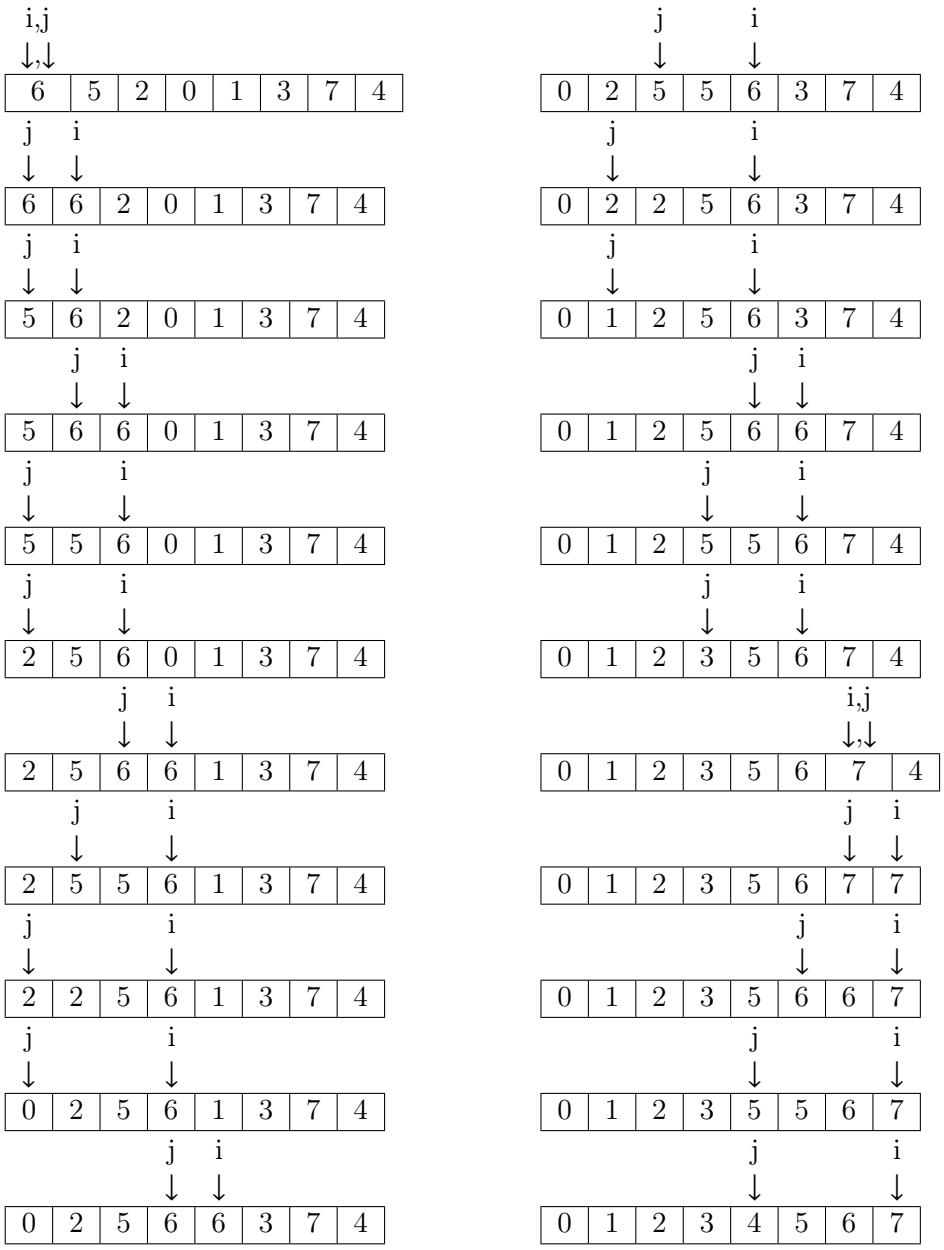


Abbildung 3.2.: Eine Beispielsequenz in Insertionsort

Die Methode *SETHEAP* wird benutzt, um aus einem Feld einen Max-Heap zu konstruieren. Per Definition ist das größte Element einer Menge, welche durch einen Max-Heap dargestellt wird, das Wurzel-Element. Sobald der Max-Heap konstruiert ist, wird das größte Element (die Wurzel) an das logische Ende des Feldes getauscht und das logische Ende verringert. Danach muss die Heap-Bedingung wiederhergestellt werden. Sobald das logische Ende am Anfang angekommen ist, ist das Feld sortiert.

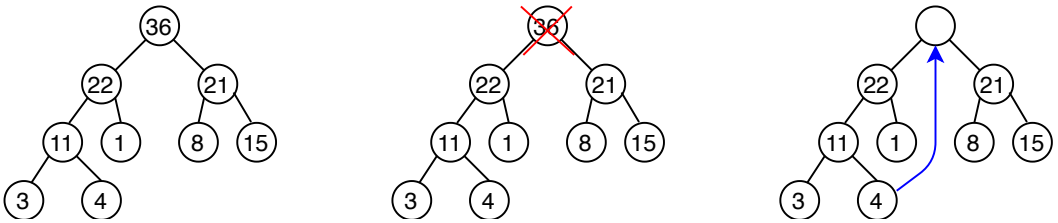


Abbildung 3.3.: Iterationsschritt beim Sortieren mit Max-Heaps. Die Grafik stammt abgeändert aus der Vorlesung Informatik 2 von Vahrenhold in 2018 [42].

Im Folgenden wird der Heapsort-Algorithmus nach J.W.J. Williams dargestellt. Dieser besteht aus den Methoden *SETHEAP*, *INHEAP*, *OUTHEAP* und *SWOPHEAP*. *SETHEAP* ordnet dabei N Elemente zu einem Heap, *INHEAP* fügt einem existierenden Heap ein neues Element hinzu, *OUTHEAP* extrahiert das letzte Element eines Heaps und *SWOPHEAP* ist das Ergebnis von *INHEAP* gefolgt von *OUTHEAP*. Die Elemente des Arrays A sind dabei nach dem Ende jeder Methode als Heap angeordnet.

Algorithmus 3.16 Heapsort

```
1: function SWOPHEAP(real array A, integer n, real in,out)
2:     integer i,j; real temp,temp1;
3:     if in ≤ A[1] then
4:         out ← in
5:     else
6:         i ← 1;
7:         A[n + 1] ← in;
8:         out ← A[1];
9:         scan: j ← i + i
10:        if j ≤ n then
11:            temp ← A[j]
12:            temp1 ← A[j + 1]
13:            if temp1 < temp then
14:                temp ← temp1
15:                j ← j + 1
16:            if temp < in then
17:                A[i] ← temp;
18:                i ← j;
19:                goto scan
20:        A[i] ← in
1: function INHEAP(real array A, integer n, real in)
2:     integer i,j;
3:     i ← n ← n + 1;
4:     scan:
5:     if i > 1 then
6:         j ← i/2;
7:         if in < A[j] then
8:             A[i] ← A[j]
9:             i ← j;
10:            goto scan
11:    A[i] ← in
1: function OUTHEAP(real array A, integer n, real out)
2:     SWOPHEAP(A,n − 1,A[n],out);
3:     n = n − 1
1: function SETHEAP(real array A, integer n)
2:     integer j;
3:     j = 1;
4:     L: INHEAP(A, j, A[j + 1]);
5:     if j < n then
6:         goto L;
```

Das Verfahren mit dem 1964 von Williams [46] vorgestelltem Pseudocode ist für diese Arbeit spannend, da es im Gegensatz zu Quicksort keinen schlechtesten Fall hat der auftreten kann und die durchschnittliche Laufzeit mit $N \log N$ optimal ist. Außerdem arbeitet es In-Place aber nicht stabil. Da Heapsort im Regelfall (2-5 mal)

langsamer ist als Quicksort, wird Quicksort für den generellen Gebrauch bevorzugt. Die Korrektheit und Laufzeit werden als gegeben angenommen und können im Buch „Algorithmen und Datenstrukturen“ [38] nachvollzogen werden.

Abbildung 3.4 zeigt eine Beispielsequenz in Heapsort. Der Zeiger *groesse* gibt das logische Ende des Feldes an.

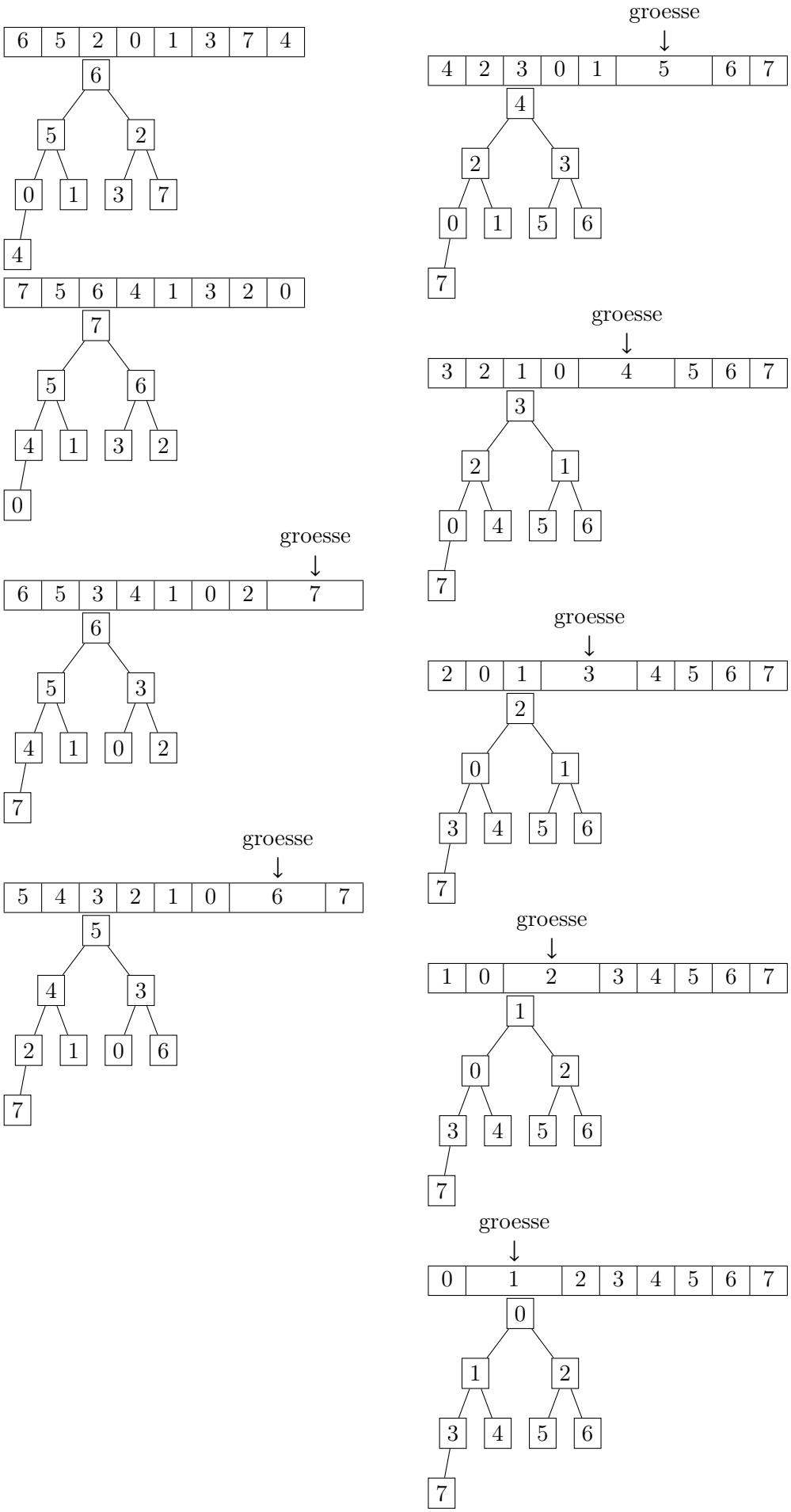


Abbildung 3.4.: Eine Beispielsequenz in Heapsort.

4. Einführung Quicksort

Der Quicksort Algorithmus wurde 1962 von C.A.R. Hoare vorgestellt. Er sortiert Eingaben durch rekursives teilen. Das Verfahren ist eher auf Einzelprozessor Systeme optimiert und arbeitet In-Place. Quicksort hat zwar eine erwartete Laufzeit von $O(N \log N)$, jedoch eine schlechteste Laufzeit von $O(N^2)$. In diesem Kapitel wird gezeigt, wieso der Algorithmus trotzdem seit über 50 Jahren eines der besten Sortierverfahren ist und auf welche Art und Weisen er optimiert werden kann.

4.1. Hoares Quicksort

Sei $A[\ell, ..., r]$ die Eingabesequenz, dann benutzt das Verfahren die Divide-and-conquer-Strategie wie folgt:

- Divide: Finde Pivot-Element $A[q]$ und teile Eingabesequenz in zwei Teilbereiche, so dass alle Elemente in $A[\ell, ..., q-1]$ kleiner als das Pivot-Element und alle Elemente in $A[q+1, .., r]$ größer als das Pivot-Element sind, wobei das Pivot-Element das q-te größte Element ist.
- Conquer: Bearbeite die Zwei Teilsequenzen rekursiv.
- Combine: Wird automatisch durch Konkatenation der Teilprobleme mit dem Pivot-Element durchgeführt.

Im Folgenden werden die Divide und die Conquer Phase vorgestellt.

4.1.1. Algorithmus

Algorithmus 4.1 Quicksort

```
1: function QUICKSORT(A[ℓ, ..., r])
2:     if ℓ < r then
3:         choosePivot(A[ℓ, ..., r])
4:         integer cut ← partition(A, ℓ, r)
5:         Quicksort(A, ℓ, cut − 1)
6:         Quicksort(A, cut + 1, r)
```

Der Algorithmus 4.1 stammt in abgeänderter Form aus dem BlockQuicksort-Paper von Edelkamp und Weiß [10]. Mit zwei Zeigern wird der aktuelle Arbeitsbereich gespeichert. Da immer auf dem selben Feld gearbeitet wird, benötigt das Verfahren keinen zusätzlichen Speicherplatz für weitere Felder. Am Anfang sollten die beiden Zeiger auf den Anfang und das Ende der Eingabesequenz zeigen. Die Methode *choosePivot* wählt ein Pivot-Element aus der Sequenz und tauscht es an das Ende des Feldes. Strategien zur Pivot-Wahl werden später besprochen. Bis dahin kann angenommen werden, dass immer das letzte Element als Pivot-Element gewählt

wird, die Methode kann bis dahin ignoriert werden. Nun bringt der Algorithmus *partition* alle Elemente im Arbeitsbereich, die größer als das Pivot-Element sind auf die rechte Seite und alle die kleiner sind auf die linke[1]. Die Grenzen dieser beiden Seiten werden mit dem Rückgabewert der Methode ermittelt. Zwischen der linken und rechten Seite befindet sich das Pivot-Element. Danach wird die Methode doppelt rekursiv aufgerufen und jeweils die beiden Hälften des aktuellen Bereichs sortiert. Dabei wird der Arbeitsbereich einmal auf die rechte und einmal auf die linke Seite gesetzt. Solange sich die Zeiger des Arbeitsbereiches noch nicht überkreuzt haben, gibt es noch mindestens ein weiteres nicht betrachtetes Element zwischen ihnen und der Vorgang wird wiederholt.

Der wesentliche Schritt bei Quicksort ist jedoch die Partitionierung (Divide). Diese wird im folgenden als Pseudocode vorgestellt.

Algorithmus 4.2 Hoares Partitionierung

```
1:  function PARTITION(A,ℓ,r)
2:      pivot ←A[r];
3:      integer i ← ℓ; j ← r − 1; pivotIndex ← r;
4:      while i < j do
5:          while i < r − 1 und A[i] < pivot do
6:              i++
7:          while j > ℓ und A[j] >pivot  do
8:              j--
9:          if i < j then
10:             swap(A[i],A[j])
11:             i++
12:             j--
13:     if A[i] > pivot then
14:         swap(A[i],A[pivotIndex])
15:     else if i = j then
16:         swap(A[++i],A[pivotIndex])
17:     return i
```

Algorithmus 4.2 zeigt einen Beispielalgorithmus für die Partitionierung nach Hoare. Es werden alle Elemente, welche kleiner als das Pivot-Element sind auf die linke Seite und alle Elemente, welche größer als das Pivot-Element sind auf die rechte Seite getauscht. Dafür wird der Zeiger der linken Seite erhöht, bis ein Element gefunden wird, welches größer als das Pivot-Element ist und somit nicht auf die linke Seite gehört, danach wird der Zeiger der rechten Seite verringert, bis ein Element gefunden wird, welches kleiner als das Pivot-Element ist und somit nicht auf die rechte Seite gehört. Sollten beide Zeiger ein Element gefunden haben, so werden diese getauscht. Dies wird solange weitergeführt, bis sich die Zeiger überkreuzen, dann wird das Pivot-Element zwischen den beiden Seiten getauscht und sein Index zurückgegeben.

[1] Elemente die einen gleichen Wert wie das Pivot-Element haben bleiben also einfach auf ihrer Seite.

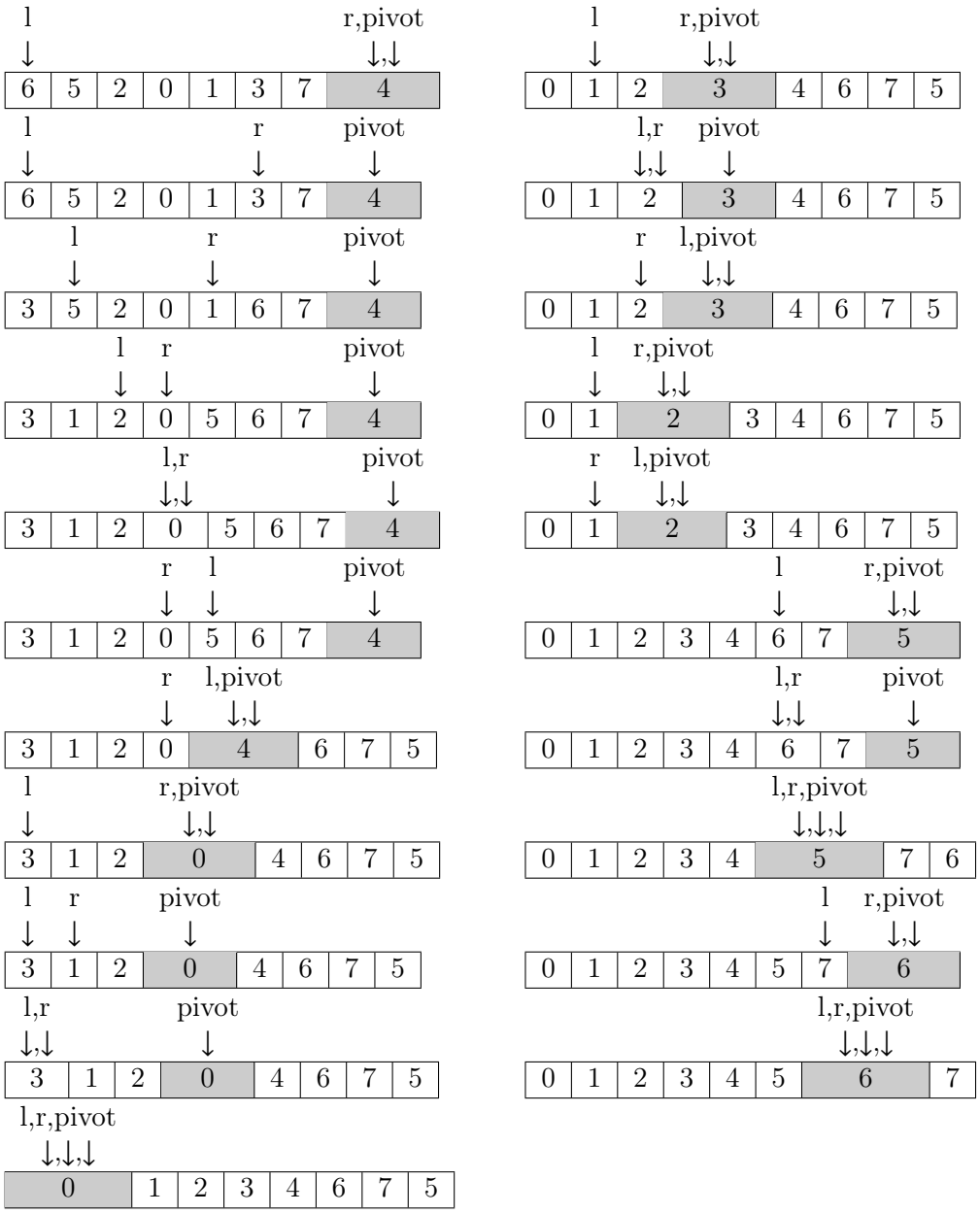


Abbildung 4.1.: Eine Beispielsequenz in Quicksort

Abbildung 4.1 zeigt eine Beispielsequenz in Quicksort. Das aktuelle Pivot-Element ist grau hervorgehoben und wird mit dem Zeiger *pivot* indiziert. Des Weiteren geben die Zeiger l und r die aktuellen Bereiche an.

4.1.2. Analyse von Quicksort

Korrektheit der Methode partition

Im Folgenden wird die Korrektheit der Methode *partition* (siehe 4.2) vom Autor mithilfe einer Schleifeninvariante bewiesen.

Schleifeninvariante 4.3. *Vor jedem Schleifendurchlauf in Zeile 4 gilt:* $\forall \ell \leq x < i$: $A[x] \leq Pivot$ *und* $\forall j < y \leq r-1$: $A[y] \geq Pivot$.

Für den Beweis zeigen wir zuerst folgendes Hilfslemma:

Lemma 4.4. *Sei N die Länge der Eingabe mit $N = r - \ell + 1$. Die Kosten für einen Aufruf von partition sind dann maximal N.*

Die Partitionierung besteht aus einer äußeren Schleife und zwei inneren Schleifen. Die äußere Schleife hat dabei die Bedingung $i < j$. In Zeilen 6, 8, 11 und 12 des Algorithmus sehen wir, dass i nur erhöht und j nur verringert wird.

Da in jeder Iteration der äußeren Schleife i inkrementiert (Zeile 11) und j dekrementiert (Zeile 12) wird, in jeder Iteration der ersten inneren Schleife i inkrementiert wird (Zeile 6) und in jeder Iteration der zweiten inneren Schleife j dekrementiert wird (Zeile 8), folgt mit der Bedingung *while* $i < j$ (Zeile 4), dass die drei Schleifen zusammen höchstens $r - \ell - 1$ mal ausgeführt werden können. Mit Abbildung 4.2 ergeben sich die Kosten der Methode *partition*: $O(r - \ell)$. Weiter kann $O(r - \ell)$ durch $max_{0 \leq \ell \leq r \leq N}\{r - \ell\}$ mit $O(N)$ abgeschätzt werden. Somit belaufen sich die Kosten eines Aufrufes von Partition auf maximal $O(N)$. □

Initialisierung: Es gilt nach Zeile 3: $i = \ell$ und $j = r - 1$. Damit ist $\ell \leq x < i = \ell$ leer und $j < y \leq r - 1 = j$ ebenso. Damit ist die Schleifeninvariante bereits erfüllt.

Aufrechterhaltung: Da vor Zeile 4 die Schleifeninvariante galt, zusätzlich in Zeile 5-6 i nur erhöht wird, falls $A[i] < Pivot$ und in Zeile 7-8 j nur verringert wird, falls $A[j] > Pivot$ ist, gilt die Invariante nach Zeile 8 immer noch. Nun folgen Zeilen 9-12. Diese werden nur ausgeführt, falls $i < j$ gilt, was bedeutet, dass die Schleifen in Zeile 5 und Zeile 7 abgebrochen sind, da $A[i] \geq Pivot$ und $A[j] \leq Pivot$ galt.

Angenommen dies gilt nicht, dann wären die Schleifen in Zeilen 5 und 7 abgebrochen da entweder $i \geq r - 1$ und/oder $j \leq \ell$ galt. In Zeile 3 wird j mit $r - 1$ und i mit ℓ initialisiert. Danach wird j nur noch verringert (Zeilen 8 und 12) und i erhöht (Zeilen 6 und 11). Es gilt: $j < r - 1$ und $i \geq \ell$. Damit muss bereits mindestens eine der beiden Aussagen: $j \leq r - 1 \leq i \Rightarrow j \leq i$ ↯ oder $j < \ell \leq i \Rightarrow j < i$ ↯ gelten. Beide Aussagen führen zu einem Widerspruch zur Annahme $i < j$.

Abbildung 4.2.: Kosten der Instruktionen in *partition*

Zeile	*Kosten der Instruktionen*
2	$O(1)$
3	$O(1)$
	$\#outerLoop \cdot \left\{ \begin{array}{c\|c} 4 & O(1) \\ & \#firstInnerLoop \cdot \left\{ \begin{array}{c\|c} 5 & O(1) \\ 6 & O(1) \end{array} \right\} \\ & \#secondInnerLoop \cdot \left\{ \begin{array}{c\|c} 7 & O(1) \\ 8 & O(1) \end{array} \right\} \\ 9-16 & O(1) \end{array} \right\}$
17	$O(1)$

Insgesamt:
$$O(1) + \#outerLoop \cdot (O(1) + \#firstInnerLoop \cdot O(1) + \#secondInnerLoop \cdot O(1)$$

Dabei ist $\#firstInnerLoop$ die Anzahl der Schleifendurchläufe in dem jeweiligen Durchlauf der äußeren Schleife und nicht die Gesamtanzahl.

Falls Zeilen 9-12 also ausgeführt werden, dann haben die Zeiger i und j jeweils ein Element gefunden, welches auf die andere Seite getauscht werden muss. In Zeile 10 werden diese Elemente in i und j dann vertauscht, es gilt danach $A[i] \leq Pivot$ und $A[j] \geq Pivot$, damit gilt die Invariante auch bei Erhöhung von i und Verringerung von j um 1.

Terminierung: Aus 4.4 folgt direkt: Die Schleife terminiert nach maximal $O(N)$ Durchläufen.

Damit ist die Schleifeninvariante 4.5 bewiesen. □

Weiter soll mit ihrer Hilfe die Korrektheit der Methode Partition bewiesen werden.

Satz 4.5. *Am Ende der Methode partition (4.2) gilt:* $\forall x < i : A[x] \leq Pivot$, $\forall x > i : A[x] \geq Pivot, A[i] = Pivot$

Aufgrund der Gültigkeit der Schleifeninvariante wissen wir, dass vor der letzten Abfrage der Schleifenbedingung in Zeile 4 gilt: $\forall \ell \leq x < i : A[x] \leq Pivot$ und $\forall j < y \leq r-1 : A[y] \geq Pivot$ und $i \geq j$. Nun gibt es vier Fälle:

Fall 1 $i > j$

Fall 2 $i = j$ und $A[i] = Pivot$

Fall 3.1 $i = j$ und $A[i] < Pivot$ und $i < r - 1$

Fall 3.2 $i = j$ und $A[i] < Pivot$ und Sonderfall: $i = r - 1$

Fall 4.1 $i = j$ und $A[i] > Pivot$ und $i > \ell$

Fall 4.2 $i = j$ und $A[i] > Pivot$ Sonderfall: $i = \ell$

Anmerkung: Da die Schleife in Zeile 4 kein weiteres Mal ausgeführt wird, ist der Fall $i < j$ nicht möglich.

Fall 1: $i > j$ bedeutet, dass $A[i] \geq Pivot$, sonst wäre j nicht ein weiteres Mal dekrementiert worden und $A[j] \leq Pivot$, sonst wäre i nicht ein weiteres Mal inkrementiert worden. Die Bedingung in Zeile 13 ist daher erfüllt, nach dem Tauschen von $A[i]$ und $A[pivotIndex] = A[r]$ gilt: $A[i] = Pivot$ und $A[r] \geq Pivot$. Damit ist die Aussage: $\forall x < i : A[x] \leq Pivot$, $\forall x > i : A[x] \geq Pivot, A[i] = Pivot$ für diesen Fall bewiesen.

Fall 2: $i = j$ und $A[i] = Pivot$ bedeutet, dass wir für den Beweis der Aussage nichts weiter zu tun brauchen und mit $A[i] \leq Pivot$ ist die Bedingung in Zeile 13 nicht erfüllt. Daher wird kein weiters Element vertauscht, es gilt bereits $\forall x < i : A[x] \leq Pivot$, $\forall x > i : A[x] \geq Pivot, A[i] = Pivot$, damit ist die Aussage für diesen Fall bewiesen.

Für Fall 3 und Fall 4 kommen jeweils zwei Möglichkeiten in Betracht, da in Fall 3 der Sonderfall betrachtet werden muss, dass sich das Element am rechten Rand befindet und in Fall 4 der Sonderfall betrachtet werden muss, dass sich das Element am linken Rand befindet. In diesen Sonderfällen ist die aus dem Algorithmus hervorgehende jeweilige rechte bzw. linke Teilsequenz leer. In beiden Möglichkeiten gilt $i = j$, weil vorher das Element an Stelle $i - 1$ und das Element an Stelle $j + 1$ getauscht wurden. Danach wurde l nach Zeile 11 erhöht und j verringert. Ein weiterer Schleifendurchlauf ist nun nicht möglich, da die Bedingung $i < j$ in Zeile 4 nicht mit wahr ausgewertet wird. Das Pivot-Element muss nun an die richtige Stelle gebracht werden.

In **Fall 3.1** gilt dabei: $A[i] < Pivot$. Würde nun einfach $A[i]$ mit dem letzten Element (dem Pivot-Element) getauscht werden, wäre das Element, welches sich ehemalig in $A[i]$ befand, auf der falschen Seite. Das Element rechts von Index i bzw. j ist aber bereits auf der rechten Seite, da der Zeiger i bzw. j die Grenze zwischen Elementen angibt welche kleiner als das Pivot-Element sind zu denen, die größer als das Pivot-Element sind und das Element genau auf der Grenze nach Bedingung kleiner dem Pivot ist. Damit kann das Element $i+1$ bzw. $j+1$ mit jedem Element der rechten Seite getauscht werden ohne die Ordnung bezüglich des Pivot-Elementes zu zerstören. Dies passiert in Zeile 16. Nach dem Erhöhen von i und Tauschen von $A[i]$ und $A[r]$ gilt: $A[i] = Pivot$ und $A[r] \geq Pivot$. Damit ist die Aussage: $\forall x < i : A[x] \leq Pivot$, $\forall x > i : A[x] \geq Pivot, A[i] = Pivot$ bewiesen.

In **Fall 4.1** gilt $A[i] > Pivot$, somit gehört das Element $A[i]$ bereits zur rechten Seite und kann mit jedem Element der rechten Seite getauscht werden ohne die Ordnung bezüglich des Pivot-Elementes zu zerstören. Dies passiert in Zeile 14. Nach dem Tauschen von $A[i]$ und $A[r]$ gilt: $A[i] = Pivot$ und $A[r] \geq Pivot$. Damit ist die Aussage: $\forall x < i : A[x] \leq Pivot$, $\forall x > i : A[x] \geq Pivot, A[i] = Pivot$ für diesen Fall bewiesen.

Die Fälle 3.2 und 4.2 können nur noch Randfälle sein, denn da die Zeiger i und j nicht nach einem Tausch gestoppt sind (das wären die Fälle: $A[i] < Pivot$ Fall 3.1, Fall 4.1), müssen sie aus den Schleifen in Zeile 5 und Zeile 7 hervorgegangen sein.

Das kann aber nur am Rand passiert sein, denn in Fall 3 wäre i weiter inkrementiert worden, da noch kein Element gefunden wurde welches größer als das Pivot ist. Es wurde gestoppt, weil $i < r$ nicht mehr erfüllt war (rechter Rand). In Fall 4 wäre j weiter dekrementiert worden, da noch kein Element gefunden wurde, welches kleiner als das Pivot ist. Es wurde gestoppt, weil $j > l$ nicht mehr erfüllt war (linker Rand).

Fall 3.2: Mit $A[i] < Pivot$ ist die Bedingung $A[i] > Pivot$ in Zeile 13 nicht erfüllt, jedoch die Bedingung $A[i] \not> Pivot$ und $i = j$ in Zeile 15. Da sich i am rechten Rand befindet, wird das Element an Stelle $i + 1$ in Zeile 16 mit sich selbst getauscht. Es gilt: das Pivot-Element hat den größten Index (r), die Grenze i bzw. j der bezüglich des Pivot-Elementes kleineren Elemente von den größeren liegt am rechten Rand ($r - 1$) und für das Element genau auf der Grenze an Index i: $A[i] < Pivot$. Damit ist die Menge der Elemente welche größer als das Pivot-Element sind, leer und das Pivot-Element steht bereits an der richtigen Stelle. Damit muss für den Beweis nichts mehr getan werden außer den Zeiger i auf das Pivot-Element zu bringen, was in Zeile 16 geschieht. Damit ist die Aussage $\forall x < i : A[x] \leq Pivot$, $\forall x > i$ (so ein x gibt es in diesem Fall nicht, da diese Menge leer ist) : $A[x] \geq Pivot, A[i] = Pivot$ für diesen Fall bewiesen.

Fall 4.2: Da sich die Zeiger i und j am linken Rand befinden und zusätzlich $A[i] > Pivot$ gilt, muss die Menge der Elemente welche kleiner als das Pivot-Element sind leer sein. Damit das Pivot-Element diese leere Menge von der Menge der Elemente welche größer als das Pivot-Element sind abtrennt, muss es an den linken Rand vertauscht werden. Mit $A[i] > Pivot$ ist die Bedingung in Zeile 13 erfüllt. Das Pivot-Element wird an die Stelle i vertauscht, wobei sich der Index i am linken Rand befindet. Damit ist die Aussage $\forall x < i$ (so ein x gibt es in diesem Fall nicht, da diese Menge leer ist) : $A[x] \leq Pivot$, $\forall x > i : A[x] \geq Pivot, A[i] = Pivot$, für diesen Fall bewiesen.

Korrekheit des Quicksort-Verfahrens

Im Folgenden wird die Korrektheit der Quicksort-Hauptmethode (4.1) vom Autor dieser Arbeit mittels vollständiger Induktion bewiesen. Wie auch zuvor werden wir dabei die Methode *choosePivot* außer Betracht lassen und annehmen, dass immer das letzte Element als Pivot-Element gewählt wird.

Satz 4.6. *Angenommen, die Partitionsmethode funktioniert korrekt (siehe Abschnitt 4.1.2), das heißt, für eine Eingabesequenz A gilt nach Ausführung der Methode:* $\forall x < cut : A[x] \leq Pivot$, $\forall x > cut : A[x] \geq Pivot, A[cut] = Pivot$, *dann ist die Eingabesequenz nach Ende der Rekursion in richtiger Reihenfolge sortiert.*

Sei dafür N die Länge der Eingabe mit $N = r - \ell + 1$.

Induktionsanfang: $N = 1$

Für $N = 1$ gilt die Behauptung, da einelementige Eingaben automatisch sortiert sind.

Induktionsvoraussetzung (I.V.): Die Behauptung gelte für ein beliebiges, aber festes N.

Induktionsschritt: $N \to N+1$
Da der Zeiger *cut* (Zeile 4) im Intervall $[\ell, ..., r]$ liegt und $r - \ell + 1 = N + 1$ gilt, haben die geteilten Bereiche: $[\ell, ..., cut - 1]$ und $[cut + 1, ..., r]$ höchstens Größe $N \overset{\text{I.V.}}{\Rightarrow}$ Beide Bereiche sind nach Ausführung der Methode in der richtigen Reihenfolge sortiert. Aus der Korrektheit der Partitionsmethode (siehe Abschnitt 4.1.2) folgt nun bereits die Bedingung, da gilt: $\forall x < cut < y : A[x] < A[cut] < A[y]$. Mit vollständiger Induktion über N folgt die Behauptung $\forall N \in \mathbb{N}$.

Laufzeit von Quicksort

Im Folgenden wird die durchschnittliche Quicksort-Laufzeit von $\approx 1,386 \cdot N \log N$ bewiesen. Die Beweis-Ideen zum durchschnittlichen Fall sind zusammengetragen aus der MIT Vorlesung „Quicksort: Randomisierte Algorithmen“ von Prof. Charles Leiserson und Prof. Erik Demaine [32], der Lektüre „Einführung in Algorithmen“ von Cormen et. al. [9] und der darauf aufbauenden UIC Vorlesung von Jeffrey S. Leon [22]. Für ein besseres Verständnis der Methode werden außerdem die Laufzeit im schlechtesten Fall sowie im besten Fall vom Autor bewiesen. Im Weiteren wird die Laufzeit der Methode Partition: $f(N) = O(r - \ell)$ mit der maximalen Laufzeit: $max_{0 \le \ell \le r \le N}\{r - \ell\} = O(N)$ abgeschätzt (siehe 4.4).

Laufzeit im schlechtesten Fall:
Im schlechtesten Fall wird immer das Pivot-Element gewählt, welche das kleinste oder größte Element ist. Dadurch wird in jeder Rekursionstiefe die Anzahl der zu betrachteten Elemente nur um einen verringert. Rekursionsgleichung mit $f(n) = O(N)$ und $T(0) = O(1)$:

$$
\begin{aligned}
T(N) &= T(0) + T(N-1) + O(N) \\
\Leftrightarrow T(N) &= O(1) + T(0) + T(N-2) + O(N-1) + O(N) \\
\Leftrightarrow T(N) &= N \cdot O(1) + \sum_{k=0}^{N} O(k) \\
\Leftrightarrow T(N) &= O(N) + \frac{N(N+1)}{2} \\
\Leftrightarrow T(N) &= O(N^2)
\end{aligned}
$$

Die quadratische Laufzeit im schlechtesten Fall ist einer der größten Nachteile von Quicksort. Diese Problematik und einen Ansatz um diese zu beheben wird in Unterabschnitt 4.2.3 vorgestellt.

Laufzeit im besten Fall:[2]

Im besten Fall wird immer das Pivot-Element benutzt, welches die Eingabesequenz in zwei gleich große Sequenzen aufteilt. Die Rekursionsgleichung[3] lautet $T(N) = 2 \cdot T(\frac{N}{2}) + f(N)$. Mit $f(n) = O(N)$ und dem zweiten Fall vom Master-Therorem (siehe: 3.4) folgt aus: $f(N) = O(N) \in \Theta(N^1) = \Theta(N^{\log_b a}) \Rightarrow T(N) \in \Theta(N \log N)$.

Laufzeit im durchschnittlichen Fall:

Da wir bisher das Pivot-Element immer als das letzte Element auswählen, sei für die Analyse angenommen, dass die Eingabe zufällig permutiert ist. Dies ist äquivalent zu einer zufälligen Pivot-Wahl. Jedes der N Elemente hat eine gleich wahrscheinliche Chance von $\frac{1}{N}$, das k-te größte Element zu sein. Für die Rekursionsgleichung: $T(N) = T(k-1) + T(N-k) + f(N)$ bei der das k-te größte Element das Pivot-Element ist, sind dann alle Werte für $k \in [1, .., N]$ gleich wahrscheinlich. Daher bilden wir das Mittel der erwarteten Laufzeit über alle möglichen k.

$$\begin{aligned} T(N) &= \frac{1}{N} \sum_{k=1}^{N} T(k-1) + T(N-k) + f(N) \\ &= \frac{1}{N} \sum_{k=1}^{N} T(k-1) + \frac{1}{N} \sum_{k=1}^{N} T(N-k) + \frac{1}{N} \sum_{k=1}^{N} f(N) \\ &= \frac{1}{N} \sum_{k=1}^{N} T(k-1) + \frac{1}{N} \sum_{k=1}^{N} T(N-k) + f(N) \end{aligned}$$

Es gilt mit:

$$\begin{aligned} \sum_{k=1}^{N} T(k-1) &= T(0) + T(1) + \ldots + T(N-2) + T(N-1) \\ &= T(N-1) + T(N-2) + \ldots + T(N-(N-1)) + T(N-N) \\ &= \sum_{k=1}^{N} T(N-k) \end{aligned} \tag{4.1}$$

$$T(N) = \frac{1}{N} \sum_{k=1}^{N} T(k-1) + \frac{1}{N} \sum_{k=1}^{N} T(N-k) + f(N)$$

[2] Nur für das Verständnis, keinerlei Relevanz in der Praxis

[3] Eigentlich muss das Pivot-Element noch herausgerechnet werden und bei ungeradem N kann nicht in gleichgroße Sequenzen geteilt werden. Asymptotisch macht dies jedoch keinen Unterschied.

$$\overset{(4.1)}{=} \frac{2}{N}\sum_{k=1}^{N} T(k-1) + f(N)$$

Ersetze im Weiteren $f(N)$ mit: $O(N)$

$$\begin{aligned}
T(N) &= \frac{2}{N}\sum_{k=1}^{N} T(k-1) + O(N) \\
\Leftrightarrow N \cdot T(N) &= 2\sum_{k=1}^{N} T(k-1) + N^2 \\
\Leftrightarrow (N-1) \cdot T(N-1) &= 2\sum_{k=1}^{N-1} T(k-1) + (N-1)^2 \\
\Leftrightarrow N \cdot T(N) - (N-1) \cdot T(N-1) &= 2\sum_{k=1}^{N} T(k-1) + N^2 - 2\sum_{k=1}^{N-1} T(k-1) - (N-1)^2) \\
\Leftrightarrow N \cdot T(N) - (N-1) \cdot T(N-1) &= 2\sum_{k=1}^{N} T(k-1) - 2\sum_{k=1}^{N-1} T(k-1) + N^2 - (N-1)^2 \\
\Leftrightarrow N \cdot T(N) - (N-1) \cdot T(N-1) &= 2T(N-1) + N^2 - (N^2 - 2N + 1) \\
\Leftrightarrow N \cdot T(N) &= (N-1) \cdot T(N-1) + 2T(N-1) + 2N - 1 \\
\Leftrightarrow N \cdot T(N) &= (N+1) \cdot T(N-1) + 2N - 1 \\
\Leftrightarrow \frac{T(N)}{N+1} &= \frac{T(N-1)}{N} + \frac{2N-1}{N \cdot (N+1)} \\
\Leftrightarrow \frac{T(N)}{N+1} &< \frac{T(N-1)}{N} + \frac{2(N+1)}{N \cdot (N+1)} \\
\Leftrightarrow \frac{T(N)}{N+1} &< \frac{T(N-1)}{N} + \frac{2}{N}
\end{aligned}$$

Substituiere $\frac{T(N)}{N+1}$ mit: $\hat{T}(N)$

$$\begin{aligned}
&\Leftrightarrow \hat{T}(N) < \hat{T}(N-1) + \frac{2}{N} \\
&\Leftrightarrow \hat{T}(N) < \hat{T}(N-2) + \frac{2}{N-1} + \frac{2}{N} \\
&\Leftrightarrow \hat{T}(N) < \hat{T}(N-3) + \frac{2}{N-2} + \frac{2}{N-1} + \frac{2}{N} \\
&\Leftrightarrow \hat{T}(N) < \hat{T}(1) + \frac{2}{N-(N-2)} + \ldots + \frac{2}{N-2} + \frac{2}{N-1} + \frac{2}{N} \\
&\overset{\hat{T}(1)=0}{\Leftrightarrow} \hat{T}(N) < 2 \cdot \sum_{k=2}^{N} \frac{1}{k}
\end{aligned}$$

$$\Leftrightarrow \hat{T}(N) < 2 \cdot \sum_{k=1}^{N} \frac{1}{k} - 2$$

Mit der n-ten Harmonischen Nummer nach 3.5:

$$\mathcal{H}_N = 1 + \frac{1}{2} + .. + \frac{1}{N} = \ln N + \gamma + \frac{1}{2N} - \frac{1}{12N^2} + \frac{\epsilon_N}{120N^4} \tag{4.2}$$

Für: $0 < \epsilon_N < 1$ und $\gamma \approx 0.57721$ (Euler-Mascheroni Konstante)

$$\Leftrightarrow \hat{T}(N) < 2 \cdot \sum_{k=1}^{N} \frac{1}{k} - 2 \overset{(4.2)}{\approx} 2 \cdot (\ln N + \gamma + \frac{1}{2N} - \frac{1}{12N^2} + \frac{\epsilon_N}{120N^4}) - 2$$
$$\Leftrightarrow \hat{T}(N) = 2 \cdot \ln N + 2 \cdot \gamma + \frac{1}{N} - \frac{2}{12N^2} + \frac{2\epsilon_N}{120N^4} - 2$$
$$\Leftrightarrow \hat{T}(N) \approx 2 \cdot \ln N + \frac{1}{N} - \frac{2}{12N^2} + \frac{2\epsilon_N}{120N^4}$$

Weiter Mit:

$$\log_2 N = \frac{\log_e N}{\log_e 2} \Leftrightarrow \ln N = \log_e N = \log_2 N \cdot \log_e 2 \tag{4.3}$$

$$\Leftrightarrow \hat{T}(N) \overset{(4.3)}{\approx} 2 \cdot \log_2 N \cdot \log_e 2 + \frac{1}{N} - \frac{2}{12N^2} + \frac{2\epsilon_N}{120N^4}$$
$$\Leftrightarrow \hat{T}(N) \approx 1,386 \cdot log_2 N + \frac{1}{N} - \frac{2}{12N^2} + \frac{2\epsilon_N}{120N^4}$$
$$\overset{\hat{T}(N)=\frac{T}{N+1}}{\Leftrightarrow} T(N) \approx (N+1) \cdot 1,386 \cdot log_2 N + \frac{1}{N} - \frac{2}{12N^2} + \frac{2\epsilon_N}{120N^4}$$
$$\Leftrightarrow T(N) \approx 1,386 N \log N$$

Wir erhalten somit eine Laufzeit, welche 38,6% schlechter als das bewiesene Optimum ist.

Quicksort hat nicht nur theoretisch eine gute Laufzeit sondern kann auch sehr schnell implementiert werden, sodass eine kleine Konstante C im Term $CN \log N$ benutzt werden kann. Im Vergleich zu Mergesort, einem Verfahren, welches in $N \log N$ sortiert, schneidet Quicksort zwar schlechter ab, jedoch benutzt das Verfahren dafür eine sehr kleine Anzahl an Instruktionen. Des Weiteren benutzen wir bisher immer ein zufälliges Element als Pivot-Element, die Konstante $c = 1,386$ kann durch andere Auswahlverfahren auf bis zu $c = 1 + \epsilon$ mit einem sehr kleinen ϵ verringert werden. Zusätzlich sollte nicht vergessen werden, dass Quicksort im Gegensatz zu Mergesort In-Place arbeitet.

Sprungfehler in Quicksort

Grundlegend kann in Quicksort jeder Element-Pivot Vergleich einen Sprungfehler hervorrufen. Im Folgenden wollen wir uns die Sprungfehler in Quicksort für einen statischen Sprungvorhersager anschauen. In *partition* haben wir drei While-Schleifen (Zeile 4, 5, 7). Die erste ruft pro Aufruf der Partitionierung nur einen Sprungfehler in der letzten Iteration hervor, die beiden anderen können in jeder Iteration einen Sprungfehler hervorrufen. Außerdem existieren drei if-Bedingungen (Zeile 9, 13, 15). Die erste ruft einen Sprungfehler pro Aufruf der Methode hervor, die anderen beiden schließen sich gegenseitig aus und rufen zusammen auch einen Sprungfehler pro Aufruf der Methode hervor. Die Schleife in Zeile 4 wird höchstens $\frac{r-\ell-1}{2}$ mal durchlaufen. Insgesamt erhalten wir für *partition*: $1 + \frac{(r-\ell-1)}{2} \cdot (1+1) + 1 + 1$ Sprungfehler. Damit können die Sprungfehler abgeschätzt werden durch: $O(N)$. In der Hauptmethode von Quicksort kommt noch ein Sprungfehler aus Zeile 2 dazu. Damit erhalten wir analog zu Abschnitt 4.1.2: $\approx 1,386N \log N$ Sprungfehler. Dabei haben wir die Sprungfehler in der Methode *partition* nach oben abgeschätzt. Für schärfere Begrenzungen wird an dieser Stelle auf die Analyse in der Publikation „Analyse von Sprungfehlern in Quicksort“ von Martínez et. al. [24] verwiesen.

4.1.3. Pivot-Wahl

Die Wahl des richtigen Pivot-Elementes ist ein zentraler Punkt in der Optimierung von Quicksort. Neben der Strategie, immer das letzte Element als Pivot-Element auszuwählen, gibt es weitere, deutlich bessere Strategien. Dabei ist die Pivot-Wahl offensichtlich entscheidend für die Laufzeit, denn das Pivot-Element beeinflusst unter anderem die Größe der Teilprobleme, so gibt es neben $\frac{N}{2}$ zu $\frac{N}{2} - 1$ Aufteilungen genauso $N-2$ zu 1 Aufteilungen. Es gibt jedoch viel mehr Aspekte in Quicksort, die durch ein gut gewähltes Pivot-Element gesteuert werden können. So können die in Abschnitt 4.1.2 angegebenen Vergleiche verringert werden, womit sich automatisch die Anzahl der falschen Sprungvorhersagen verringert. Weiter werden wir sehen, dass eine gute Pivot-Wahl nah am Median nicht unbedingt gut sein muss, da die Sprungvorhersagen damit schwerer werden. Zusätzlich wird der Schritt von einem fix gewählten Element zu einem zufälligen betrachtet. Dieser sorgt für eine Randomisierung von Quicksort, welches eine große Sicherheitslücke schließt (siehe weiter unten in diesem Kapitel). Letzten Endes wird die Pivot-Wahl jedoch immer ein Trade-Off sein, da komplexere Auswahl-Strategien die besser sicherstellen, dass die Teil-Probleme gleich groß sind, im Gegenzug größeren Berechnungsaufwand implizieren. Da die Strategie der Pivot-Wahl ein großes, gut erforschtes Themengebiet ist, sei hier darauf hingewiesen, dass dies nur eine kleine Einführung darstellen und die Pivot-Wahl nicht unbedingt im Vordergrund dieser Arbeit stehen soll.

Median

Wenn man sich das Quicksort-Verfahren näher anschaut, wird man schnell feststellen, dass das Pivot-Element die Eingabesequenz am besten in zwei gleich große Teilmengen aufteilen sollte. Dies würde man erreichen, falls das Pivot-Element das Median Element der Eingabesequenz wäre. Natürlich ist es jedoch viel zu kostenintensiv, das Median Element der Eingabesequenz zu berechnen. Die Ersparnisse im folgenden Sortiervorgang wären komplett überschattet von dem Berechnungsaufwand. Stattdessen ist eine bekannte Strategie der Pivot-Wahl die Bildung des Medians aus einem Teilmenge der Eingabesequenz. Die Teilmenge hat dabei meistens die Form $2t + 1$ für ein $t \in \mathbb{N}$, $t = 1$ stellt dabei mit $Median - Of - Three$ die am meisten benutzte Variante dar. Aus dieser $2t + 1$-elementigen Teilmenge der Eingabe wird dann der Median als Pivot-Element gewählt [25].

Die meist benutzten Median Strategien der Pivot-Wahl sind im Folgenden aufgelistet:

- $Median - of - Three$
- $Median - of - Five$
- $Median - of - Twenty - Three$
- $Median - of - Three - Medians - of - Three$
- $Median - of - Three - Medians - of - Five$
- $Median - of - Five - Medians - of - Five$
- $Median - of - \sqrt{(N)}$

Der tatsächliche Aufwand und die Optimalität des Ergebnisses für verschiedene Pivot-Wahl Strategien ist dabei abhängig von der Größe, der Menge oder auch dem Typ der Eingabedaten. Eine bessere Wahl des Medians führt zu weniger Vergleichen, mehr Sprungfehlern (da extra Sprünge in der Median-Berechnung auftreten) und ungefähr gleich vielen Swaps. Eine Optimierung von Edelkamp und Weiß [10] bestand darin, die Median Berechnung durch vorherbestimmte Instruktionen sprungfrei zu gestalten.

Skewed Pivots

Als *skewed*[4] bezeichnet man Pivot-Elemente, welche absichtlich so gewählt werden, dass die Größe eines der beiden Teilprobleme überwiegt. Dieser Ansatz soll es ermöglichen, Sprünge wieder vorhersehbarer zu gestalten. Eine diesbezügliche Analyse findet man z.B. in der Abhandlung „Wie Sprunfehler Quicksort beeinflussen“ von Kaligosi und Sanders [17].

[4] eine mögliche Übersetzung lautet: „verzerrt“

Randomisiertes Quicksort und Killer-Sequenzen

Falls die Pivot-Wahl Strategie bekannt ist, können Eingaben erstellt werden, welche eine Laufzeit von $O(N^2)$ benötigen. Dies gilt sowohl für eine feste Pivot-Wahl (z.B. immer das letzte Element auswählen) aber auch für berechnete Pivot-Elemente (z.B. das Median Element einer Teilmenge (erstes, zweites, letztes Element)). Dies stellt eine Sicherheitslücke dar und kann durch zufällige Auswahlen verhindert werden. Daher sollte die Strategie so abgeändert werden, dass nicht immer das letzte sondern ein zufälliges Element als Pivot-Element gewählt wird oder das Pivot-Element nicht als Median des ersten, zweiten und letzten Elementes gewählt wird, sondern von drei zufälligen Elementen. So stellten Chaudhuri und Dempster 1993 [7] einen Algorithmus dar, welcher in linearer Laufzeit eine worst-case Sequenz für Quicksort mit festem Pivot-Element findet. Solch eine konstruierte Sequenz, welche die Laufzeit garantiert auf $O(N^2)$ bringt, wird *Killer-Sequenz* genannt. Ein Beispiel für den Ablauf bei einer solchen Sequenz wird in Abbildung 4.3 dargestellt.

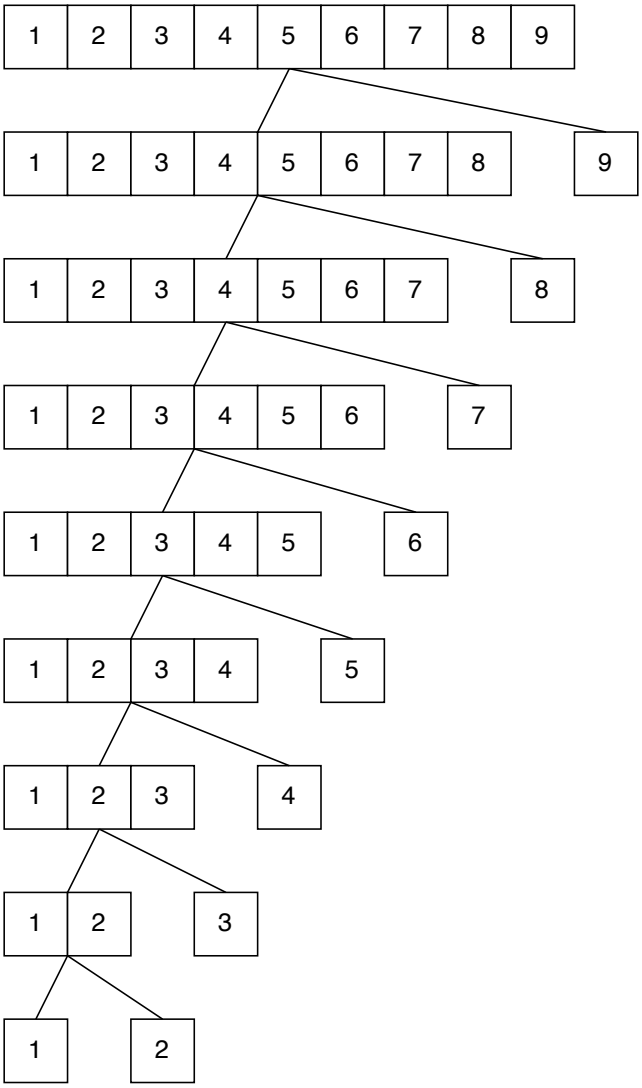


Abbildung 4.3.: Darstellung einer Worst-Case Sequenz von Quicksort. Die Partitionierung teilt die Eingabe jedes mal in $n-1$ und 1 Element(e) auf.

Nach Ottmann et. al. [38] sorgt eine Zufalls-Strategie dafür, dass es keine schlechten Eingabesequenzen mehr gibt, da der auf diese Weise randomisierte Quicksort-Algorithmus jede Eingabe (annähernd) gleich behandelt. Man kann den schlechtesten Fall zwar nicht vermeiden, jedoch kann man gut zeigen, dass der Erwartungswert für die erforderliche Anzahl Schlüsselvergleiche beim Sortieren einer beliebigen aber festen Eingabesequenz mit randomisiertem Quicksort in $O(N \log N)$ liegt.

4.2. Optimierungen und Varianten

Wie im Verlaufe der Arbeit auffallen wird, lässt sich jeder vorgestellte Algorithmus[5] auf die 1962 entstandene Idee von Hoare zurückführen. Die Laufzeit und mögliche

[5]Natürlich ausgenommen Insertionsort und Heapsort

Verbesserungen dieser Grundidee setzen sich im Prinzip aus drei Teilen zusammen.

- Optimierung der Quicksort-Hauptmethode
- Optimierung der Partitionierung
- Optimierung der Pivot-Wahl Strategie

Der erste Teil der Optimierungsansätze bezieht sich auf die Quicksort-Hauptmethode. Dabei können gefundene Optimierungen auf fast alle vorgestellten Algorithmen übertragen werden, da diese alle auf Quicksort aufbauen. Das Ziel wird es am Ende sein, viele der gefundenen Verbesserungen am effizientesten zusammen zu führen.

Weiter können die Verfahren verbessert werden, in dem man die Partitionierung austauscht. Im Gegensatz zu den gerade besprochenen Optimierungen fungiert die Partitionierung als Blackbox, welche einfach ersetzt werden kann. Gefundene Verbesserungen und Ansätze können in einem neuen Verfahren nicht so einfach hinzugefügt werden, vielmehr basieren die Ideen der verschiedenen Partitionierungen aufeinander. Die beiden Ausgangspunkte bilden dabei in den meisten Fällen die Partitionierung nach Hoare (siehe 4.2) oder jene nach Lomuto, die in Unterabschnitt 4.2.1 vorgestellt wird.

Schließlich gibt es noch die verschiedenen Strategien der Pivot-Wahl (siehe Unterabschnitt 4.1.3). Diese ändern jedoch nicht die Idee hinter dem Verfahren, sondern sorgen nur für einen Einfluss auf die Aufteilung und sollen daher in dieser Arbeit nicht im Vordergrund stehen.

Im Folgenden soll auf verschiedene Optimierungsansätze sowie Varianten des Quicksort-Verfahrens eingegangen werden. Die vorgestellten Optimierungen der Hauptmethode werden im Verlaufe der Arbeit immer wieder benutzt werden, während die vorgestellten Varianten der Partitionierung in erster Linie das intuitive Verständnis für die verschiedenen Ansätze stärken soll.

4.2.1. Lomutos Partitionierung

Im Laufe der Arbeit werden Quicksort-Varianten vorgestellt, welche teilweise hochgradig nicht trivial sind. Die Algorithmen dieser Verfahren lassen sich nicht auf einer Seite darstellen und werden in der Implementierung noch komplexer. Im Folgenden wird mit Lomutosort eine weitere Möglichkeit vorgestellt, in Quicksort zu partitionieren. Diese sorgt dafür, dass wir den Ansatz im nächsten Kapitel deutlich vereinfachen können (siehe Abschnitt 5.2). Eine solche Vereinfachung dient nicht nur dem verbesserten Verständnis der Leser, sondern wohlmöglich auch einer Steigerung der Effizienz in der Implementierung.

Lomutos einseitig gerichtete Form der Partitionierung basiert auf dem Ansatz, das Feld anstatt mit zwei sich überkreuzenden Zeigern (vgl. Hoare), mit zwei nach rechts laufenden Zeigern abzulaufen. Durch die beiden Zeiger werden drei Bereiche definiert in denen sich die Elemente befinden:

- Elemente kleiner als das Pivot-Element
- Elemente größer als das Pivot-Element
- nicht betrachtete Elemente

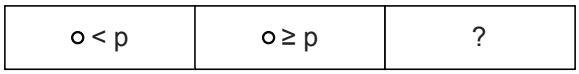


Abbildung 4.4.: Die drei Bereiche des Feldes mit Elementen kleiner als das Pivot-Element, größer als das Pivot-Element und nicht betrachtet. Das Symbol ∘ steht dabei für alle Elemente, welche die Bedingung erfüllen

Algorithmus 4.7 Lomutos Partitionierung

```
1: function LomutoPartition(A[ℓ,...,r])
2:     pivot ← A[r]
3:     i ← 1
4:     for j ← ℓ; j < r; j ← j + 1 do
5:         if A[j] < p then
6:             Swap A[i] and A[j]
7:             i ← i + 1
8:     Swap A[i] and A[n]
9:     return i
```

Der Algorithmus wurde in abgeänderter Form dem BlockQuicksort-Paper von Aumüller und Hass [4] entnommen. Im Vergleich zur Partitionierung nach Hoare in 4.2 werden hier ca. 50% der Zeilen gespart. In der Implementierung und durch Optimierungen kann die Ersparnis weiter erhöht werden.

Abbildung 4.5 zeigt eine Beispielsequenz in Lomutosort. Das aktuelle Pivot-Element ist grau hervorgehoben und wird mit dem Zeiger *pivot* indiziert. Des Weiteren geben die Zeiger *l* und *r* die aktuellen Bereiche an.

Korrektheit und Laufzeit

Die Korrektheit für die Partitionierung nach Lomuto ist schneller und einfacher gezeigt, als jene bei Hoares Partitionierung. Dies ist ein weiterer Grund, warum in den meisten Vorlesungen Lomutosort gegenüber Quicksort für die Vorstellung und Analyse präferiert wird. Da sie eine ähnliche Idee wie die von Quicksort hat,

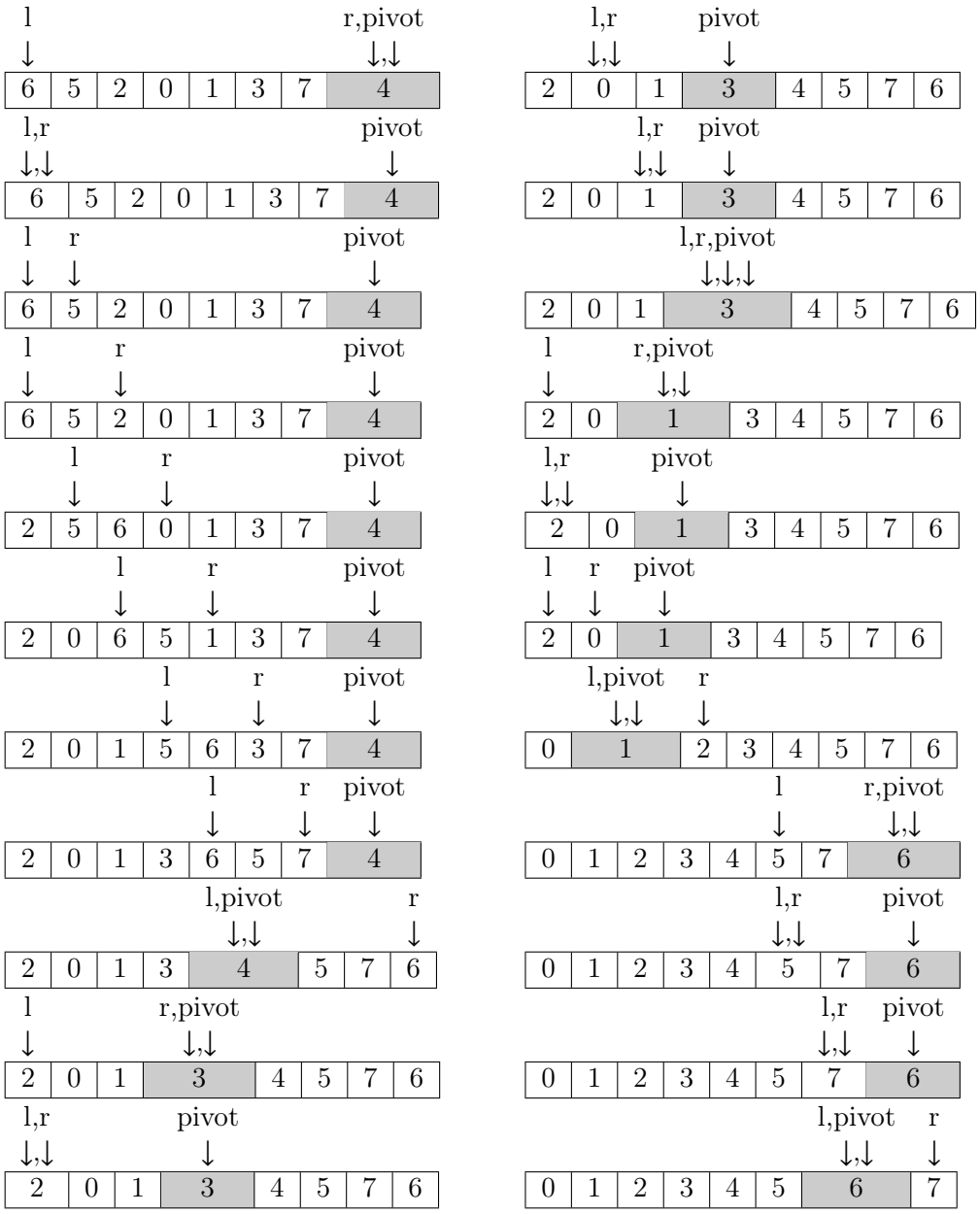


Abbildung 4.5.: Eine Beispielsequenz in Lomutosort

soll hier nur kurz die Idee vorgestellt werden. Dieses Mal brauchen wir das Feld nur aus einer Richtung zu iterieren, dabei soll die Invariante lauten: $\forall x \in [\ell, i]$: $A[x] \leq Pivot, \forall x \in [i, j] : A[x] \geq Pivot$. In jedem Schritt wird j auf ein neues Element gesetzt, dieses wurde noch nicht betrachtet und wird dann entweder in den ersten Bereich mit Elementen kleiner gleich dem Pivot-Element oder in den zweiten Bereich mit Elementen größer gleich dem Pivot-Element getauscht. Somit ist die Invariante wieder erfüllt. Die Terminierung der Partitionierung folgt automatisch beim Ende der For-Schleife.

Es ist nicht schwer zu sehen, dass die Partitionierung nach Lomuto in der Laufzeit nicht mit der Partitionierung nach Hoare mithalten kann. Während bei Hoare, sowohl Elemente welche kleiner, als auch Elemente welche größer als das Pivot-Element sind auf der richtigen Seite landen können ohne bewegt zu werden (die beiden Seiten werden gleichzeitig vom Rand aus aufgebaut), kann dies bei Lomutosort nur für Elemente passieren, welche größer als das Pivot-Element sind. Die Elemente, welche kleiner als das Pivot-Element sind, müssen ein weiteres Mal getauscht werden. Des Weiteren sorgt eine Vertauschung in Hoares Version dafür, dass zwei Elemente auf der richtigen Seite landen, während in Lomutosort nur ein neues Element auf die richtige Seite gebracht wird.

Theorem 4.8. *(Laufzeit der Lomuto Partitionierung)*
Sei N die Länge der Eingabe mit $N = r - \ell + 1$. Die Laufzeit der Partitionierung nach Lomuto liegt dann in $O(N)$.

Beweis:

Abbildung 4.6.: Instruktionen in Lomutosort

Zeile	*Kosten der Instruktionen*
$2-3$	$O(1)$
4	$O(r-\ell-1)$
	$\cdot \{ \boxed{5-7} \mid \boxed{1+1+3 = O(1)} \}$
$8-9$	$3+1 = O(1)$

Insgesamt: $O(1) + O(r-\ell-1) \cdot O(1)$

Ingesamt: $O(r-\ell)$ Kosten, damit kann die Partitionierung durch $max_{0 \leq \ell \leq r \leq N}\{r-\ell\}$ mit $O(N)$ abgeschätzt werden. □

Tatsächlich kann man einem Zitat von Aumüller und Hass [4] entnehmen: „Im Durchschnitt, wo der Durchschnitt über alle $n!$ Permutationen der Menge {1,...,n} genommen ist, braucht Lomutosort drei mal mehr Vertauschungen als Hoares Partitionierung und scannt 50% mehr Elemente.“ Ein weiterer Nachteil von Lomutosort ist eine hohe Anfälligkeit gegenüber Duplikaten. In Abschnitt 5.2 werden wir jedoch sehen, dass Lomutos Partitionierung besser und robuster wird wenn man ein zweites Pivot-Element hinzunimmt.

4.2.2. Insertionsort als Basis-Fall

Hoare hat bereits 1962 in seiner Abhandlung vorgeschlagen, die Sortiermethode zu wechseln, sobald das Teilproblem klein genug wird. Seine Aussage begründet sich daraus, dass die Zusatzkosten für die Partitionierung bei kleineren Teilproblemen in Relation zur Problemgröße zu hoch sind. Eine etablierte Sortiermethode für den Basisfall ist Insertionsort (siehe Unterabschnitt 3.5.1).

Dabei gibt es zwei Möglichkeiten, die Sortiermethode als Basis-Fall zu nutzen:

Algorithmus 4.9 Quicksortmit Insertionsort

```
1: function QUICKSORT(A[ℓ,...,r])
2:     if r − ℓ > THRESHOLD then
3:         choosePivot(A[ℓ,...,r])
4:         cut ← partition(A,ℓ,r)
5:         Quicksort(A,ℓ,cut − 1)
6:         Quicksort(A,cut + 1,r)
7:     else
8:         Insertionsort(A[ℓ,...,r])
```

Im ersten Fall wird der Basisfall in jedem Teilproblem welches klein genug ist aufgerufen.

Algorithmus 4.10 Quicksortmit Insertionsort

```
1: Quicksort(A[0,...,|A| − 1])
2: Insertionsort(A[0,...,|A| − 1])
```

Im zweiten Fall wird der Basis-Fall erst nach Abschluss aller Rekursionsaufrufe angewandt. Die beiden Fälle haben dabei einen marginalen Trade-Off zwischen Cache-Fehlern und zusätzlichen Vergleichen.

4.2.3. Introsort

Quicksort hat eine Worst-Case Laufzeit von $\Theta(N^2)$. Diese tritt auf, falls das Array mehrmals in ungleiche Partitionsgrößen aufgeteilt wird (siehe Abbildung 4.3). Es gibt Versuche, diese Beschränkung zu senken, so könnte man z.B. (linear) viel Zeit in die Suche eines Median Elementes investieren, da die Rekursionstiefe bei Aufteilung in gleiche Größen auf $\log N$ beschränkt ist, jedoch wird dadurch die durchschnittliche Laufzeit zu sehr beeinträchtigt, wodurch andere Sortierverfahren mit einer Laufzeit von $\Theta(N \log N)$ im schlechtesten Fall aber höheren Konstanten im Durchschnittlichen Fall attraktiver werden. Das Heapsort-Verfahren, welches bereits in Unterabschnitt 3.5.2 vorgestellt wurde erfüllt diese Kriterien, da es durchschnittlich zwar 2-5 mal langsamer als Quicksort ist, jedoch eine Laufzeit von $\Theta(N \log N)$ im schlechtesten Fall hat. Mit Introsort hat Musser [28] eine Lösung vorgestellt, in der er beide Algorithmen Quicksort und Heapsort kombiniert. Dafür legt man eine

Schranke für die Rekursionstiefe fest und sobald diese in Quicksort überschritten wird, wechselt man zu Heapsort. Die Schranke, ab der zu Heapsort gewechselt wird, muss laut Musser in $O(\log N)$ liegen. Jede gewählte Konstante führe dann zu einer Laufzeit von $O(N \log N)$, jedoch solle sie nicht zu klein gewählt werden, da sonst zu oft Heapsort aufgerufen wird und die Laufzeit schlechter wird. Im gleichen Paper wurde durch empirische Werte der Wert auf $2\lfloor \log N \rfloor$ festgelegt. Folgend ist Mussers Introsort-Algorithmus [28] dargestellt.

Algorithmus 4.11 Algorithm INTROSORT

```
1: function INTROSORT(A,f,b)
2:     INTROSORT_LOOP(A, f, b, 2⌊log(b − f)⌋)
3:     INSERTION_SORT(A, f, b)

1: function INTROSORT_LOOP(A,f,b,depth_limit)
2:     while b − f > size_threshold do          ▷ Schranke zu Insertionsort
3:         if depth_limit ← 0 then
4:             Heapsort(A, f, b)
5:             return
6:         depth_limit = depth_limit − 1
7:         p ← PARTITION(A, f, b, MEDIAN_OF_3(A[f], A[f +(b−f)/2], A[b−1]))
8:         INTROSORT_LOOP(A,p,b,depth_limit)
9:         b ← p
```

Durch die Beschränkung der Baumhöhe auf $O(\log N)$ und eine garantierte Heapsort Laufzeit von $O(N \log N)$ erhalten wir für Introsort eine Laufzeit von: $\Theta(N \log N)$ im schlechtesten Fall. Im weiteren Verlauf der Arbeit werden wir Introsort nicht als eigenständigen Sortieralgorithmus benutzen, vielmehr werden wir jedoch die Idee, an einem bestimmten Punkt zu Heapsort zu wechseln noch öfter sehen.

4.2.4. Sedgewicks und Yaroslavskiys Dual-Pivot Quicksort

Aufgrund der vielseitigen Optimierungsversuche von Quicksort seit der Veröffentlichung durch Hoare im Jahre 1962, gab es schon früh die Idee, den Partitionsprozess mit einem oder mehreren weiteren Pivot-Elementen durchzuführen, da dies die Anzahl der Datenzugriffe verringert. Die ersten Studien dazu beruhen auf der Arbeit von Sedgewick im Jahr 1975 [35] und Hennequin [13] im Jahr 1991. Diese ersten Untersuchungen waren jedoch nicht vielversprechend hinsichtlich möglicher Optimierungen und waren wahrscheinlich die Ursache, dass es 47 Jahre dauerte, bis im Jahr 2009 durch Yaroslavskiy, Bentley, and Bloch [6] ein Dual-Pivot Ansatz gefunden wurde, welcher sich etablieren konnte und mittlerweile der Standard Algorithmus in Oracles Java 7 Runtime ist. Um nachzuvollziehen, wieso sich der Algorithmus von Yaroslavskiy durchsetzen konnte, bzw. warum dies Sedgewicks nicht konnte, werden wir beide Verfahren in diesem Kapitel vorstellen und vergleichen.

[6] Im Zuge dieser Arbeit nur noch Algorithmus von Yaroslavskiy genannt.

Algorithmus 4.12 Dual-Pivot Quicksort

```
1: function DUAL-PIVOT QUICKSORT(A[1,...,n])
2:     if n > 1 then
3:         choosePivot(A[1,..,n])                    ▷ Pivots A[1] ≤ A[n]
4:         (i, j) ← partition(A[1,...,n])
5:         Dual-Pivot Quicksort(A[1,...,i-1])
6:         Dual-Pivot Quicksort(A[i+1,...,j-1])
7:         Dual-Pivot Quicksort(A[j+1,...,n])
```

Dual-Pivot Quicksort (4.12) stellt den Standard Quicksort-Algorithmus, erweitert für ein zweites Pivot-Element dar und stammt aus dem BlockQuicksort-Paper von Aumüller und Hass [4]. Die Methode *choosePivot* wählt dabei zwei Pivot-Elemente p und q. Diese werden im Allgemeinfall an die erste und letzte Stelle getauscht. Die Partitionierung teilt die Eingabesequenz danach in drei Bereiche, welche von Dual-Pivot Quicksort rekursiv abgearbeitet werden. Der große Vorteil von auf Quicksort basierenden Ansätzen, welche auf mehr als ein Pivot-Element zurückgreifen, ist die verringerte Anzahl an benötigten Datenzugriffen um die Eingabesequenz zu sortieren. Dies wird von Aumüller und Hass [4] mit folgendem Zitat unterstrichen: „Dieser Ansatz verbessert die Speicherzugriffsmuster drastisch, eine Verbesserung, welche durch keinen anderen Weg, wie zum Beispiel eine gute Pivot-Wahl, möglich ist."

Die Algorithmen

Folgend werden die Algorithmen von Sedgewick (4.13) und Yaroslavskiy (4.14) dargestellt und dem Leser näher gebracht. Dabei sollte man vor Augen haben, dass die Verfahren von Hoare, Sedgewick und Yaroslavskiy die Eingabesequenz jeweils unterschiedlich aufteilen. Dies ist in Abbildung 4.7 dargestellt. Der Schlüssel zum Erfolg von Yaroslavskiys Dual-Pivot Quicksort ist dabei die Idee, die benötigte Anzahl Vergleiche zu reduzieren, indem vorher überlegt wird, mit welchem Pivot-Element zuerst verglichen wird.

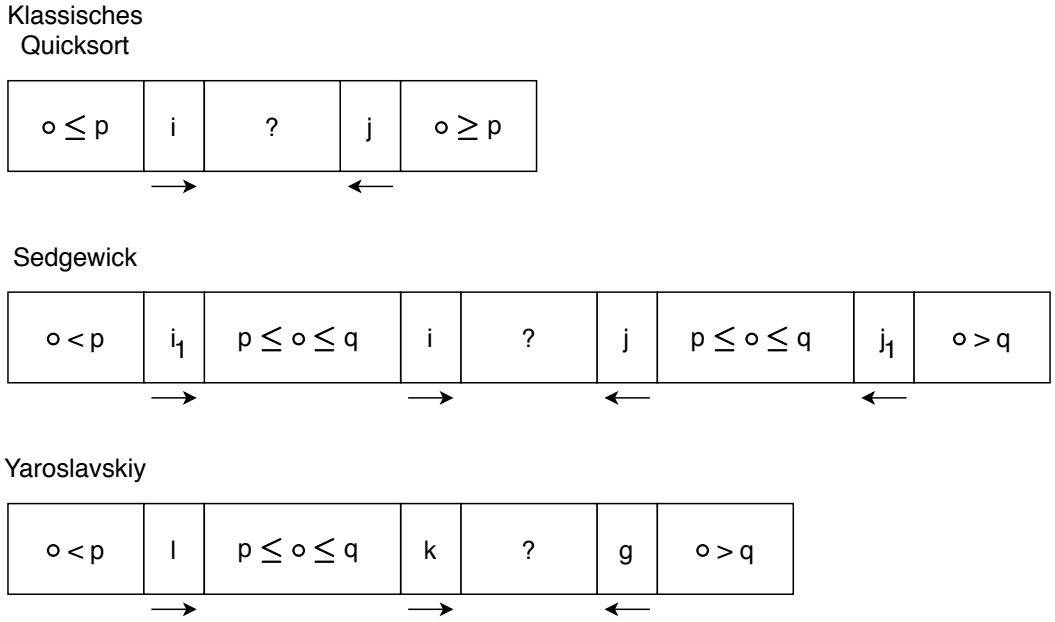


Abbildung 4.7.: Vergleich der Bereiche und Zeiger verschiedener Verfahren. Das Symbol ∘ steht dabei für alle Elemente, welche die Bedingung erfüllen. Die Abbildung stammt von Wild und Nebel [45].

Algorithmus 4.13 Sedgewicks Dual-Pivot Quicksort

```
1: function DUAL-PIVOT QUICKSORT(A,ℓ,r)   ▷ Wir Nehmen folgende Grenze an: A[0] = −infty.
2:     if r − ℓ ≥ 1 then
3:         i ← ℓ ; i1 ← ℓ ; j ← r; j1 ← r; p ← A[ℓ]; q ← A[r]
4:         if p > q then
5:             Swap p and q
6:         while true do
7:             i ← i + 1
8:             while A[i] ≤ q do
9:                 if i ≥ j  then
10:                    break outer while loop
11:                if A[i] < p then
12:                    A[i1] ← A[i];
13:                    i1 ← i1 + 1;
14:                    A[i] ← A[i1];
15:                i ← i + 1
               j ← j − 1
16:            while A[j] ≥ p do
17:                if A[i] > q then
18:                    A[j1] ← A[j]; j1 ← j1 − 1; A[j] ← A[j1]
19:                if i ≥ j then
20:                    break outer while loop
21:                j ← j − 1
22:            A[i1] ← A[j]; A[j1] ← A[i]
23:            i1 ← i1 + 1; j1 ← j1 − 1
24:            A[i] ← A[i1]; A[j] ← A[j1]
25:        A[i1] ← p; A[j1] ← q
26:        Dual-Pivot Quicksort(A, ℓ, i1 − 1)
27:        Dual-Pivot Quicksort(A, i1 + 1, j1 − 1)
28:        Dual-Pivot Quicksort(A, j1 + 1, r)
```

Der von Wild et. al. [45] vorgestellte Algorithmus stellt als erstes sicher, dass das linke Pivot-Element das kleinere ist und das rechte das größere. Danach teilt Sedgewicks Dual-Pivot Quicksort-Algorithmus die Eingabesequenz in drei Bereiche, wie in Abbildung 4.7 zu sehen. Dabei werden die Pivot-Elemente abgespeichert und ihre Plätze überschrieben.

Der erste Bereich wird abgegrenzt durch den Zeiger i_1, dort befinden sich die Elemente, welche kleiner als das linke Pivot-Element sind. Symmetrisch dazu befinden sich die Elemente, welche größer als das rechte Pivot-Element sind im letzte Bereich, dieser wird abgegrenzt durch den Zeiger j_1. In der Mitte befinden sich die noch nicht betrachteten Elemente, abgegrenzt durch die Zeiger i und j. Die verbleibenden zwei Bereiche zwischen i_1 und i bzw. j_1 und j halten die Elemente vor, welche sich zwischen den beiden Pivot-Elementen befinden. Dabei sieht man, dass die Bereiche zur Mitte hin aufgebaut werden, weswegen die zwei Zeiger i und j ähnlich wie bei dem Standard

Quicksort-Verfahren aufeinander zulaufen. Der Aufbau erfolgt folgendermaßen: Falls der Zeiger i auf ein Element zeigt, welches kleiner als das Pivot-Element ist, wird es entweder in den ersten Bereich getauscht und dann i erhöht, oder nur i erhöht, abhängig davon, ob es kleiner als das linke Pivot-Element ist. Damit wird entweder der linke Bereich vergrößert und der mittlere weiter in die Mitte geschoben oder nur der mittlere Bereich vergrößert in Richtung Mitte. Dies erfolgt bei der rechten Seite analog. Die aufeinander zulaufenden Zeiger i und j stoppen, falls i auf ein Element größer als das rechte Pivot-Element zeigt und j auf ein Element kleiner als das linke Pivot-Element zeigt. Falls solche zwei Elemente gefunden wurden, können diese in den ersten und letzten Bereich getauscht werden. Dabei ist das Verfahren hier jedoch anders als beim klassischen Quicksort, schließlich zeigen die Zeiger i und j auf Elemente welche sich im mittleren Bereich befinden und können nicht so einfach getauscht werden (sonst wären sie immer noch beide im mittleren Bereich). Deswegen hält der Algorithmus zwei weitere Zeiger i_1 und j_1 vor. Diese weiteren Zeiger grenzen den linken bzw. rechten Bereich von der Mitte ab und zeigen am Anfang auf das linke Pivot-Element am linken Rand und das rechte Pivot-Element am rechten Rand. Dadurch, dass diese beiden Felder überschrieben werden können, hat der Algorithmus immer zwei Plätze im Feld die überschrieben werden können und muss daher keinen aufwändigen Dreieckstausch mit Zwischenspeicherung machen. Die Werte an Position i und j werden nun nach j_1 und i_1 geschrieben und sowohl i_1 als auch j_1 inkrementiert bzw. dekrementiert. Danach werden i_1 und j_1 in i und j geschrieben, damit die Werte in i_1 und j_1 im nächsten Schritt wieder überschrieben werden können ohne etwas zwischenzuspeichern. Die Bereiche links und rechts haben sich um einen Wert vergrößert, der mittlere Bereich wurde von jeder Seite aus einen in die Mitte verschoben. Wie beim klassischen Quicksort wird, sobald sich die Zeiger i und j überkreuzt haben, die Schleife abgebrochen. Hier wird nicht das Pivot-Element in die Mitte getauscht, sondern die Pivot-Elemente werden an die Positionen i_1 und j_1 geschrieben.

Algorithmus 4.14 Yaroslavskiy's Dual-Pivot Quicksort

```
1: function DUAL-PIVOT QUICKSORT(A, ℓ, r) ▷ Wir Nehmen folgende Grenze
   an: A[0] = −infty.
2:     if r − ℓ ≥ 1 then
3:         p ← A[ℓ]; q ← A[r]
4:         if p > q then
5:             Swap p and q
6:         l ← ℓ + 1; g ← r − 1; k ← l
7:         while k ≤ g do
8:             if A[k] < p then
9:                 Swap A[k] and A[l]
10:                l ← l + 1
11:            else
12:                if A[k] > q then
13:                    while A[g] > q and k < g do
14:                        g ← g − 1
15:                    Swap A[k] and A[g]
16:                    g ← g − 1
17:                    if A[k] < p then
18:                        Swap A[k] and A[l]
19:                        l ← l + 1
20:            k ← k + 1
21:        l ← l − 1; g ← g + 1
22:        Swap A[ℓ] and A[l]
23:        Swap A[r] and A[g]
24:        Dual-Pivot Quicksort(A, ℓ, l − 1)
25:        Dual-Pivot Quicksort(A, l + 1, g − 1)
26:        Dual-Pivot Quicksort(A, g + 1, r)
```

Der zweite von Wild et. al. [45] vorgestellte Algorithmus stellt auch wieder sicher, dass das linke Pivot-Element kleiner als das rechte Pivot-Element ist. Danach wird die Eingabesequenz wieder in drei Bereiche aufgeteilt (siehe auch hier Abbildung 4.7). Im ersten Bereich befinden sich die Elemente, welche kleiner als das linke Pivot-Element sind, abgegrenzt durch den Zeiger l. Im mittleren Bereich befinden sich die Elemente, welche zwischen den beiden Pivot-Elementen liegen und rechts davon (immer noch im mittleren Bereich) die nicht betrachteten Elemente, abgegrenzt durch den Zeiger k. Im rechten Bereich befinden sich schließlich die Elemente, welche größer als das rechte Pivot-Element sind. Dieser Bereich ist durch den Zeiger g abgegrenzt.

Der Zeiger k wird in der Schleife hochgezählt und wandert nach rechts. In jeder Iteration gibt es nun drei Möglichkeiten:

1. Das neue Element ist kleiner als das linke Pivot-Element, dann tausche es in den ersten Bereich.
2. Das neue Element ist größer gleich dem linken Pivot-Element und kleiner gleich dem rechten Pivot-Element, dann muss nichts getauscht werden.

3. Das neue Element ist größer gleich dem linken Pivot-Element und größer als das rechte Pivot-Element. Nun muss das gefundene Element in den dritten Bereich getauscht werden. Da der dritte Bereich auch an die noch nicht betrachteten Elemente grenzt, wird dort nach dem ersten neuen Element x gesucht, welches nicht in diesen Bereich gehört, also kleiner gleich dem rechten Pivot-Wert ist. Anschließend werden die beiden Elemente getauscht, eventuell muss nun x noch einmal mit einem Element aus dem ersten Bereich getauscht werden, da x wieder kleiner als das linke Pivot-Element sein könnte.

Abschließend wird das linke Pivot an die linke Grenze der Mitte und das rechte Pivot an die rechte Grenze der Mitte getauscht.

Der Vergleich zwischen Sedgewicks Partitionierung und Yaroslavskiys Partitionierung

	Vergleiche	Vertauschungen
Classic Quicksort	$2(N+1)\mathcal{H}_{N+1} - \frac{8}{3}(N+1)$ $\approx 2N\ln N - 1.51N + \mathcal{O}(\ln N)$	$\frac{1}{3}(N+1)\mathcal{H}_{N+1} - \frac{7}{9}(N+1) + \frac{1}{2}$ $\approx 0.33N\ln N - 0.58N + \mathcal{O}(\ln N)$
Sedgewick	$\frac{32}{15}(N+1)\mathcal{H}_{N+1} - \frac{856}{225}(N+1) + \frac{3}{2}$ $\approx 2.13N\ln N - 2.57N + \mathcal{O}(\ln N)$	$\frac{4}{5}(N+1)\mathcal{H}_{N+1} - \frac{19}{25}(N+1) - \frac{1}{4}$ $\approx 0.8N\ln N - 0.30N + \mathcal{O}(\ln N)$
Yaroslavskiy	$\frac{19}{10}(N+1)\mathcal{H}_{N+1} - \frac{711}{200}(N+1) + \frac{3}{2}$ $\approx 1.9N\ln N - 2.46N + \mathcal{O}(\ln N)$	$\frac{3}{5}(N+1)\mathcal{H}_{N+1} - \frac{27}{100}(N+1) - \frac{7}{12}$ $\approx 0.6N\ln N + 0.08N + \mathcal{O}(\ln N)$

Tabelle 4.1.: Daten eines theoretischen Laufzeit-Vergleiches von Wild und Nebel [45] zwischen den Dual-Pivot Verfahren von Sedgewick und Yaroslavskiy und klassischem Quicksort. Die Ausgleichsterme der Harmonischen Zahl $\mathcal{H}$ weichen hier etwas von der Definition der Harmonischen Zahl in dieser Arbeit (siehe Abschnitt 3.2) ab, dies macht asymptotisch jedoch keinen Unterschied

Die Tabelle 4.1 vergleicht die Verfahren Quicksort, Sedgewicks Dual-Pivot Quicksort und Yaroslavskiys Dual-Pivot Quicksort anhand der Kriterien Vergleiche und Vertauschungen. Man sieht sofort, wieso Sedgewick 1975 Dual-Pivot Quicksort nicht als Verbesserung angesehen hat. Sein vorgestellter Algorithmus ist sowohl im Aspekt der Vergleiche, als auch in jenem der Vertauschungen schlechter als das Quicksort-Verfahren. Des Weiteren kann man sehen, dass zwar die theoretischen Vergleiche in dem Ansatz von Yaroslavskiy besser sind als die in Quicksort, dafür jedoch doppelt so viele Vertauschungen benötigt werden. Da die Vertauschungen jedoch sowieso von den Vergleichen dominiert werden, selbst bei einem Dreieckstausch mit 3 benötigten Instruktionen pro Vertauschung, ist mit Yaroslavskiys Dual-Pivot Quicksort aus theoretischer Sicht Verbesserung gegenüber Quicksort gefunden. Ob sich das Verhalten im Praxistest wiederfindet bleibt zu zeigen und ist abhängig von der Effizienz mit der Vertauschungen und Vergleiche ausgeführt werden können.

Begründung zur Überlegenheit der Yaroslavskiys Partitionierung

Wie schon im Laufe dieser Arbeit öfter gesehen, können kleine Anpassungen und Optimierungen große Auswirkungen auf die Laufzeit haben. Auch hier ist dies der Fall, denn die Anzahl der Vergleiche von Elementen mit den Pivot-Elementen hängt sowohl von den Pivot-Elementen selbst ab, als auch davon, mit welchem Pivot-Element wir die Elemente zuerst vergleichen. In dem Artikel „Durchschnittliche Fallanalyse von Dual-Pivot Quicksort in Java 7“ von Wild et. al. [45] wurde gezeigt, dass die Partitionierung Asymmetrien in der Element-Verteilung bei eben solchen Vergleichen ausnutzen kann. Während Quicksort mit einem einzelnen Pivot-Element immer $N-1$ Vergleiche braucht, hängt dies bei Dual-Pivot Quicksort davon ab, mit welchem Pivot-Element das zu klassifizierende Element zuerst verglichen wird. Es ist klar, dass für Elemente welche zwischen p und q liegen immer zwei Vergleiche gebraucht werden. Bei Elementen welche kleiner als p bzw. größer als q sind ist dies nur der Fall, falls diese zuerst mit dem jeweils anderen Pivot-Element verglichen wurden.

Am Anfang der Partitionierung stehen die zwei Pivot-Elemente p und q fest. Im Folgenden sollen die restlichen $N-2$ Elemente klassifiziert werden.

Definition 4.15. Sei S die Menge aller Elemente die kleiner als beide Pivot-Elemente sind, M die Menge aller Elemente die in der Mitte der beiden Pivot-Elemente liegen und L die Menge aller Elemente welche größer als beide Pivot-Elemente sind. Außerdem sei p das kleinere und q das größere Pivot-Element.
$S := \{1,..,p-1\}, M := \{p+1,...,q-1\}, L := \{q+1,...,N\}$

p								q
2	4	7	8	1	6	9	3	5
p	m	l	l	s	l	l	m	q
1	2	4	3	5	6	9	8	7
s	p	m	m	q	l	l	l	l

Abbildung 4.8.: Darstellung der Klassifizierung in S, M, L nach Wild et. al. [45]. Dabei gehört s zu S, m zu M und l zu L

Definition 4.16. Des Weiteren werden die Positionsmengen: S,M,L wie folgt definiert:
$\mathcal{S} := \{2,...,p\}, \mathcal{M} := \{p+1,...,q-1\}, \mathcal{L} := \{q,...,n-1\}$

	$\mathcal{S}$	$\mathcal{M}$	$\mathcal{M}$	$\mathcal{L}$	$\mathcal{L}$	$\mathcal{L}$	$\mathcal{L}$	
2	4	7	8	1	6	9	3	5
1	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*

Abbildung 4.9.: Darstellung der Positionsmengen nach Wild et. al. [45]

Mit der Vorarbeit aus 4.15 und 4.16 schauen wir nun auf Tabelle 4.2 welche von Wild et. al. [45] erarbeitet wurde. Zu sehen ist dabei eine hohe Asymmetrie. Im Durchschnitt über alle Permutationen sind s-type Elemente am wahrscheinlichsten in einer $\mathcal{S}$-Position und l-type Elemente am Wahrscheinlichsten in einer $\mathcal{L}$-Position. Dies macht sich 4.14 in Zeile 13+14 zu Nutzen. Neu betrachtete Elemente auf dem l-Zeiger werden zuerst mit dem linken Pivot-Element p verglichen und jene, auf dem g-Zeiger werden zuerst mit dem rechten Pivot-Element q verglichen. In 4.13 wird genau dieses Verfahren falsch herum angewandt: In Zeile 8 wird das Element auf dem i-Zeiger mit dem rechten Pivot-Element q verglichen, während in Zeile 16 das Element auf dem j-Zeiger mit dem linken Pivot-Element p verglichen wird. Dies erklärt den Unterschied in der Laufzeit zwischen Sedgewicks Partitionierung und Yaroslavskiys Partitionierung, trotz der gleichen Idee im Algorithmus. Für ausführlichere Erklärungen sei dabei auf die referenzierte Abhandlung von Wild und Nebel verwiesen.

	$\mathcal{S}$	$\mathcal{M}$	$\mathcal{L}$
s	$\frac{1}{6}(N-1)$	$\frac{1}{12}(N-3)$	$\frac{1}{12}(N-3)$
m	$\frac{1}{12}(N-3)$	$\frac{1}{6}(N-1)$	$\frac{1}{12}(N-3)$
l	$\frac{1}{12}(N-3)$	$\frac{1}{12}(N-3)$	$\frac{1}{6}(N-1)$

Tabelle 4.2.: Asymmetrien bei der Elementverteilung in Bezug auf Dual-Pivot Klassifizierung nach Wild et. al. [45].

Weitere Optimierungen

Dieser Abschnitt stellt weitere Optimierungen für das Dual-Pivot Verfahren vor. Erst werden zwei Varianten der Partitionierung auf Basis vorheriger Erkentnisse vorgestellt, anschließend wird eine Optimierung für die Hauptmethode vorgestellt.

Der analytische Ansatz aus Abschnitt 4.2.4, zunächst zu überlegen, mit welchem Pivot-Element zuerst verglichen werden soll kann weitergeführt werden um zwei Verfahren zu entwickeln, die mit $1,8N \ln N$ Vergleichen von Elementen mit den Pivot-Elementen auskommen. Die beiden Varianten wurden dabei von Aumüller und Hass in der Abhandlung „optimale Partitionierung für Dual-Pivot Quicksort“ [2] vorgestellt, die entsprechenden Beweise werden dort angeführt. An dieser Stelle wollen wir auf den Beweis verzichten und auf die Abhanlung verweisen.

Der erste Ansatz beruht auf einer Stichprobe: Sample vorab eine gewisse Anzahl Elemente in die drei Klassen S,M,L. Führe danach die Partition durch und prüfe für alle Elemente zuerst gegen das Pivot-Element ausgewählt durch: $|S| > |L|\ ?\ p : q$. Sei sz die vorab Sampleprobe der Größe $n^{\frac{2}{3}}$. Dann ist die durchschnittliche Anzahl an Vergleichen vom Algorithmus $1,8N \ln N + sz \ln N$. Das Problem ist hierbei ein großer Fehler-Term $sz \ln N$.

Für den zweiten Ansatz muss man sich Folgendes vorstellen: Angenommen man weiß in jedem Schritt wie viele Elemente in S und L noch nicht betrachtet wurden. Wir nennen diese Mengen $\hat{S}$ und $\hat{L}$. Dann wähle das Pivot-Element aus mit: $|\hat{S}| > |\hat{L}| \ ? \ p : q$. Man kann sich leicht vorstellen, dass dies optimal ist, da wir so die Wahrscheinlichkeit, mit dem anderen Pivot-Element zu vergleichen, minimal halten. Leider gibt es für uns keine effiziente Möglichkeit, die Mengen $\hat{S}$ und $\hat{L}$ zu bestimmen. Deshalb ist die Idee, andersherum zu argumentieren. Wir wissen in jedem Schritt, wie viele Elemente aus S und L wir jeweils schon gesehen haben. Diese Mengen seien $\bar{S}$ und $\bar{L}$, dann können wir das Pivot-Element wieder auswählen mit: $|\bar{S}| > |\bar{L}| \ ? \ p : q$. Mit der vorgestellten Methodik erreichen wir eine durchschnittliche Anzahl an Pivot-Vergleichen von $1,8N \ln N + O(N)$. Diese Anzahl an Vergleichen ist zwar theoretisch besser, die Annahme, dass Vergleiche mit den Pivot-Elementen die Laufzeit ausmachen, ist jedoch nicht unbedingt immer richtig. Außerdem muss bei diesen Verfahren entweder vorher ein Sample[7] erstellt werden oder die bereits klassifizierten Elemente müssen mitgezählt werden. Dadurch sind die Verfahren in der Praxis nicht unbedingt besser als Yaroslavskiys Dual-Pivot Quicksort. Trotzdem sollte klar geworden sein, wie es möglich ist, die Partitionierung in Dual-Pivot Quicksort zu optimieren.

Laut Aumüller und Hass [4] kann auch der in 4.12 vorgestellte Ansatz für die Hauptmethode von Dual-Pivot Quicksort für Duplikate in der Eingabe optimiert werden. Man nehme an, dass mit $A[i] = A[j]$ die Pivot-Elemente einen gleichen Wert haben. Dann befinden sich im Teilbereich $A[i+1, ..., j-1]$ nur Elemente mit gleichem Wert wie die Pivot-Elemente, der Aufruf in Zeile 6 kann sich also erspart werden. Die Optimierung könnte wie folgt umgesetzt werden:

```
1: if A[i] != A[j] then
2:     Dual-Pivot Quicksort(A[i+1,...,j-1])
```

Mit passender Implementierung für *choosePivot*, *partition* und Dual-Pivot Quicksort erhalten wir somit einen nach 3.10 glatten Algorithmus.

4.2.5. Multi-Pivot Quicksort

Ursprünglich wurden nur wenige Multi-Pivot Quicksort-Varianten erforscht bzw. Varianten, welche zwar mehr als ein Pivot-Element aber immer noch nicht viele benutzten. Dies lag daran, dass diese in zwei unabhängigen Dissertationen als unpraktisch klassifiziert wurden. Dies änderte sich, als 2009 Yaroslavskiys Dual-Pivot Quicksort zum Standard in Java 7 wurde. Dies führte zu einem Aufschwung in diesem Themengebiet und könnte ein Grund dafür gewesen sein, dass Kushagra et. al. in 2014 eine noch schnellere Variante vorstellen konnten: Multi-Pivot Quicksort mit drei Pivot-Elementen [21].

[7] Eine mögliche Übersetzung lautet „Stichprobe“

Durch das Hinzufügen von Pivot-Elementen zum Quicksort-Verfahren wird die Partitionierung deutlich komplizierter. Im Gegenzug werden (für bestimmte Anzahlen an Pivot-Elementen) weniger Elemente gescannt. Dies sorgt für eine Laufzeitverbesserung von Multi-Pivot Quicksort gegenüber Quicksort, da besseres Cache-Verhalten erreicht wird. Erprobte Anzahlen für Pivot-Elemente sind dabei 3 und 5, bei einem weiteren Anstieg werden die Verfahren wieder schlechter, da das Cache-Verhalten wieder abnimmt. So schreiben Aumüller et. al. 2016 in einer Abhandlung über die Relativität von Multi-Pivot Quicksort [3], das Verfahren sei für $k \geq 4$ ein offenes Problem, während Experimente durchgeführt von Kushagra et. al. bereits für $k = 7$ deutlich schlechtere Ergebnisse anzeigen.

Erhöht man die Anzahl der Pivot-Elemente trotzdem weiter, erhält man ab einer hinreichend großen Anzahl schließlich Samplesort Abschnitt 6.1, ein Sortierverfahren mit hoher potentieller Parallelität, wodurch die Laufzeit wieder besser wird.

In k-pivot Multi-Pivot Quicksort werden k Elemente der Eingabe gewählt und sortiert, so dass sie die Pivot-Elemente $p_1 \leq ... \leq p_k$ ergeben. Die Aufgabe ist es dann, die Eingabe in die entstehenden $k + 1$ Gruppen aufzuteilen, dafür brauchen wir zwei weitere Elemente als Grenzen: $p_0 = -\infty$ und $p_{k+1} = \infty$. Gruppe i wird dann dargestellt durch A_i und besteht aus den Elementen $p_i \leq x_i \leq p_{i+1}$, $0 \leq i \leq k$. Diese Gruppen werden dann wieder, jeweils rekursiv, sortiert. Die entstehenden Bereiche sind in Abbildung 4.10 zu sehen.

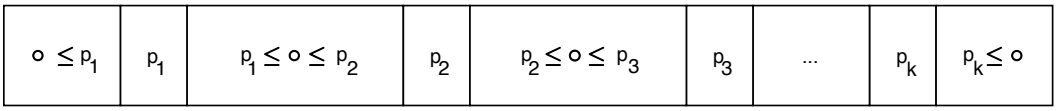


Abbildung 4.10.: Zeiger und Bereiche in Multi-Pivot Verfahren nach Aumüller et. al. [3]. Das Symbol ∘ steht dabei für alle Elemente, welche die Bedingung erfüllen

Wie wir bereits bei Sedgewicks Dual-Pivot Quicksort (siehe 4.13) gesehen haben, kann es bei Quicksort-Verfahren, welche mehr als ein Pivot-Element benutzen, hilfreich sein, die *swap*-Methode durch eine Rotation zu ersetzen. Im Anhang wird ein solcher, von Aumüller et. al. [3] vorgestellter Algorithmus (1.1) dargestellt.

Als Beispiel für einen Multi-Pivot Quicksort-Algorithmus wird folgend der Fall mit drei Pivot-Elementen behandelt:

Algorithmus 4.17 3-Pivot Partition

Require: $A[\ell] < A[\ell+1] < A[r]$ are the three pivots

```
1: function PARTITION3(A,ℓ,r)
2:     a ← ℓ + 2,b ← ℓ + 2
3:     c ← r − 1,d ← r − 1
4:     p ← A[ℓ],q ← A[ℓ + 1], r ← A[r]
5:     while b ≤ c do
6:         while A[b] < q and b ≤ c do
7:             if A[b] < p then
8:                 Swap(A[a], A[b])
9:                 a ← a + 1
10:            b ← b + 1
11:        while A[c] > q and b ≤ c do
12:            if A[c] > r then
13:                Swap(A[c], A[d])
14:                d ← d − 1
15:            c ← c − 1
16:        if b ≤ c then
17:            if A[b] > r then
18:                if A[c] < p then
19:                    Swap(A[b], A[a]), Swap(A[a], A[c])
20:                    a ← a + 1
21:                else
22:                    Swap(A[b], A[c])
23:                Swap(A[c], A[d])
24:                b ← b + 1, c ← c − 1, d ← d − 1
25:            else
26:                if A[c] < p then
27:                    Swap(A[b], A[a]), Swap(A[a], A[c])
28:                    a ← a + 1
29:                else
30:                    Swap(A[b], A[c])
31:                b ← b + 1, c ← c − 1
32:    a ← a − 1, b ← b − 1, c ← c + 1, d ← d + 1
33:    Swap(A[ℓ + 1], A[a]), Swap(A[a], A[b])
34:    a ← a − 1
35:    Swap(A[ℓ], A[a]), Swap(A[r], A[d])
```

Der Algorithmus 4.17 wurde 2014 von Kushagra et. al. [21] vorgestellt. Wie der Name bereits verrät, arbeitet der Algorithmus mir drei Pivot-Elementen: Das kleinste (linke) Pivot-Element p, das mittlere Pivot-Element q und das größte (rechte) Pivot-Element r. Außerdem werden vier Zeiger benutzt. Zwei Hauptzeiger b und c die aufeinander zulaufen und nicht betrachtete Elemente klassifizieren, sowie Zeiger a und d, welche den linken Bereich (Elemente kleiner als das linke Pivot-Element) und rechten Bereich (Elemente größer als das rechte Pivot-Element) abgrenzen.

Das Vorgehen soll im Folgenden in drei Teile aufgeteilt werden, dabei werden der erste und zweite Teil wiederholt, bis sich die Hauptzeiger kreuzen, danach folgt der dritte Teil:
1. Teil (Zeile 6-15): Die zwei Hauptzeiger b und c bewegen sich aufeinander zu. Im Gegensatz zum klassischen Quicksort muss jedes Mal, wenn ein Element betrachtet wird bei dem der Zeiger weiterlaufen würde (für den linken Zeiger b bedeutet dies, dass das Element kleiner als das mittlere Pivot-Element ist, für den rechten symmetrisch dasselbe) überprüft werden, ob das Element kleiner als das linke, bzw. größer als das rechte Pivot-Element ist und eventuell vertauscht werden. Sobald zwei Elemente gefunden wurden, für die die Bedingung nicht gilt: $A[b] < q$ und $A[c] > q$ ist Teil 1 beendet und wir gehen über zu Teil 2 bzw. Teil 3.
2. Teil (Zeile 16-31): Falls sich die zwei Hauptzeiger b und c noch nicht gekreuzt haben, müssen nun die gefundenen Elemente $A[b]$ und $A[c]$ an die richtige Position gebracht werden. Dafür schauen wir uns vier Fälle an, welche in Abbildung 4.11 dargestellt sind.

Fall 1: Das linke Element ($A[b]$) ist größer als das rechte Pivot-Element (r) und das rechte Element ($A[c]$) ist kleiner als das linke Pivot-Element (p).

Fall 2: Das linke Element ($A[b]$) ist größer als das rechte Pivot-Element (r) und das rechte Element ($A[c]$) ist **nicht** kleiner als das linke Pivot-Element (p).

Fall 3: Das linke Element ($A[b]$) ist **nicht** größer als das rechte Pivot-Element (r) und das rechte Element ($A[c]$) ist kleiner als das linke Pivot-Element (p).

Fall 4: Das linke Element ($A[b]$) ist **nicht** größer als das rechte Pivot-Element (r) und das rechte Element ($A[c]$) ist **nicht** kleiner als das linke Pivot-Element (p).

3. Teil (Zeile 32-35): Falls sich die zwei Hauptzeiger b und c bereits gekreuzt haben, sind alle Elemente klassifiziert und in den richtigen Bereichen. Die Pivot-Elemente müssen an ihre richtige Position gebracht werden, so dass sie die Bereiche abgrenzen.

4.2.6. Single-Pivot Quicksort mit Duplikaten

Um Quicksort mit einem Pivot-Element in einen nach 3.10 glatten Algorithmus zu überführen, muss man mehr Arbeit leisten als bei Dual-Pivot Quicksort, da wir die Elemente deren Schlüssel identisch mit dem des Pivot-Elementes sind, irgendwo speichern müssen. Wegner hat dafür in seiner Arbeit „Quicksort für gleiche Schlüssel“ [44] vier verschiedene Möglichkeiten vorgestellt:

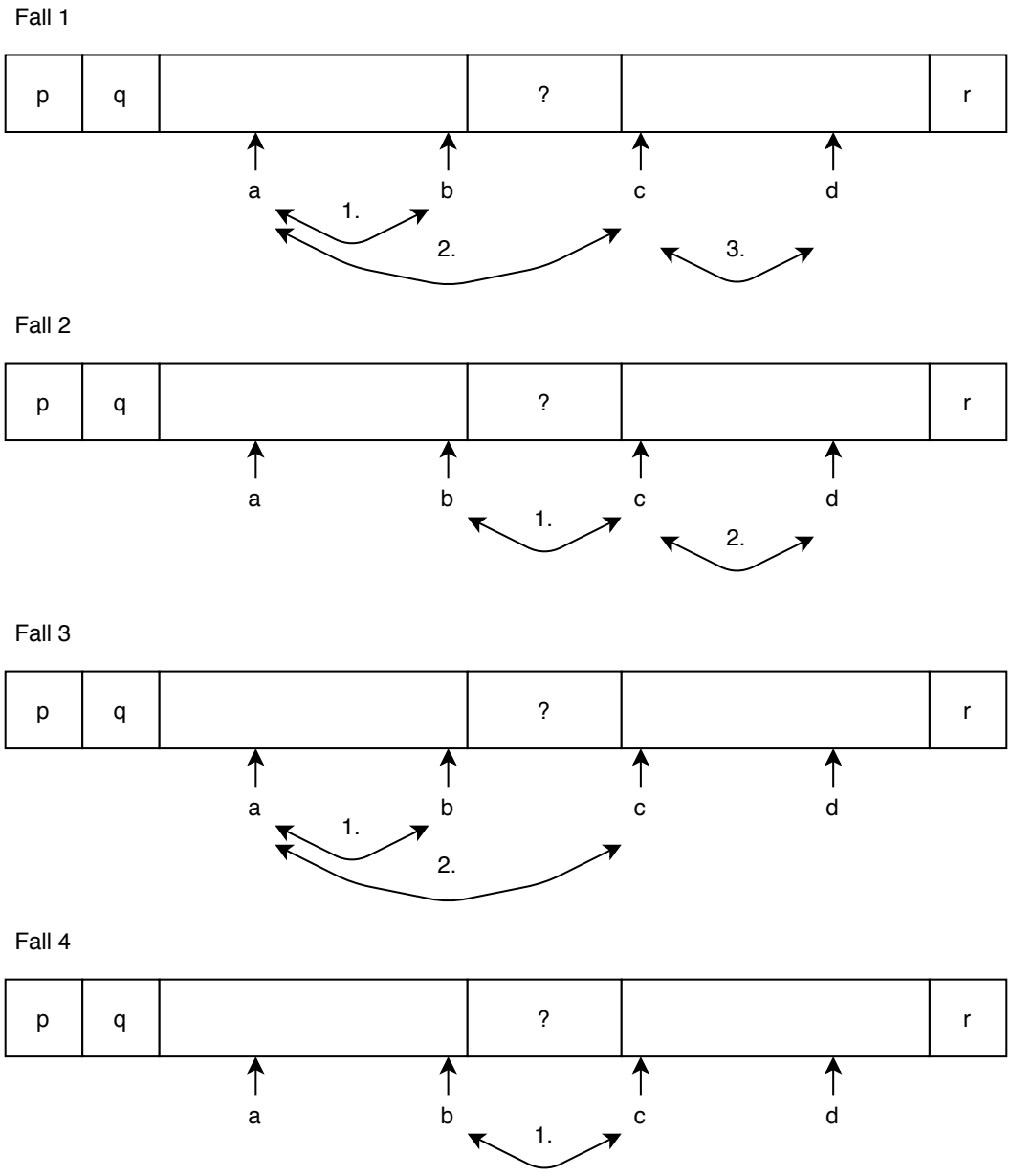


Abbildung 4.11.: Vorgehen in 3-Pivot *Partition*. Darstellung des 2. Teils, Fall 1 - 4.

- Elemente mit zum Pivot-Element identischem Schlüssel werden am Anfang und Ende der Eingabesequenz gesammelt und später zusammen mit dem Pivot-Element in die Mitte getauscht.
- Die Elemente werden nur am Anfang oder nur am Ende gesammelt.
- Die Elemente werden als wachsender Block durch die Eingabesequenz geschoben, bis die richtige Position erreicht wird.
- Die Elemente werden in einem zweiten Durchgang gesammelt.

Wir werden uns auf die erste Möglichkeit fokussieren und bearbeiten dafür den Algorithmus aus 4.2.

Algorithmus 4.18 Hoares Glatt-Partitionierung

```
1:  function partition(A[ℓ,...,r],pivot)
2:      x ← ℓ, y ← r
3:      while ℓ < r do
4:          while A[ℓ] <pivot do
5:              ℓ++
6:          while A[r] >pivot do
7:              r−−
8:          if ℓ < r then
9:              if (A[ℓ]) > pivot and A[r] < pivot then
10:                 swap(A[ℓ],A[r])
11:             if A[ℓ] > pivot and A[r] = pivot then
12:                 swap(A[ℓ],A[r])
13:                 swap(A[ℓ],A[x + 1])
14:                 x ← x + 1
15:             if A[ℓ] = pivot and A[r] < pivot then
16:                 swap(A[ℓ],A[r])
17:                 swap(A[ℓ],A[y − 1])
18:                 y ← y − 1
19:             if A[ℓ] = pivot and A[r] = pivot then
20:                 swap(A[ℓ],A[x + 1])
21:                 swap(A[r],A[y − 1])
22:                 x ← x + 1
23:                 y ← y − 1
24:             ℓ++
25:             r−−
26:     return ℓ
```

In 4.18 ist die Erste der vier von Wegner vorgestellten Möglichkeiten dargestellt. Sie wurde dem Buch „Algorithmen und Datenstrukturen“ von Ottmann et. al. [38] entnommen.

4.2.7. Weitere Varianten

Im Folgendem Abschnitt sollen dem interessierten Lesern weitere Anregungen und Ideen gegeben werden, auf welche Art und Weise man Quicksort abändern und verbessern könnte. Da auf diese Varianten jedoch nicht weiter eingegangen werden soll, werden hier nur die grundlegenden Ideen dargelegt.

Quickersort

Die Quicksort-Variante Quickersort von Scowen [34] basiert auf der Idee, die Eingabesequenz in zwei gleich große Sequenzen aufteilen zu wollen. Anstatt aufwändig den Median zu berechnen, wie im vorherigen Abschnitt 4.1.3 dieses Kapitels gesehen, wird hier davon ausgegangen, dass falls die Eingabe bereits einigermaßen sortiert ist, das mittlere Element genauso gut ist wie das Median-Element. Man kann leicht sehen, dass es sich bei diesem Ansatz um einen Spezialfall von Quicksort mit einer festen Pivot-Wahl handelt. Dieser wird für die meisten Eingabesequenzen schnell von Quicksort-Varianten mit $Median - Of - Three$ und ähnlichen Pivot-Wahl Strategien übertroffen, welche das Median Element nicht zu aufwändig berechnen.

Meansort

Der Meansort-Algorithmus von Motzkin [27] ist eine Variante von Quicksort, bei dem das Pivot-Element als Durchschnitt der Sequenz gewählt wird. Im ersten Durchlauf wird ein festgelegtes Pivot-Element, wie zum Beispiel das erste Element, benutzt. Bei jedem Partitionschritt wird nun zeitgleich der Durchschnitt beider Seiten mit berechnet und als Pivot-Element der nächsten beiden Partitionierungen benutzt.

CKSort

Der CKSort-Algorithmus von Cook [43], [8] ist eine Kombination aus Insertionsort und Quicksort mit anschließendem Zusammenführen, CKSort ist dabei optimiert für fast sortierte Sequenzen. Der Algorithmus vergleicht paarweise Elemente und verschiebt falsch geordnete Elementpaare in ein weiteres Array. Falls dabei ein Paar verschoben wird, wird das nächste Paar aus vorherigem und folgendem Element gewählt. Wenn die gesamte Sequenz durchlaufen ist, wird das zweite Array entweder mit Quicksort (falls dort mehr als 20 Elemente enthalten sind) oder mit Insertionsort sortiert. Die beiden Arrays werden zum Schluss zusammengeführt.

Dabei kann Quicksort, vor allem in Hinblick auf die Pivot-Bestimmung, durch Varianten ausgetauscht werden. Außerdem kann man an der Grenze zwischen Insertionsort und Quicksort schrauben.

Bsort

Der Bsort-Algorithmus von Wainwright [43], vorgestellt in 1985 hat vermutlich seinen Namen, da er die Idee von Bubblesort benutzt, nach jedem Schritt zu überprüfen ob die Sequenz bereits sortiert ist und optimiert Quicksort damit für fast sortierte oder fast umgekehrt sortierte Eingaben. Dabei wird das mittlere Element als Pivot-Element gesetzt, um eventuell vorhandene Ordnung nicht zu zerstören.

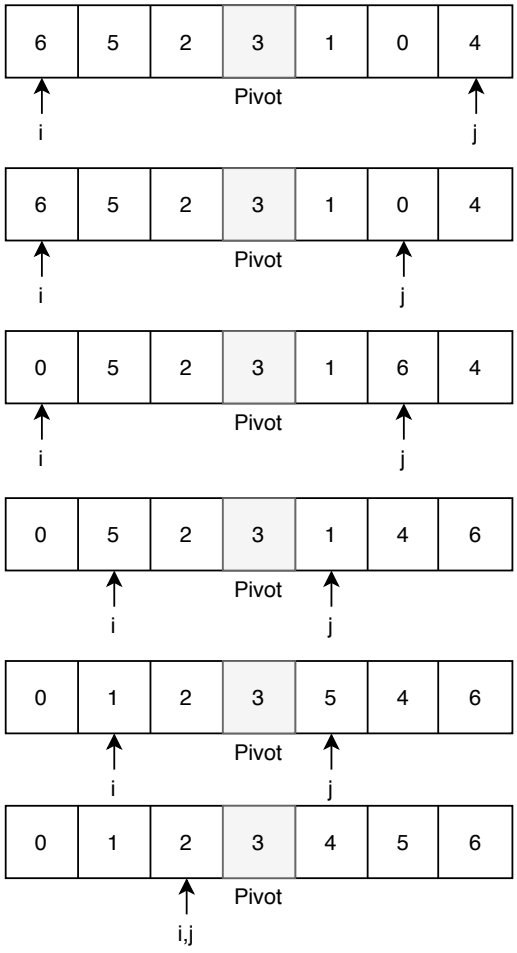


Abbildung 4.12.: Konstruiertes Beispiel für Bsort.

Im ersten Schritt werden die Zeiger i und j solange nach rechts bzw. links bewegt, bis sich ein Element findet, welches größer bzw. kleiner als das Pivot-Element ist. Anschließend werden die beiden Elemente getauscht. Soweit ist der Bsort-Algorithmus identisch zu Quicksort. Nach dem Tausch wird geprüft, ob das Element von den bisher betrachteten das größte bei Elementen links vom Pivot-Element oder das kleinste rechts vom Pivot-Element ist. Dafür vergleichen wir das Element mit dem linken bzw.

rechten Nachbarn. Falls getauscht werden muss wissen wir, dass die jeweilige Seite nicht mehr unbedingt sortiert sein muss. Danach wird wie bei normalem Quicksort der linke Zeiger inkrementiert und der rechte dekrementiert und wir beginnen von vorne. Die Partitionierung stoppt, sobald sich i und j begegnen. Sollten auf einer der beiden Seiten noch Elemente sein, wird das Pivot-Element entsprechend eingefügt. Sollte auf beiden Seiten kein Element mit seinem Nachbarn getauscht worden sein, ist die Eingabe bereits sortiert und wir können aufhören. Ansonsten fahren wir rekursiv fort.

Wainwright zeigt in seiner Publikation „Eine Klasse von Sortieralgorithmen basierend auf Quicksort“ [43], der Algorithmus Bsort benutzt nie mehr als $O(N \log N)$ Vergleiche und funktioniert am besten für sortierte oder umgekehrt sortierte Sequenzen mit $O(N)$ Vergleichen. Ein Verhalten für den schlechtesten Fall besitzt der Algorithmus nicht, durchschnittlich ist er dadurch besser als Standard-Quicksort.

5. BlockQuicksort

In Kapitel 2 wurde das Konzept der Sprungfehler vorgestellt, inklusive verschiedener Strategien diese zu vermeiden. Danach wurden in Kapitel 4 viele Optimierungen und Varianten für das Quicksort-Verfahren hinsichtlich unterschiedlicher Kriterien erarbeitet. In diesem Kapitel soll nun der Fokus auf dem Verfahren BlockQuicksort liegen. Dieser Algorithmus basiert auf Quicksort und wurde durch die vorgestellten Ansätze, Techniken und Ideen optimiert. Dafür werden die folgenden Schritte vorgestellt:

- Die Optimierung der Quicksort-Methode und ein erster sprungfreier Partitionierungsansatz
- Die Vorstellung der Block Partitionierung
- Die Bedeutung der Pivot-Wahl bei BlockQuicksort

Der vorgestellte Ansatz ruft dabei durchschnittlich $\epsilon \cdot C(N) + O(N)$ Sprungfehler hervor. Dabei ist ϵ von der Blockgröße abhängig, während $C(N)$ die durchschnittlichen Vergleiche in Quicksort bezeichnet und somit von der Pivot-Wahl abhängig ist. Es gilt $0 < \epsilon < 1$ für sinnvolle[1] Blockgrößen und $C(N) = \alpha N \log N$ mit $\alpha \geq 1$. Für eine Blockgröße von 128 und einer Pivot-Wahl nach der Median-Of-Three Strategie erhalten wir damit $\frac{1}{16} N \log N + O(N)$ Fehlvorhersagen im Schnitt. Nach einer Analyse von Edelkamp und Weiß [10] kann mit diesem Verfahren eine Verbesserung von bis zu 80% gegenüber der Quicksort-Implementierung von std::sorterreicht werden. Wiederum für (fast) sortierte oder (fast) verkehrt sortierte Eingaben ist die Performance schlechter.

5.1. BlockQuicksort

In der Laufzeit-Analyse sowie der Sprungfehler-Analyse von BlockQuicksort werden wir den Übergang in den Sonderfall Heapsort ignorieren. Diese Entscheidung richtet sich teilweise nach Edelkamp und Weiß [10], diese begründeten sie wie folgt: „Da die Wahrscheinlichkeit einer großen Rekursionstiefe klein ist, solange die Sequenz noch groß ist, ist diese Annahme keine Beschränkung.“ Mit einem kleinen Gedankenexperiment (**kein** Beweis) soll überlegt werden, warum diese Annahme keine Einschränkung ist.

Gedankenexperiment: Auslassen des Heapsort Sonderfalls in der Analyse

Wir benutzen die *Median − Of − Three* Strategie für die Pivot-Wahl, ab einer Rekursionstiefe von $2 \log_2 N + 3$ wechseln wir zu Heapsort und ab einer Sequenzgröße von 16 zu Insertionsort. In der Lektüre „Einführung in Algorithmen“ [9] von Cormen et. al. wird die Wahrscheinlichkeit für eine α zu $1-\alpha$ Aufteilung bei der Median-Of-Three

[1] Welche Blockgrößen „sinnvoll“ sind, werden wir in Kapitel 7 sehen.

Strategie behandelt (Exercise 7.4-6). Ohne Beweis soll für $0 < \alpha < 1$ gelten: Die Wahrscheinlichkeit im schlechtesten Fall eine α zu $1 - \alpha$ Aufteilung zu bekommen liegt bei $1 + 4\alpha^3 - 6\alpha^2$. Konkret bedeutet dies, dass eine Aufteilung von $\alpha = 0.25$ zu $1 - \alpha = 0.75$ im schlechtesten Fall, eine Wahrscheinlichkeit von 0.6875% hat. Führen wir diesen Fall weiter und betrachten die Sequenzen bei Aufteilungen von 3 zu 1, dann erhalten wir:

$$\begin{aligned} N \cdot (\frac{1}{4})^k &= 16 \\ \Leftrightarrow N &= 4^k \cdot 16 \\ \Leftrightarrow \log_4 N &= \log_4(4^k \cdot 16) \\ \Leftrightarrow \log_4 N &= k + \log_4 16 \\ \Leftrightarrow \log_4 N = \log_2 N \cdot \log_4 2 &= k + \log_4 16 \\ \Leftrightarrow k = 0.5 \cdot \log_2 N - 2 &< 2 \log_2 N + 3 \end{aligned}$$

Man kann sehen, dass für Aufteilungen welche immer 3:1 ausgehen, der Fall mit Übergang zu Heapsort gar nicht ausgelöst wird. Zusätzlich wissen wir, dass die Wahrscheinlichkeit eine schlechtere Aufteilung zu bekommen bei 31.25% liegt. Es ist klar, dass die Rekursionspfade, welche zu dem Heapsort Sonderfall führen eine Ausnahme sind und die betroffene Anzahl Elemente hinreichend klein ist um asymptotisch weder in der Laufzeit, noch in der Sprungfehleranalyse einen Unterschied zu machen. Dies soll uns in dieser Arbeit als Argumentation reichen, den Heapsort Sonderfall in der Analyse nicht zu betrachten.

5.1.1. Vorarbeit

Die Optimierung der Quicksort-Methode

Auf dem bisherigem Quicksort Ansatz, siehe 4.1, können nun die entsprechenden Optimierungen angewandt werden welche wir in den vorherigen Kapiteln eingeführt haben. Im Folgenden wird der Quicksort-Algorithmus erweitert (siehe 5.1, 5.2). Dafür wird Insertionsort als Basis-Fall und Heapsort als Begrenzung für den schlechtesten Fall benutzt. Auch wenn die Hauptmethode in diesem Fall noch nichts mit den besagten „Blöcken“ zu tun hat, ist dies doch die optimierte Version, welche für das BlockQuicksort-Verfahren benutzt und daher BlockQuicksort genannt wird. Die *THRESHOLDS* wurden nach den Erfahrungswerten von Edelkamp und Weiß [10] gewählt, die Grenze für Insertionsort wurde entsprechend auf kleiner 17 und die Grenze für Heapsort auf größer $2 \log N + 3$ gesetzt. Außerdem benutzen wir hier die Variante, den Basis-Fall Insertionsort in jeder Rekursion einzeln zu bearbeiten (Zeile 16), anstatt dies am Ende der Rekursion zu tun. Dies hat Vorteile für das Cache-

Verhalten, welche jedoch minimal sind. Als letztes wurde das klassische Quicksort-Verfahren in Zeile 6 bzw. 9 optimiert. Hier wird dafür gesorgt, dass die kleinere Teilsequenz immer zuerst bearbeitet wird. Zur Begründung reicht es, sich noch einmal die Worst-Case Sequenz in Abbildung 4.3 anzuschauen. Ohne Optimierung würde immer zuerst die linke Teilsequenz rekursiv bearbeitet werden und die rechte Sequenz auf dem Stack liegen. Bis der Stack das erste Mal kleiner würde, müsste er bereits $N-1$ verschiedene Sequenzen vorhalten. Falls wir nun jedoch immer die kleinere Sequenz zuerst bearbeiten, würde der Stack (in diesem extremen Beispiel) immer nur eine Sequenz vorhalten müssen, im generellen Fall jedoch maximal $\log N$ Sequenzen.

Algorithmus 5.1 Rekursiver BlockQuicksort

```
1:  function REKURSIVER BLOCKQUICKSORT(A[ℓ,...,r], depth)
2:      if r − ℓ > sizeTHRESHOLD and depth < depthTHRESHOLD then
3:          depth++
4:          choosePivot(A[ℓ,...,r])                ▷ Pivot-Element an letzter Stelle
5:          integer cut ← BlockPartition(A,ℓ,r)
6:          if cut − 1 − ℓ < r − cut + 1 then      ▷ kleinere Sequenz als erstes
7:              BlockQuicksort(A,ℓ,cut − 1,depth)
8:              BlockQuicksort(A,cut + 1,r,depth)
9:          else
10:             BlockQuicksort(A,cut + 1,r,depth)
11:             BlockQuicksort(A,ℓ,cut − 1,depth)
12:     else
13:         if r − ℓ > sizeTHRESHOLD then
14:             Heapsort(A,ℓ,r)
15:         else
16:             Insertionsort(A,ℓ,r)
```

Eine weitere Optimierung ist es, Rekursion zu vermeiden (Eine 1978 von Sedgewick eingeführte Optimierung [37]), wofür ein expliziter Stack eingeführt werden muss. Die vom Autor erstellte Optimierung in 5.1 dient daher ausschließlich dem intuitiven Verständnis der Zusammenführung vorheriger Optimierungen. Im Folgenden (siehe 5.2) wird der benutzte Algorithmus dargestellt, dieser ist um einen expliziten Stack erweitert.

Algorithmus 5.2 BlockQuicksort

```
 1: function BlockQuicksort(A[ℓ,...,r])
Require: stack sei ein Array mindestens der Größe depthTHRESHOLD · 2
 2:     integer current ← 0                          ▷ der Zeiger für den Stack
 3:     integer depth ← 0                            ▷ der Zähler für die Rekursionstiefe
 4:     do
 5:         if r − ℓ < sizeTHRESHOLD and depth < depthTHRESHOLD then
 6:             choosePivot(A, ℓ, r)                 ▷ Pivot-Element an letzter Stelle
 7:             integer cut ← BlockPartition(A, ℓ, r)
 8:             if cut − 1 − ℓ < r − cut + 1 then
 9:                 stack[current] ← cut + 1
10:                 stack[current + 1] ← r
11:                 r ← cut − 1
12:             else
13:                 stack[current] ← ℓ
14:                 stack[current + 1] ← cut − 1
15:                 ℓ ← cut + 1
16:             current + +
17:             depth + +
18:         else
19:             if r − ℓ > sizeTHRESHOLD then
20:                 Heapsort(A, ℓ, r)
21:             else
22:                 Insertionsort(A, ℓ, r)
23:             ℓ = stack[current]
24:             r = stack[current + 1]
25:             current − −
26:             depth − −
27:     while current > 0
```

Die dargestellten Algorithmen (5.1 und 5.2) sind vom Autor der Arbeit aus den vorherigen Anforderungen für die Optimierung des Quicksort-Verfahrens basierend auf dem BlockQuicksort-Paper von Edelkamp und Weiß [10] erstellt worden. Der Algorithmus geht wie folgt vor: Es wird ein Stack vorgehalten, auf welchem die Teilsequenzen gespeichert werden, die nicht direkt bearbeitet werden können. Solange der Stack nach dem ersten Durchlauf noch nicht leer ist, es also noch nicht bearbeitete Sequenzen gibt, tritt einer von zwei Fällen auf:

1. Die betrachtete Sequenz ist größer als die Schranke für den Basisfall und die Rekursionstiefe kleiner als die Schranke der Begrenzung für den schlechtesten Fall

2. Mindestens eine der beiden Bedingungen wurde verletzt.

Der erste Fall ist der Standardfall. Es wird ein Pivot-Element gewählt, die Eingabesequenz partitioniert und die kleinere Teilsequenz im nächsten Durchlauf betrachtet. Die Grenzen der größeren Sequenz werden dabei auf den Stack geschrieben. Im zweiten Fall wird auf der Eingabesequenz entweder Insertionsort als Basis-Fall oder Heapsort als Begrenzung für den schlechtesten Fall durchgeführt. Dadurch entstehen keine neuen Teilsequenzen, für den nächsten Schleifendurchlauf wird die Eingabesequenz vom Stack geholt, sofern der Stack noch Grenzen gespeichert hat, welche eine neue Sequenz definieren.

In Abbildung 5.1 ist die Auswahl-Strategie der Teilsequenz aus BlockQuicksort dargestellt. Die kleinere Teilsequenz wird als nächstes bearbeitet, die größere wird auf dem Stack zwischengespeichert.

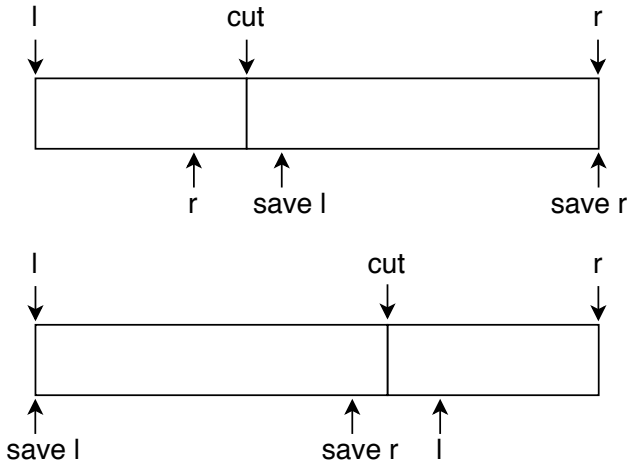


Abbildung 5.1.: Auswahl-Strategie der Teilsequenzen in BlockQuicksort

Korrektheit

Die Korrektheit von BlockQuicksort in 5.2 wird in dieser Arbeit nicht bewiesen, da daran nichts Neues gezeigt wird. Im Folgenden wird die Idee vorgestellt. Betrachtet wird der Standardfall (kein Heapsort) bis hin zur Grenze von Insertionsort. Der Korrektheitsbeweis ist analog zum Beweis von Quicksort in 4.6 mit vollständiger Induktion unter Berücksichtigung des Stacks durchzuführen. Man kann leicht zeigen, dass die Auswahl der nächsten Sequenz (Zeile 10 und 14) keinen Einfluss auf die Korrektheit hat. Beide Sequenzen werden auf jeden Fall bearbeitet und im jeweiligen Teilproblem wird nur auf den eigenen Bereich zugegriffen, wodurch die Reihenfolge der Bearbeitung keine Rolle spielt. Der Rest des Beweises folgt direkt aus der Korrektheit von Insertionsort und der Korrektheit von Heapsort.

Ein erster Sprungfreier Partitionsansatz

Ausgangspunkt der Partitionierung in BlockQuicksort war der schlanke Algorithmus Tuned Quicksort. Dies ist ein auf Quicksort basierendes Verfahren, welches die Partitionierung nach Lomuto benutzt und Sprungfehler vermeidet, indem vorherbestimmte Instruktionen benutzt werden. Das Verfahren wurde von Elmarsy et. al. entwickelt, der in 5.3 dargestellte Algorithmus wurde auf Grundlage der Forschung von Katajainen [19] erstellt.

Algorithmus 5.3 Tuned Quicksort

```
1:  function TUNED_QUICKSORT(A[ℓ,...,r])
2:      Pivot ← A[r]
3:      integer i ← ℓ
4:      for integer j ← ℓ, j < r, j++ do
5:          boolean smaller ← (A[j] < Pivot)
6:          integer delta ← smaller · (j − i)
7:          Swap(A[i], A[i + delta])
8:          i ← i + smaller
9:      Swap(A[i], A[r])
10:     return i
```

In dem Algorithmus wird die in Abschnitt 3.4.1 vorgestellte Strategie benutzt, Sprünge durch vorherbestimmte Instruktion zu vermeiden. Explizit wird hier die Instruktion *SETcc* benutzt welche ein Byte unter einer Bedingung setzt (siehe Zeile 5). Die Strategie soll hier wieder eher dem intuitiven Verständnis dienen, dieses Mal für den Einsatz vorherbestimmter Instruktionen, daher sollen Laufzeitverhalten und Korrektheit erst einmal nebensächlich sein. Trotzdem folgt eine kurze Einschätzung der Effizienz von Tuned Quicksort im Vergleich zu Lomutosort.

Abbildung 5.2.: Instruktionen in Tuned Quicksort

Zeile	*Kosten der Instruktionen*
$2-3$	$1+1=O(1)$
4	$r-\ell-1$ *Schleifendurchläufe* $=O(r-\ell-1))$
	abgeschätzt durch $max_{0\leq\ell\leq r\leq N}\{r-\ell-1\}=O(N)$
	$\cdot\{$ $5-8$ | $1+1+3+1=O(1)$ $\}$
$9-10$	$3+1=O(1)$

Insgesamt: $O(1)+O(N)\cdot O(1)=O(N)$

Erinnerung: Lomutosort (siehe Unterabschnitt 4.2.1) hat $O(N)$ Kosten. Tuned Quicksort schafft dies auch, ohne dabei jedoch Sprungfehler hervorzurufen. Auf den zweiten Blick sieht man jedoch einen Unterschied. Während bei Lomutosort der *Swap*-Aufruf nur dann ausgeführt wird, wenn er benötigt wird, so wird er bei Tuned Quicksort immer ausgeführt.

5.1.2. Block Partitionierung

Die Partitionierung spielt mit einem Faktor von $O(N)$ in der Laufzeit und den meisten hervorgerufenen Sprungfehlern eine entscheidende Rolle. Nachdem die Hauptmethode von Quicksort in Abschnitt 5.1.1 größtmöglich optimiert wurde, soll daher nun auch ein neuer Ansatz der Partitionierung vorgestellt werden. Ziel dabei ist es, eine Partitionierungsstrategie zu finden, welche eine ähnlich gute Laufzeit wie die Partitionierung nach Hoare oder Lomuto hat, dafür aber ohne Sprungfehler läuft. Eine solche Strategie wurde mit dem Algorithmus *BlockPartition* gefunden.

Um die Idee hinter der Block Partitionierung besser greifbar zu machen, wurde die Partitionierung nach Hoare einmal in 5.4 vereinfacht aber mit gleichem Vorgehen dargestellt. Die Block Partitionierung basiert nun auf dem Gedanken, in eben jenem Algorithmus (5.4) die Vergleiche (Zeile 3-4) von den Vertauschungen (Zeile 5) zu trennen. Dafür erstellen wir für eine konstante Blockgröße B zwei Puffer $offsets_L[0,...,B-1]$ und $offsets_R[0,...,B-1]$. Diese zwei Puffer halten die zu tauschenden Elemente vor, welche wie im originären Verfahren mithilfe zweier sich kreuzender Zeiger, jedoch ohne die Benutzung von Entscheidungssprüngen gefunden werden. An dieser Stelle wird auf dem Verfahren von Elmarsy et. al. aus Abschnitt 5.1.1 aufgebaut[2]. Da die Puffer nur eine konstante Größe von B haben, muss immer wieder zwischen dem Leeren der Puffer (Vertauschungen) und dem Füllen der Puffer (Vergleiche) gewechselt werden. Anschließend wird wie gewohnt das Pivot-Element an die richtige Stelle getauscht.

Algorithmus 5.4 Hoares Partitionierung aus 4.2 zusammengefasst

```
1: function PARTITION(A,ℓ,r,pivot)
2:     while ℓ < r do
3:         while A[ℓ] < pivot do ℓ++
4:         while A[r] > pivot do r--
5:         if ℓ < r then Swap(A[ℓ],A[r]); ℓ++; r--
6:     return ℓ
```

Durch dieses Trennen der Zeilen werden in der Hauptschleife die Entscheidungssprünge vermieden, welche auf Vergleichen basieren, dabei bleiben nur die einzelnen Entscheidungssprünge im Kontrollfluss, sowie jene für Insertionsort und Heapsort übrig. Im direkten Vergleich mit Quicksort bleibt die Anzahl der Vergleiche und Vertauschungen gleich, lediglich die Anzahl der in den Speicher geladenen Elemente erhöht sich um einen Faktor von 2, da jedes Element sowohl für den ursprünglichen Vergleich als auch für den Tausch in den Speicher geladen werden muss. Da wir jedoch kleine Puffer mit konstanter Größe benutzen ist hier die Chance hoch, dass die Elemente sich noch im Cache[3] befinden und daher nicht erneut geladen werden müssen.

[2] Auch wenn die Partitionierung dort auf der Basis von Lomuto dargestellt wird, ist die Idee für die Partitionierung auf der Basis von Hoare die gleiche.

[3] L1 Cache

Deswegen ist es essentiell, die Blockgröße nicht zu groß zu wählen. Da eine größere Blockgröße die Sprungfehler jedoch weiter verringert, ist dies ein Tuning-Parameter. Eine etablierte Wahl ist die Blockgröße 128.

Algorithmus 5.5 Block Partitionierung

```
1: function BlockPartition(A[ℓ,...,r])
2:     pivot ← A[r]
3:     r − −
4:     integer offsets_L[0,...,B − 1], offsets_R[0,...,B − 1]
5:     integer start_L, start_R, num_L, num_R ← 0
6:     while r − ℓ + 2 > 2B do
7:         if num_L = 0 then
8:             start_L ← 0
9:             for i = 0,...,B − 1 do
10:                offsets_L[num_L] ← i
11:                num_L ← num_L + (pivot ≥ A[ℓ + i])
12:        if num_R = 0 then
13:            start_R ← 0
14:            for i = 0,...,B − 1 do
15:                offsets_R[num_R] ← i
16:                num_R ← num_R + (pivot ≤ A[r − i])
17:        integer num ← min(num_L, num_R)
18:        for j = 0,...num − 1 do
19:            swap(A[ℓ + offsets_L[start_L + j]], A[r − offsets_R[start_R + j]])
20:        num_L ← num_L − num, num_R ← num_R − num
21:        start_L ← start_L + num, start_R ← start_R + num
22:        if num_L = 0 then
23:            l ← ℓ + B
24:        if num_R = 0 then
25:            r ← r − B
26:    scan and rearrange remaining Elements
27:    return pivot index
```

Der Algorithmus 5.5: *BlockPartition* wurde aus dem BlockQuicksort-Paper von Edelkamp und Weiß [10] entnommen. Im Folgenden ist die Struktur, mit der gearbeitet wird, dargestellt.

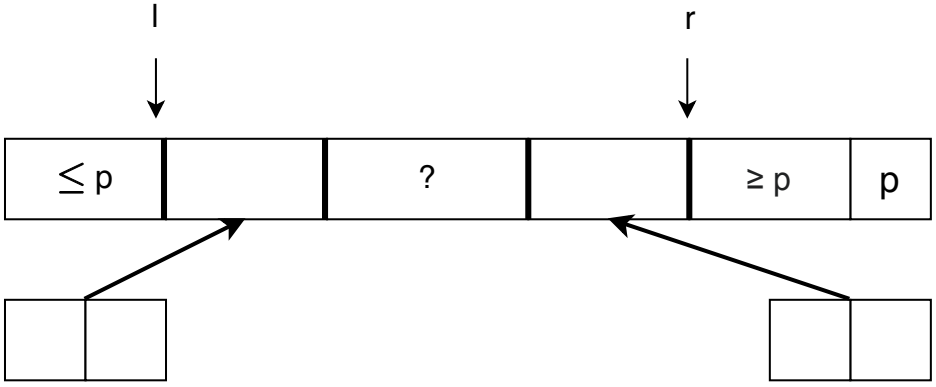


Abbildung 5.3.: Struktur mit der die Methode *BlockPartition* arbeitet

in Abbildung 5.3 sehen wir die bereits klassifizierten Bereiche, welche durch ℓ und r abgetrennt sind. Darauf folgen die beiden aktuell betrachteten Bereiche, eindeutig dargestellt durch die hervorgehobenen Trennlinien. Die beiden Puffer referenzieren dabei auf genau diesen Bereich. In der Mitte befinden sich die noch nicht betrachteten Elemente, gekennzeichnet durch das Fragezeichen.

Abbildung 5.4 zeigt eine Beispielsequenz in BlockQuicksort. Das aktuelle Pivot-Element ist grau hervorgehoben. Weiter geben die Zeiger l und r die aktuellen Bereiche an, ferner ist der aktuelle Pufferbereich in gelb und mit Trennlinien hervorgehoben. Die Zeiger i_L und i_R zeigen auf den ersten Index im linken bzw. rechten Puffer, welcher ein noch zu tauschendes Element referenziert.

5.1.3. Analyse von Block Partition

Korrektheit

Alle Beweise ignorieren den letzten maximal $2B$ großen Bereich, da die Zeile 24 ein Implementierungsdetail ist, welches in dieser Arbeit nicht betrachtet wird. Im Umfang dieser Arbeit wird angenommen, dass aus der Korrektheit des Algorithmus von Zeile 1-23, die Korrektheit des gesamten Algorithmus folgt.

In Abbildung 5.5 sind die Zeiger dargestellt, welche in den folgenden Beweisen benötigt werden. Die Zeiger *begin* und *end* geben den Bereich in der Sequenz an, auf den *BlockPartition* aufgerufen wird. Da der Zeiger *end* dabei auf das Pivot-Element zeigt, ist der zu klassifizierende Bereich durch $[begin, end-1]$ angegeben. Die aktuellen Blöcke sind mit $[\ell, \ell + B)$ und $(r - B, r]$ gesetzt. Auch wenn dies im Algorithmus nicht so definiert ist, werden die Zeiger *begin* und *end* für die Beweise als die Werte von ℓ und r definiert, mit denen die Methode aufgerufen wird.

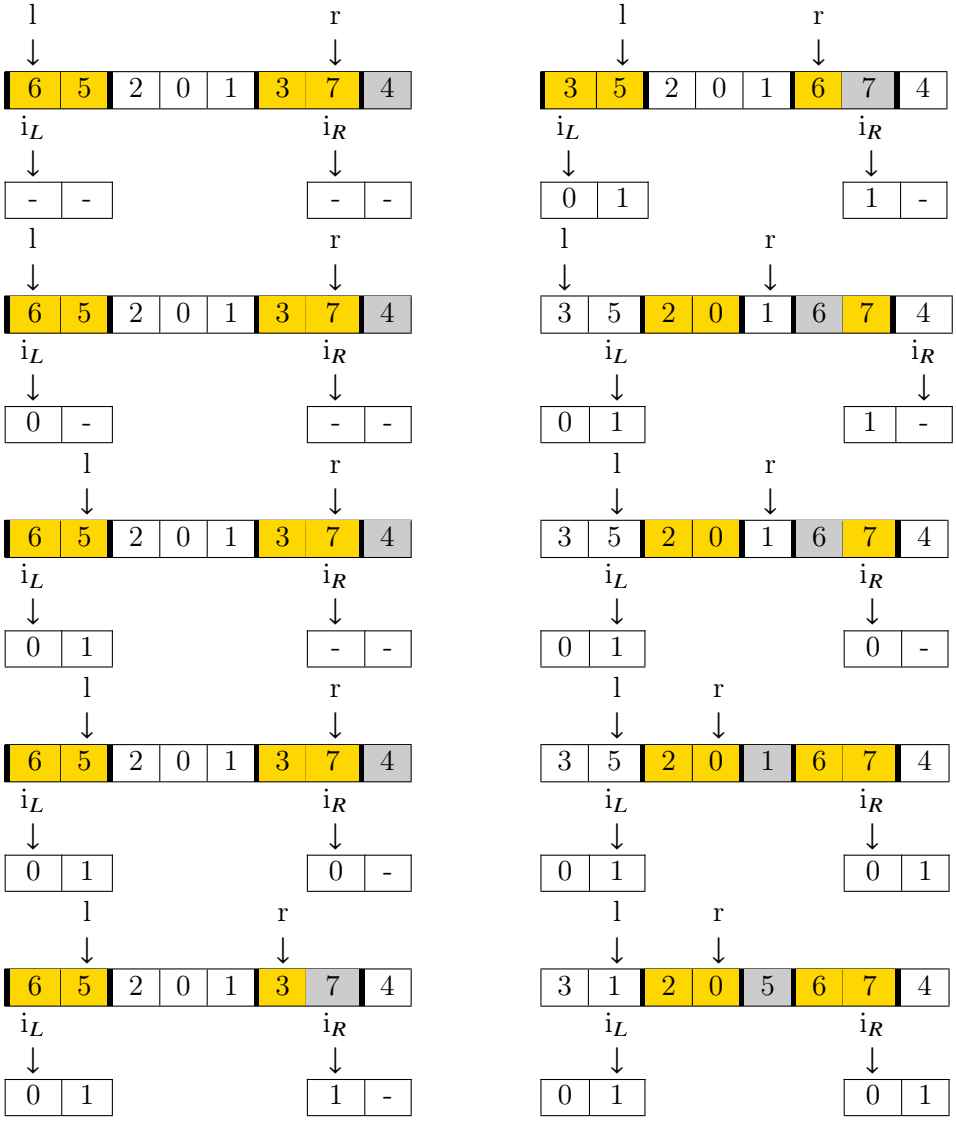


Abbildung 5.4.: Eine Beispielsequenz in Insertionsort

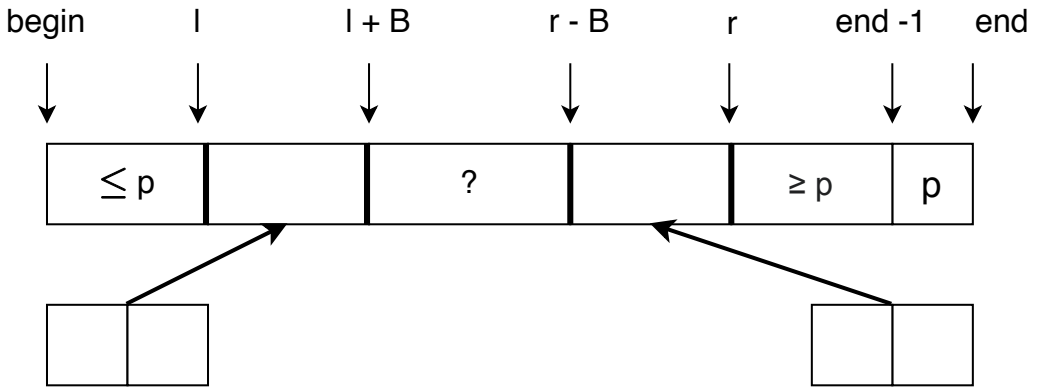


Abbildung 5.5.: Die Zeiger in Block Partition

Lemma 5.6. *Vor jedem Schleifendurchlauf von Zeile 6 ist mindestens einer der beiden Puffer leer.*

Beweis: Angenommen dies gilt nicht. Die Puffer-Zähler num_L und num_R geben die Anzahl der in den beiden Puffern richtigen Elemente an (falsche Elemente werden überschrieben und nicht mitgezählt, Zeile 10-11 und 15-16). Die Zeilen 11 und 16 sind die einzigen Zeilen in denen die Puffer-Zähler erhöht, also Elemente zu den Puffern hinzugefügt werden. Diese Zeilen müssen im ersten Durchlauf aufgerufen werden, da sonst beide Puffer im nächsten Durchlauf direkt leer werden und die Annahme widerlegt ist. O.B.d.A. ist beim nicht Erfüllen der Annahme (also in der Iteration in der keiner der beiden Puffer leer ist) num_L der kleinere Zähler, also der linke Puffer der Puffer mit weniger Elementen. Es gilt $num_L \leq num_R$, $num_L > 0$, $num_R > 0$ und mit Zeile 15 gilt $num = num_L$. Nun wird die Schleife in Zeile 18 $num = num_L$ mal ausgeführt, in jeder Iteration wird Zeile 20 ausgeführt und sowohl num_L als auch num_R werden dekrementiert. Dies ist die einzige Stelle, an der num_L und num_R dekrementiert werden. Beim Abbruch der Schleife (nach der num-ten Iteration) muss gelten $num_L = 0$. Es folgt, dass der linke Puffer leer ist. Da er nur in Zeile 11 gefüllt werden kann, muss der linke Puffer vor dem nächsten Schleifendurchlauf bereits leer sein. ↯ Widerspruch zur Annahme. Somit muss die Behauptung bereits gelten: vor jedem Schleifendurchlauf von Zeile 6 ist mindestens einer der beiden Puffer leer. □

Lemma 5.7. *Sei B die Blockgröße und N die Länge der Eingabe mit $N = r - \ell + 1$. In jedem Schleifendurchlauf erhöht sich der betrachtete Bereich um mindestens B. Dabei wird entweder ℓ um B erhöht und / oder r um B verringert.*

Beweis: Die Aussage folgt bereits direkt aus 5.6 und Zeilen 23, 25. Es gilt: mindestens einer der beiden Puffer leert sich in jedem Durchlauf. Falls ein Puffer leer ist, wird für diesen entsprechend am Ende der Iteration in Zeile 23 (für den linken

Puffer) $\ell+ = B$ gerechnet oder in Zeile 25 (für den rechten Puffer) $r- = B$ gerechnet. Da der betrachtete Bereich sich aus $[begin, \ell)$ und $(r, end - 1]$ zusammensetzt, hat sich dieser um mindestens B vergrößert. □

Schleifeninvariante 5.8. *Vor jedem Durchlauf der Schleife in Zeile 6 gilt:* $\forall x \in [begin, \ell) : x \leq Pivot$ *und* $\forall y \in (r, end - 1] : y \geq Pivot$.

Initialisierung:
Vor dem ersten Durchlauf gilt $end - 1 = r$ und $\ell = begin$. Die Bereiche $[begin, begin)$ und $(end - 1, end - 1]$ sind somit leer, die Bedingung ist erfüllt.

Aufrechterhaltung:

Zeile 7-11 sammelt alle Elemente x im Bereich $[\ell, \ell + B)$ für die gilt: $x \geq Pivot$. Diese gesammelten Elemente werden im linken Puffer indiziert. Es kann sein, dass ein Element im Puffer steht, welches nicht vertauscht werden soll. Diese falschen Elemente entstehen durch Zeile 10. Dort wird jedes der gelesenen Elemente in den Puffer geschrieben aber Elemente, die dort nicht hineingehören, da sie die entsprechende Bedingung $pivot \geq A[\ell + i]$ nicht erfüllen, werden in der nächsten Iteration überschrieben. Wenn jedoch in der letzten Iteration ein falsches Element betrachtet wird, bleibt es im Puffer. Dafür wird der Zähler num_L benutzt, dieser gibt die Anzahl der gesammelten Elemente an. Falsche Elemente im Puffer werden also nicht betrachtet. Elemente y im Bereich $[\ell, \ell + B)$, welche sich nicht im Puffer befinden oder von num_L nicht betrachtet werden, sind also bereits richtig klassifiziert als $y \leq Pivot$.

Symmetrisch folgt dasselbe für die Zeilen 12-16 und den rechten Puffer bzw. den rechten Bereich $[r - B, r]$.

Alle Elemente in den Puffern, welche mithilfe von num_L oder num_R berücksichtigt werden, sind also im falschen Bereich, die restlichen Elemente sind bereits im richtigen Bereich.

Nun gibt es drei Fälle (der vierte Fall $\ell \leftarrow \ell$, $r \leftarrow r$ kann nach 5.7 nicht auftreten).

Fall 1 $\ell \leftarrow \ell + B$, $r \leftarrow r$

Fall 2 $\ell \leftarrow \ell$, $r \leftarrow r - B$

Fall 3 $\ell \leftarrow \ell + B$, $r \leftarrow r - B$

1. Fall: $\ell \leftarrow \ell + B$, $r \leftarrow r$
l wurde um B erhöht. Damit die Schleifeninvariante aufrecht erhalten bleibt muss gezeigt werden: $\forall x \in [\ell, \ell + B) : x \leq Pivot$, denn dann gilt mit der Erfüllung der

Invariante für den letzten Schleifendurchlauf: $\forall x \in [begin, \ell + B) : x \leq Pivot$ und $\forall y \in (r, end - 1] : y \geq Pivot$.

Der Zeiger ℓ kann nur durch Zeile 23 erhöht worden sein, dafür muss die Bedingung $num_L = 0$ erfüllt sein. Wir können außerdem annehmen, dass direkt vor Zeile 20 $num_L! = 0$ galt, sonst wäre in Zeilen 9-11 kein falsches Element gefunden worden und die Invariante würde direkt gelten. Dadurch, dass in Zeile 22 $num_L = 0$ gilt und in Zeile 24 $num_R! = 0$ gilt (sonst wären wir in Fall 3) folgt damit $num = num_L < num_R$.

Im linken Puffer sind num_L falsche Elemente indiziert, im rechten Puffer sind num_R also mit $num_R > num_L$ mindestens so viele falsche Elemente indiziert wie im linken Puffer. Daher finden wir in Zeile 19 für jedes falsche Element im linken Bereich ein falsches Element im rechten Bereich mit dem getauscht werden kann. Nach num_L Vertauschungen sind im linken Bereich alle Elemente richtig, die Invariante ist erfüllt.

Fall 2: $\ell \leftarrow \ell$, $r \leftarrow r - B$ und Fall 3: $\ell \leftarrow \ell + B$, $r \leftarrow r - B$ sind analog zum 1. Fall, mit der Besonderheit, dass im dritten Fall in Zeile 17 gilt $num_L = num_R = num$ und wir damit in Zeile 20 sowohl $num_L = 0$ als auch $num_R = 0$ erhalten. Die restliche Argumentationskette ist analog zu Fall 1.

Terminierung:
Die Schleife terminiert nach maximal $\lceil \frac{N}{B} \rceil - 2$ Durchläufen.

Beweis: Nach 5.7 erhöht sich der betrachtete Bereich in jeder Iteration um mindestens B, die Differenz $r - \ell$ wird also in jeder Iteration um mindestens B verringert, am Anfang gilt $r - \ell + 2 = N$ (r wird direkt am Anfang dekrementiert). Da die Schleife terminiert falls $r - \ell + 2 \leq 2B$ gilt, und die Eingabe aus B Stücken der Größe $\frac{N}{B}$ besteht, muss sie nach maximal $\lceil \frac{N}{B} \rceil - 2$ Durchläufen terminieren. □

Satz 5.9. *Sei B die Blockgröße und N die Länge der Eingabe mit $N = r - \ell + 1$. Der Algorithmus betrachtet bis auf maximal $2B$ Elemente alle Elemente der Eingabesequenz, es gilt nach der Hauptschleife in Zeile 6 $r - \ell \leq 2B$ und für die betrachteten Elemente gilt: $\forall x \in [begin, \ell) : x \leq Pivot$ und $\forall y \in (r, end - 1] : y \geq Pivot$.*

Beweis: Folgt direkt aus der Korrektheit der Schleifeninvariante 5.8. □

Laufzeit

Theorem 5.10. *(Laufzeit der Partitionierung)*
Sei N die Länge der Eingabe mit $N = r - \ell + 1$. Die Laufzeit der Partitionierung liegt dann in $O(N)$.

Wie im Beweis der Schleifeninvariante in 5.8 im Abschnitt der Terminierung gezeigt wurde, besteht die Hauptschleife in Zeile 6 aus maximal $\lceil \frac{N}{B} \rceil - 2$ Iterationen.

Abbildung 5.6.: Instruktionen in *BlockPartition*

Zeile	*Kosten der Instruktionen*
$2-5$	$1+1+2B+4 = O(B)$
6	$\lceil \frac{N}{B} \rceil - 2$ *Schleifendurchläufe* $= O(\frac{N}{B})$
	$\cdot$ { $7-8$: $1+1 = O(1)$; $9-11$: $B \cdot 1 + 2 = O(B)$; $12-13$: $1+1 = O(1)$; $14-16$: $B \cdot 1 + 2 = O(B)$; 17: $1 = O(1)$; $18-19$: $B \cdot 3 = O(B)$; $20-25$: $2+2+1+1+1+1 = O(1)$ }
$26-27$	$O(B)$

Insgesamt: $O(B) + O(\frac{N}{B}) \cdot O(B) + O(B) = O(N)$ □

Sprungfehler

Die Anzahl der Sprungfehler wird mithilfe der Ideen von Edelkamp und Weiß [10] bewiesen.

Theorem 5.11. *Sei $C(N)$ die durchschnittliche Anzahl an Vergleichen in* Quicksort *mit einer Pivot-Menge konstanter Größe. Dann ruft* BlockQuicksort *mit Blockgröße B (ohne eine Grenze der Rekursionstiefe und mit der selben Pivot-Wahl Strategie) höchstens $\frac{6}{B} \cdot C(N) + O(N)$ Sprungfehlvorhersagen hervor. Im Besonderen ruft* Block-Quicksort *mit Median-of-three weniger als $\frac{8}{B} N \log N + O(N)$ Sprungfehlvorhersagen hervor.*

Beweis: In Zeilen 9, 14 und 18 von 5.5 sind For-Schleifen, diese verursachen jeweils einen Sprungfehler. Dazu kommen die If-Bedingungen [4] in Zeilen 7, 12, 17, 22 und 24. Dabei verursachen die If-Bedingungen in Zeilen 7 und 12 sowie 22 und 24 jeweils paarweise nur einen Sprungfehler. Dies liegt daran, dass mindestens einer der beiden Puffer leer sein muss (siehe 5.6). Somit haben wir insgesamt 6 Sprungfehler pro Durchlauf der Hauptschleife in Zeile 6. Wir wissen außerdem, dass jedes Element in der Eingabesequenz genau einmal mit dem Pivot-Element verglichen wird. Pro Durchlauf der Schleife wird mindestens ein neuer Puffer mithilfe von B Element-Pivot Vergleichen gefüllt. Die Anzahl der Hauptschleifen Iterationen ist nun: $\frac{Vergleiche}{B} = \frac{N}{B}$. Somit erhalten wir pro Ausführung der Partitionierung $6\frac{N}{B} = \frac{6}{B} \cdot N$ Sprungfehler. Dazu kommen die konstanten Sprungfehler in jedem Aufruf der Partitionierung (c) und die Sprungfehler in der Quicksort Hauptschleife (c'). Die Grenze für die Aufrufe der Partitionierung liegt bei N, da jedes Element nur ein einziges mal Pivot-Element sein kann und somit sind diese Sprungfehler durch $(c+c') \cdot N$ begrenzt. Abschließend müssen

[4] die Minimum-Bestimmung ist nichts anderes als eine If-Bedingung

noch die Sprungfehler für Insertionsort betrachtet werden, diese liegen in $O(N)$. Da die durchschnittliche Anzahl an Vergleichen in Quicksort für eine Partitionierung mit N Vergleichen mit $C(N)$ angegeben werden kann ($C(N) = 1.386N \log N$ für ein zufälliges Pivot-Element), erhalten wir insgesamt für die Sprungfehler in BlockQuicksort: $\frac{6}{B} \cdot C(N) + (c + c')N + O(N) = \frac{6}{B}C(N) + O(N)$ □

Wir sehen: Die Anzahl der Sprungfehler kann mit steigender Blockgröße reduziert werden. Dies ist ein Tuning-Parameter, da er, falls zu groß gewählt, die guten Cache Aspekte beseitigt. Nach Edelkamp und Weiß [10] gilt: für realistische Eingaben und eine vernünftige Blockgröße dominiert in den Sprungfehlern der lineare Term.

Satz 5.12. *Sei N die Länge der Eingabe mit $N = r - \ell + 1$. Für einen statischen Sprungvorhersager wie in Unterabschnitt 2.2.1 kann der Term $O(N)$ aus 5.11 mit $3,125N$ begrenzt werden.*

Beweis: Wir zeigen zuerst folgendes Lemma:

Lemma 5.13. *BlockQuicksort hat $\frac{11}{8}N$ Sprungfehler, die durch das Insertionsort-Verfahren herbeigeführt sind.*

Beweis: Zeile 7 in 5.2 ergibt einen Sprungfehler für die Abfrage ob BlockQuicksort weiter partitioniert wird und Zeile 21 einen weiteren bei der Frage ob zu Heapsort gewechselt wird. Anschließend ruft in Zeile 24 Insertionsort selbst $r-\ell+1$ Sprungfehler hervor.

Vergleiche dafür 3.14: Die Schleife in Zeile 6 muss $r - \ell$ mal verlassen werden und führt dabei jeweils zu einem Sprungfehler, abschließend muss noch die Schleife in Zeile 3 verlassen werden.

Da unsere Grenze *sizeTHRESHOLD*, unter der wir zu Insertionsort wechseln, den Wert 17 hat, nehmen wir an, dass die durchschnittliche Größe der Sequenzen 8 beträgt. Damit gilt $r - \ell + 1 = 9$. Für die Kosten insgesamt muss nun noch $\frac{N}{8}$ multipliziert werden. Wir erhalten für die durch Insertionsort herbeigeführten Sprungfehler: $(1 + 1 + 9) \cdot \frac{N}{8} = \frac{11}{8}N$. □

Der Term $O(N)$ setzt sich aus den gerade betrachteten Sprungfehlern, hervorgerufen durch Insertionsort, sowie den konstanten Sprungfehlern pro Aufruf der Partitionierung zusammen. Dies sind hauptsächlich Fehler durch Scannen und Umsortieren der verbleibenden Elemente in 5.2 Zeile 24, welche implementierungsabhängig sind und wir hier nicht näher betrachten wollen. Die Klassifikation von Edelkamp und Weiß [10] ergab einen Wert von $\frac{14}{8}N$. Somit ergibt sich für den Term $O(N) = \frac{11}{8}N + \frac{14}{8}N = 3,125N$

5.1.4. Tuning von BlockQuicksort

Wie wir im Kapitel zu Dual-Pivot Quicksort (siehe Unterabschnitt 4.2.4) und Multi-Pivot Quicksort (siehe Unterabschnitt 4.2.5) bereits gesehen haben, kann es durchaus

Sinn ergeben, die *swap*-Methode zu optimieren. Da in Zeilen 16 und 17 von 5.5 mehrere Vertauschungen hintereinander durchgeführt werden, ist es auch in BlockQuicksort eine Verbesserung, die *swap*-Instruktion zu ersetzen. Zur Erinnerung: Um A[0] und A[N] zu tauschen werden folgende drei Instruktionen hintereinander ausgeführt:

1: $temp \leftarrow A[0]$
2: $A[0] \leftarrow A[N]$
3: $A[N] \leftarrow temp$

Sei A die Anzahl zu tauschender Elemente, dann ergibt sich die Anzahl der Instruktionen durch: $\frac{A}{2} \cdot 3$. Im Folgenden wird eine Optimierung gezeigt, die die benötigten Instruktionen auf $A + 1$ reduziert. Diese Optimierung von Edelkamp und Weiß [10] rotiert die Elemente anstatt sie zu vertauschen. Ersetzt wird dafür folgender Code aus Zeile 16 und 17 in *BlockPartition*:

1: **for** j = 0,...,num - 1 **do**
2: swap(A[l + offsets$_L$[start$_L$ + j]],A[r - offsets$_R$[start$_R$ + j]])

mit folgendem neuen Code:

1: temp $\leftarrow A[\ell + offsets_L[start_L]]$
2: $A[\ell + offsets_L[start_L]] \leftarrow A[r - offsets_R[start_R]]$
3: **for** $j = 1, ..., num - 1$ **do**
4: $A[r - offsets_R[start_R + j - 1]] \leftarrow A[\ell + offsets_L[start_L + j]]$
5: $A[\ell + offsets_L[start_L + j]] \leftarrow A[r - offsets_R[start_R + j]]$
6: $A[r - offsets_R[start_R + num - 1]] \leftarrow temp$

Der Prozess ist in Abbildung 5.7 dargestellt.

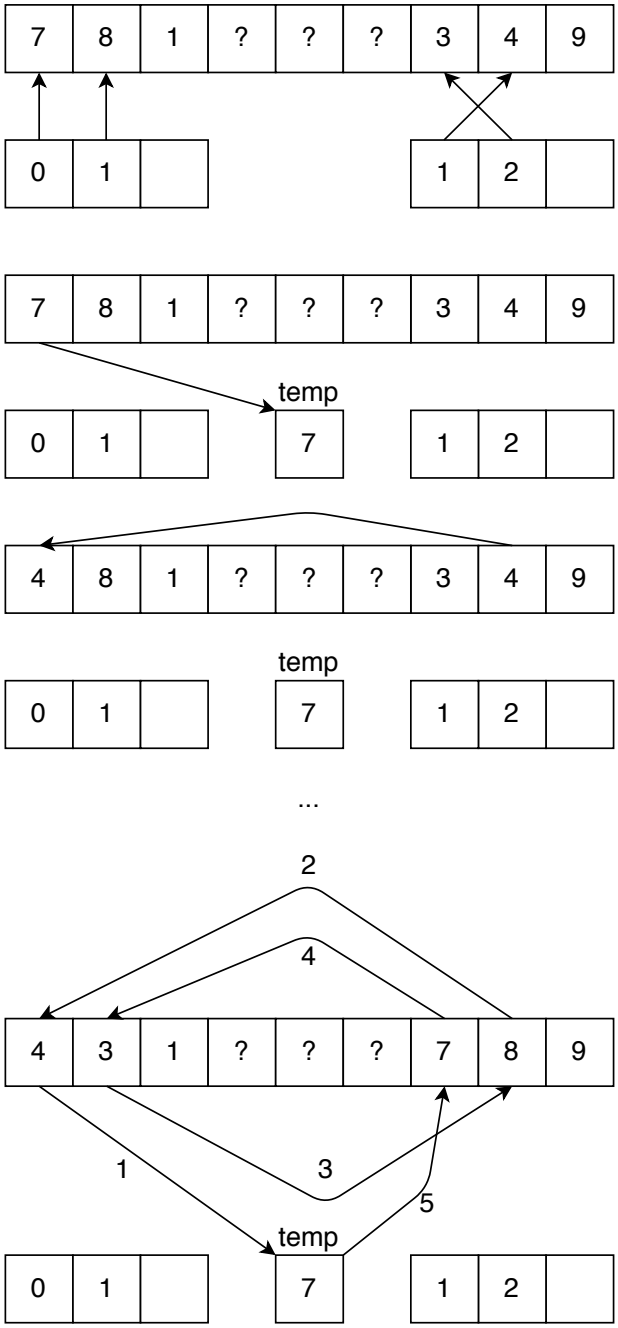


Abbildung 5.7.: Vertauschungen in BlockQuicksort mithilfe von zyklischem Rotieren

Das erste Element wird in einen Hilfsspeicher geschrieben, danach werden die Elemente aus Puffern der rechten und linken Seite abwechselnd überschrieben. Am Ende wird das letzte Element durch den Wert im Hilfsspeicher überschrieben.

5.2. BlockLomuto als simple Alternative

Mit BlockQuicksort in Abschnitt 5.1 und IPS^{4}o in Abschnitt 6.3 werden in dieser Arbeit Verfahren vorgestellt welche zeigen, dass Optimierungsversuche von Sortieralgorithmen schnell hochgradig nicht-trivial werden können. Dies hat mehrere Nachteile, angefangen vom erschwerten Testen, über einer komplizierteren Suche nach weiteren Optimierungsmöglichkeiten, bis hin zu dem eventuellen Verlust von Effizienz in der Implementierung. Dies wird durch die Aussage von Axtmann et. al. [5] unterstrichen, in welcher Code-Komplexität als ein Hauptnachteil beim Vermeiden von Sprungfehlern angeführt wird. Die in diesem Abschnitt vorgestellten Verfahren BlockLomuto und Dual-Pivot BlockLomuto stellen daher Varianten von BlockQuicksort dar, welche für (theoretische) Einschnitte in der Laufzeit wesentlich an Komplexität verlieren. In Laufzeitversuchen werden wir zusätzlich feststellen, dass diese Variaten in der Praxis durchaus mit BlockQuicksort mithalten können.

5.2.1. BlockLomuto

Die Strategie hinter dem Verfahren ist die gleiche, wie auch in Lomutosort (siehe Unterabschnitt 4.2.1). Wie der Name bereits verrät, wird auch hier die Taktik von Lomuto verwendet, die Zeiger in die gleiche Richtung laufen zu lassen. Zusätzlich will man an der ursprünglichen Idee aus BlockQuicksort (siehe Unterabschnitt 5.1.2) festhalten, die Vergleiche von den Vertauschungen zu trennen. Die Ergebnisse von Vergleichen werden also wieder in einem Puffer gespeichert. Die Struktur ist in Abbildung 5.8 dargestellt. Die beiden Bereiche mit Elementen kleiner dem Pivot-Element und größer dem Pivot-Element wachsen nach rechts, mit dem Resultat, dass die Zeiger nicht aufeinander zulaufen und es daher nur einen Puffer gibt.

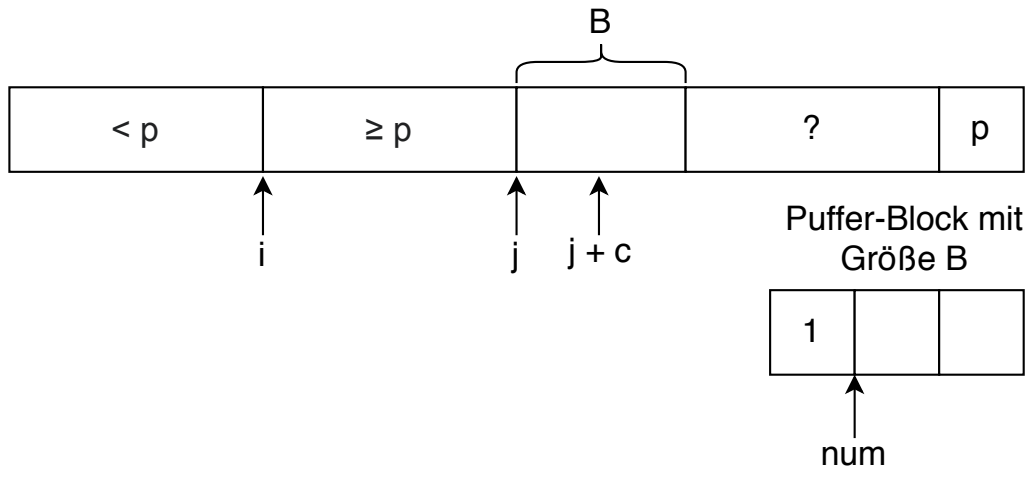


Abbildung 5.8.: Die Zeiger, Bereiche und Puffer im BlockLomuto-Verfahren nach Aumüller und Hass [4]

Das Verfahren BlockLomuto in 5.14 wurde von Aumüller und Hass in ihrer Abhandlung „Einfaches und schnelles BlockQuicksort mit dem Partitionierungsschema von Lomuto“ [4] vorgestellt, es arbeitet dabei mit einem Zwei-Schritte Prinzip: In Zeilen 6 bis 8 werden die Elemente, welche kleiner als das Pivot-Element sind, im Puffer gespeichert. Anschließend werden diese in Zeilen 9 bis 11 in den richtigen Bereich vertauscht.

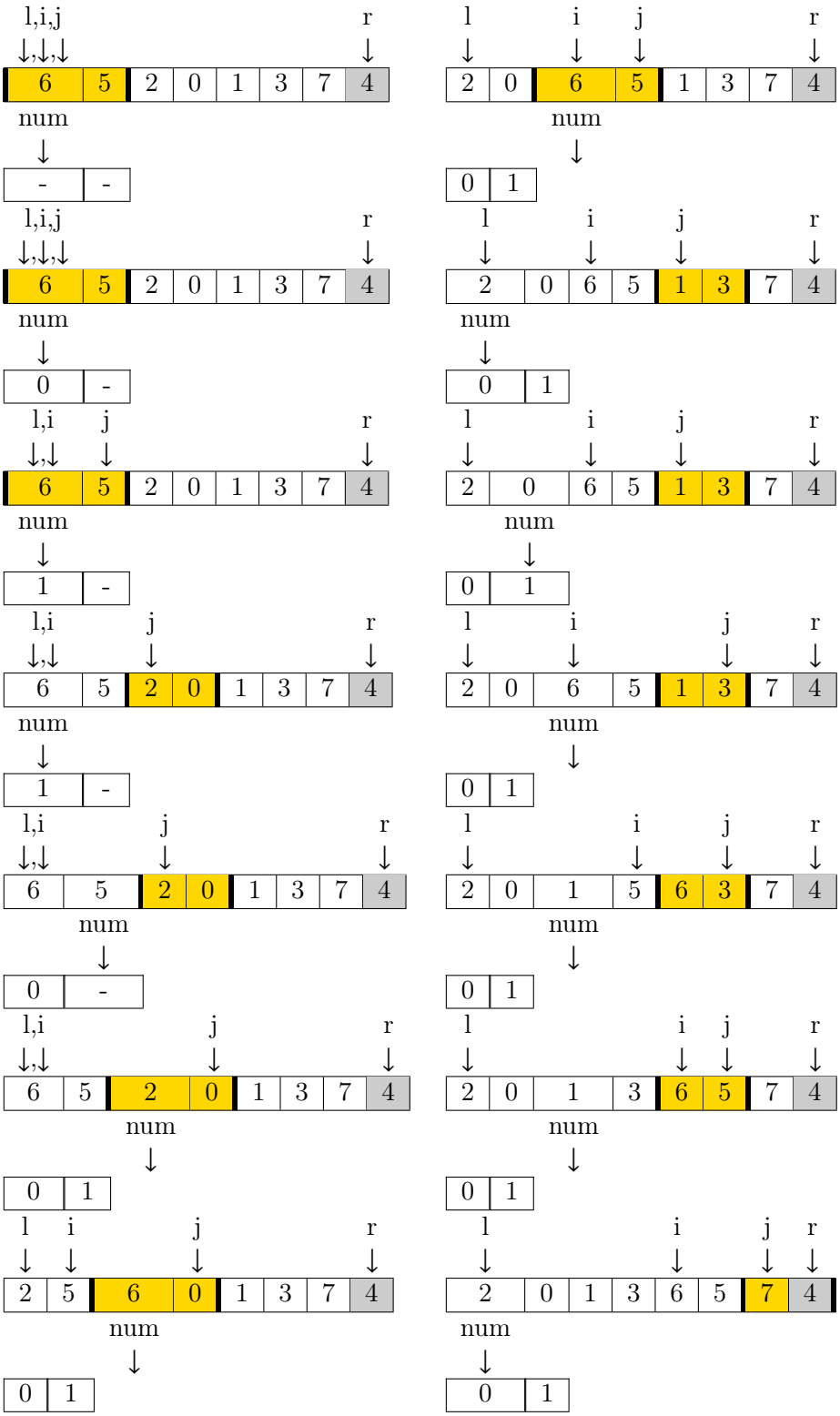
Algorithmus 5.14 Block Partitioning

```
 1: function BLOCKLOMUTO1(A[ℓ,...,r])
Require: r − ℓ > 0, Pivot in A[r]
 2:     p ← A[r];
 3:     integer block[0,..,B − 1], i, j ← ℓ, num ← 0
 4:     while j < r do
 5:         t ← min(B, r − j);
 6:         for c ← 0; c < t; c ← c + 1 do
 7:             block[num] ← c;
 8:             num ← num + (p > A[j + c]);
 9:         for c ← 0; c < num; c ← c + 1 do
10:             Swap A[i] and A[j + block[c]]
11:             i ← i + 1
12:         num ← 0;
13:         j ← j + t;
14:     Swap A[i] and A[r];
15:     return i;
```

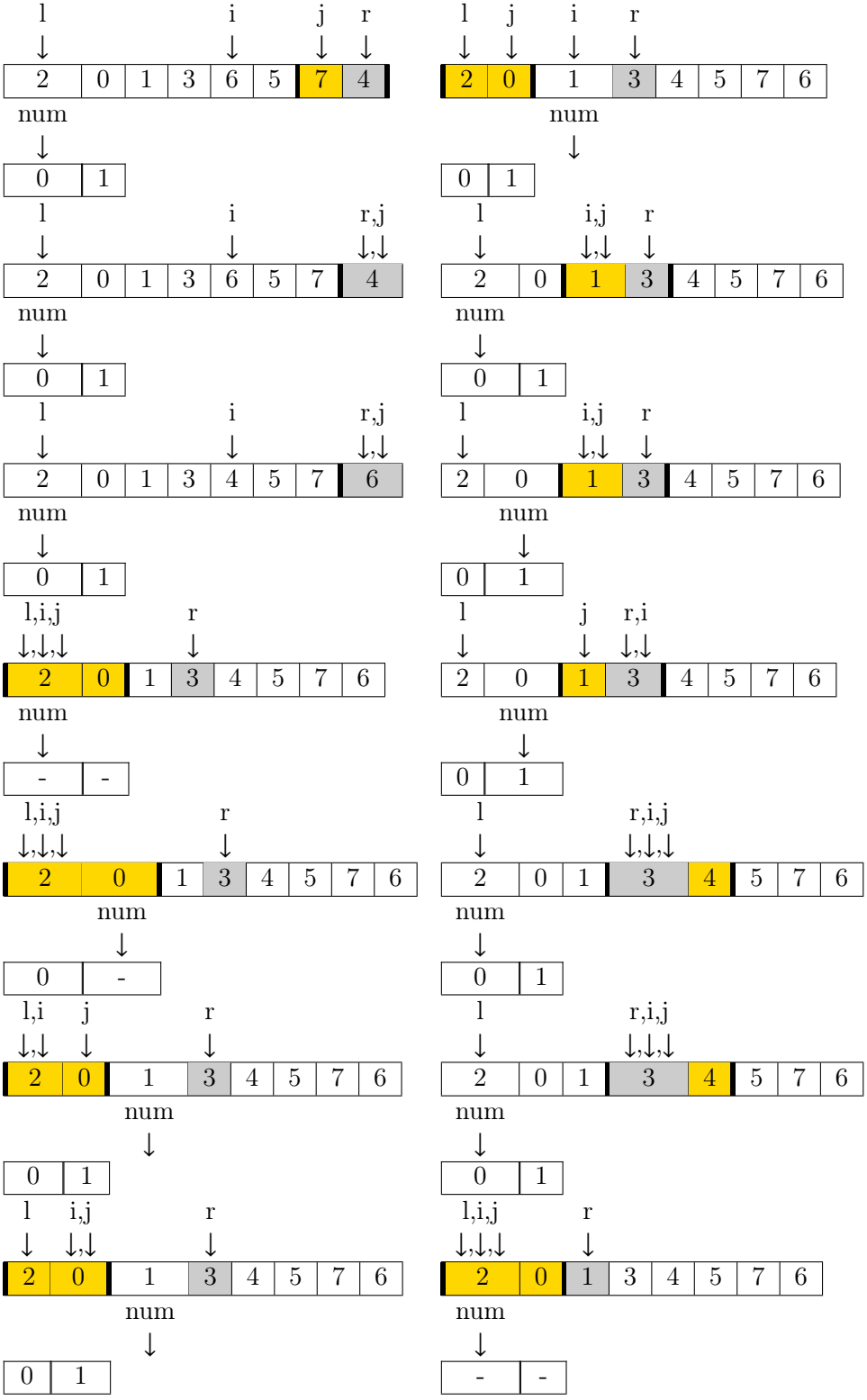
Abbildung 5.9 zeigt eine Beispielsequenz in BlockLomuto. Das aktuelle Pivot-Element ist grau hervorgehoben. Die klassifizierten Bereiche werden durch die Zeiger i und j begrenzt. Des Weiteren geben die Zeiger l und r die aktuellen Bereiche an, der aktuelle Pufferbereich ist gelb hervorgehoben und der Zeiger *num* gibt an, wie viele Elemente im Pufferbereich gefunden wurden, welche einen kleineren Wert als das Pivot-Element haben.

5.2.2. Dual-Pivot BlockLomuto

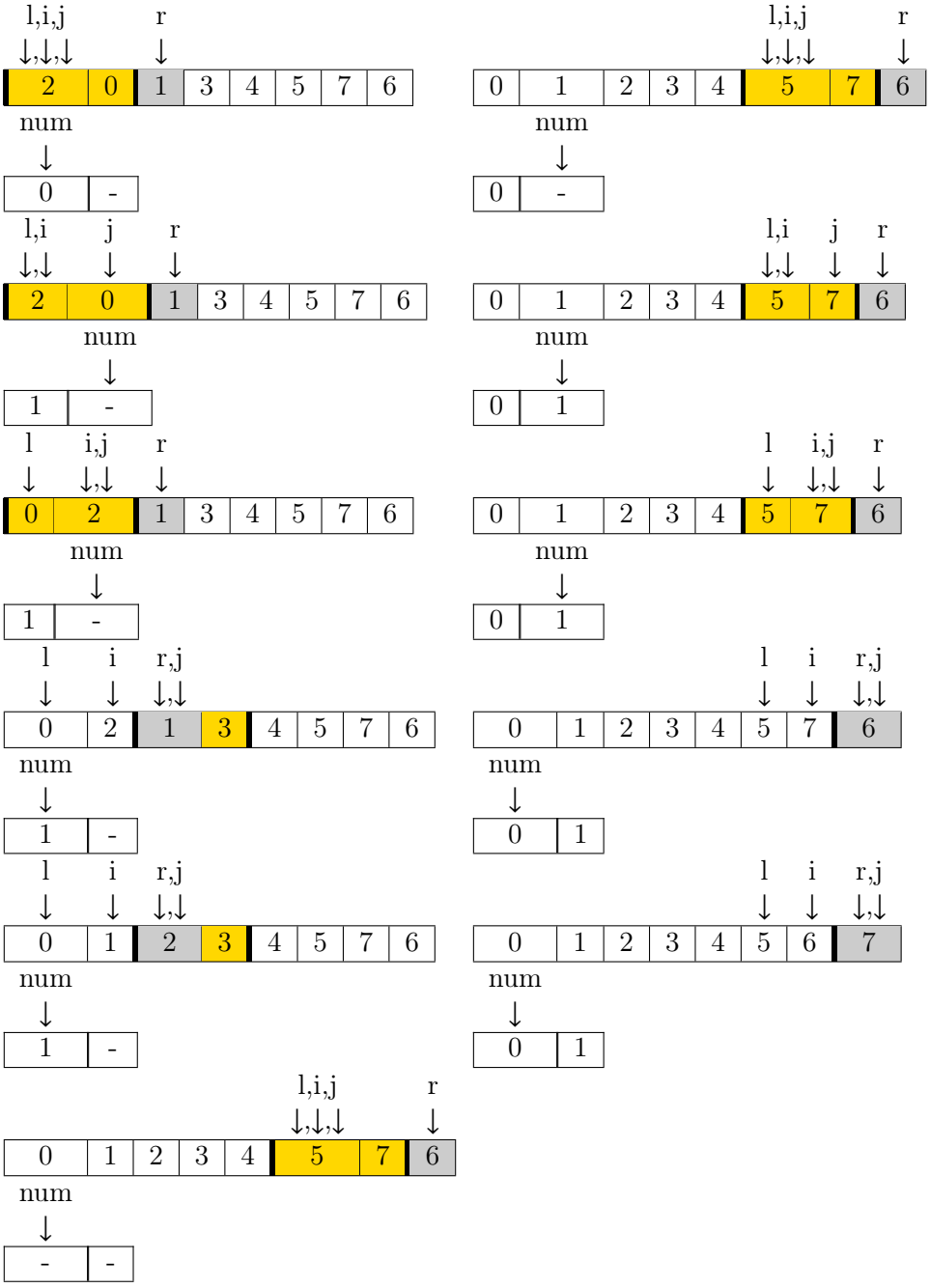
Mit dem Verfahren Dual-Pivot BlockLomuto haben Aumüller und Hass [4] eine Möglichkeit gefunden, den Ansatz von BlockQuicksort auf zwei Pivot-Elemente zu erweitern. Dabei sorgt das Hinzufügen eines weiteren Pivot-Elementes für einen robusteren Algorithmus in Bezug auf verschiedene Eingabetypen sowie Duplikate. Die Struktur wird dabei in Abbildung 5.10 dargestellt.


l,i,j
r
6 5 2 0 1 3 7 4
num
- -
l,i,j
r
6 5 2 0 1 3 7 4
num
0 -
l,i
j
r
6 5 2 0 1 3 7 4
num
1 -
l,i
j
r
6 5 2 0 1 3 7 4
num
1 -
l,i
j
r
6 5 2 0 1 3 7 4
num
0 -
l,i
j
r
6 5 2 0 1 3 7 4
num
0 1
l
i
j
r
2 5 6 0 1 3 7 4
num
0 1
l
i
j
r
2 0 6 5 1 3 7 4
num
0 1
l
i
j
r
2 0 6 5 1 3 7 4
num
0 1
l
i
j
r
2 0 6 5 1 3 7 4
num
0 1
l
i
j
r
2 0 1 5 6 3 7 4
num
0 1
l
i
j
r
2 0 1 3 6 5 7 4
num
0 1
l
i
j
r
2 0 1 3 6 5 7 4
num
0 1

(a) Seite 1


l i j r
2 0 1 3 6 5 7 4
num
0 1
l i r,j
2 0 1 3 6 5 7 4
num
0 1
l i r,j
2 0 1 3 4 5 7 6
num
0 1
l,i,j r
2 0 1 3 4 5 7 6
num
- -
l,i,j r
2 0 1 3 4 5 7 6
num
0 -
l,i j r
2 0 1 3 4 5 7 6
num
0 1
l i,j r
2 0 1 3 4 5 7 6
num
0 1
l j i r
2 0 1 3 4 5 7 6
num
0 1
l i,j r
2 0 1 3 4 5 7 6
num
0 1
l j r,i
2 0 1 3 4 5 7 6
num
0 1
l r,i,j
2 0 1 3 4 5 7 6
num
0 1
l r,i,j
2 0 1 3 4 5 7 6
num
0 1
l,i,j r
2 0 1 3 4 5 7 6
num
- -

(b) Seite 2



(c) Seite 3

Abbildung 5.9.: Eine Beispielsequenz in BlockLomuto

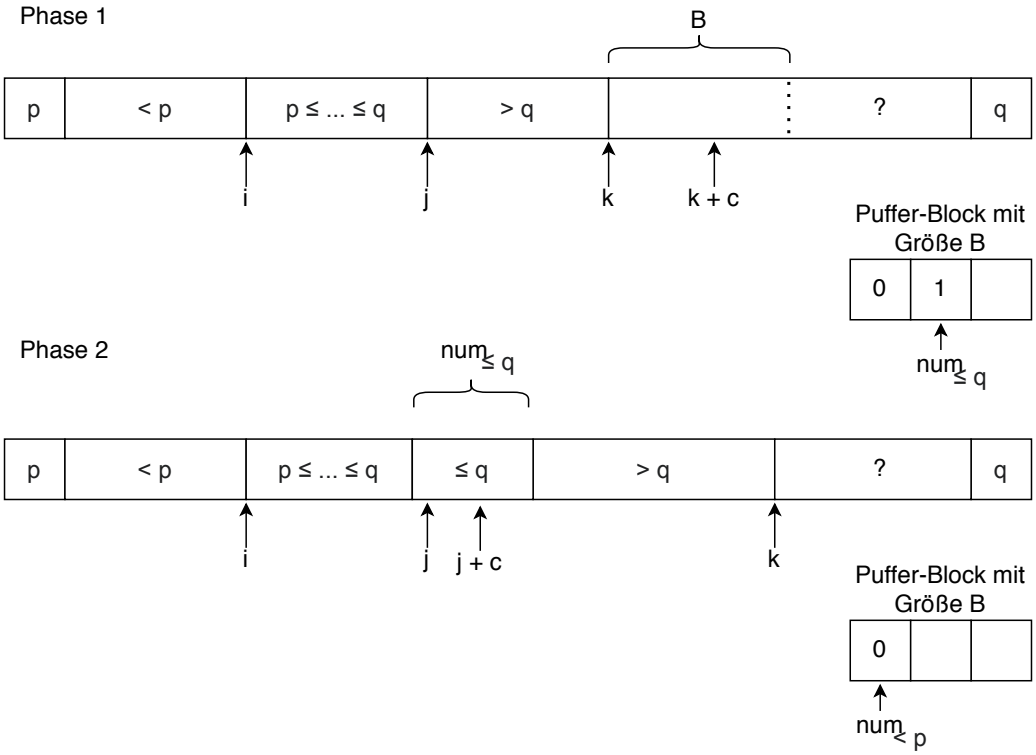


Abbildung 5.10.: Die Zeiger, Bereiche und Puffer in den zwei Phasen des Dual-Pivot Block-Lomuto-Verfahrens nach Aumüller und Hass [4]. Als Erstes wird der Puffer Block mit Elementen gefüllt, welche kleiner als das rechte Pivot-Element sind. Anschließend werden diese vertauscht. In der zweiten Phase werden die getauschten Elemente, welche kleiner als das linke Pivot-Element sind in den Puffer geschrieben und danach vertauscht.

Zusammen mit der Struktur kann das Vorgehen von Dual-Pivot BlockLomuto, welches in 5.15 dargestellt ist, beschrieben werden. Im Unterschied zum vorherigen Ansatz haben wir ein zusätzliches Pivot-Element q, einen zusätzlichen Bereich $p \leq \circ \leq q$, dessen Grenze durch einen weiteren Zeiger k vorgehalten wird. Außerdem müssen wir den vorherigen Puffer-Zeiger *num* in $num_{\leq q}$ und $num_{<p}$ aufteilen. Das Vorgehen bis Zeile 13 ist dabei das Gleiche wie vorher, die Elemente werden mit dem Pivot-Element p verglichen, die Ergebnisse mithilfe von $num_{\leq q}$ in den Puffer geschrieben und anschließend vertauscht. Den größten Unterschied zu vorherigen Verfahren findet man in Zeile 13 bis 18. Da der Algorithmus mit zwei Pivot-Elementen arbeitet, muss noch die Relation zum zweiten Pivot-Element q festgestellt werden. Dafür wird das Zwei-Schritt-Verfahren auf der gefundenen Teilmenge von Elementen kleiner dem Pivot-Element p, für das Pivot-Element q wiederholt. Der Puffer kann dabei wiederverwendet werden, da der erste Tauschvorgang bereits abgeschlossen war.

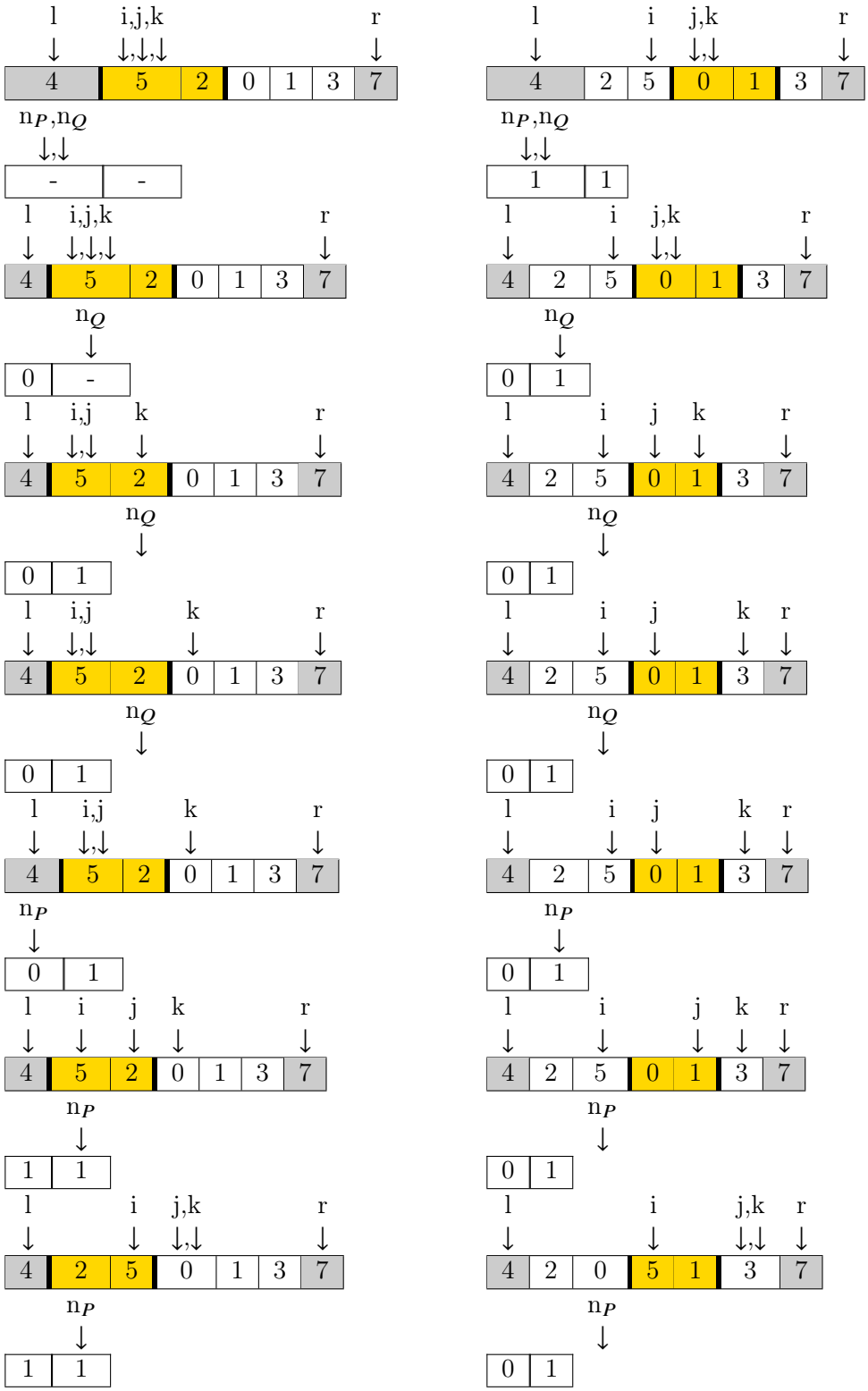
Algorithmus 5.15 Dual-Pivot Block Partitioning

```
 1: function Dual-Pivot BlockLomuto(A[ℓ,...,r])
Require: r − ℓ > 0, Pivots in A[ℓ] ≤ A[r]
 2:     p ← A[ℓ]; q ← A[r]
 3:     integer block[0,...,B − 1]
 4:     i, j, k ← ℓ + 1, num<p,num≤q ← 0
 5:     while k < r do
 6:         t ← min(B, r − k);
 7:         for c ← 0; c < t; c ← c + 1 do
 8:             block[num≤q] ← c;
 9:             num≤q ← num≤q + (q ≥ A[k + c]);
10:         for c ← 0; c < num≤q; c ← c + 1 do
11:             Swap A[j + c] and A[k + block[c]]
12:         k ← k + t
13:         for c ← 0; c < num≤q; c ← c + 1 do
14:             block[num<p] ← c;
15:             num<p ← num<p + (p > A[j + c]);
16:         for c ← 0; c < num<p; c ← c + 1 do
17:             Swap A[i] and A[j + block[c]]
18:             i ← i + 1
19:         j ← j + num≤q;
20:         num<p, num≤q ← 0;
21:     Swap A[i − 1] and A[1];
22:     Swap A[j] and A[n];
23:     return (i − 1, j);
```

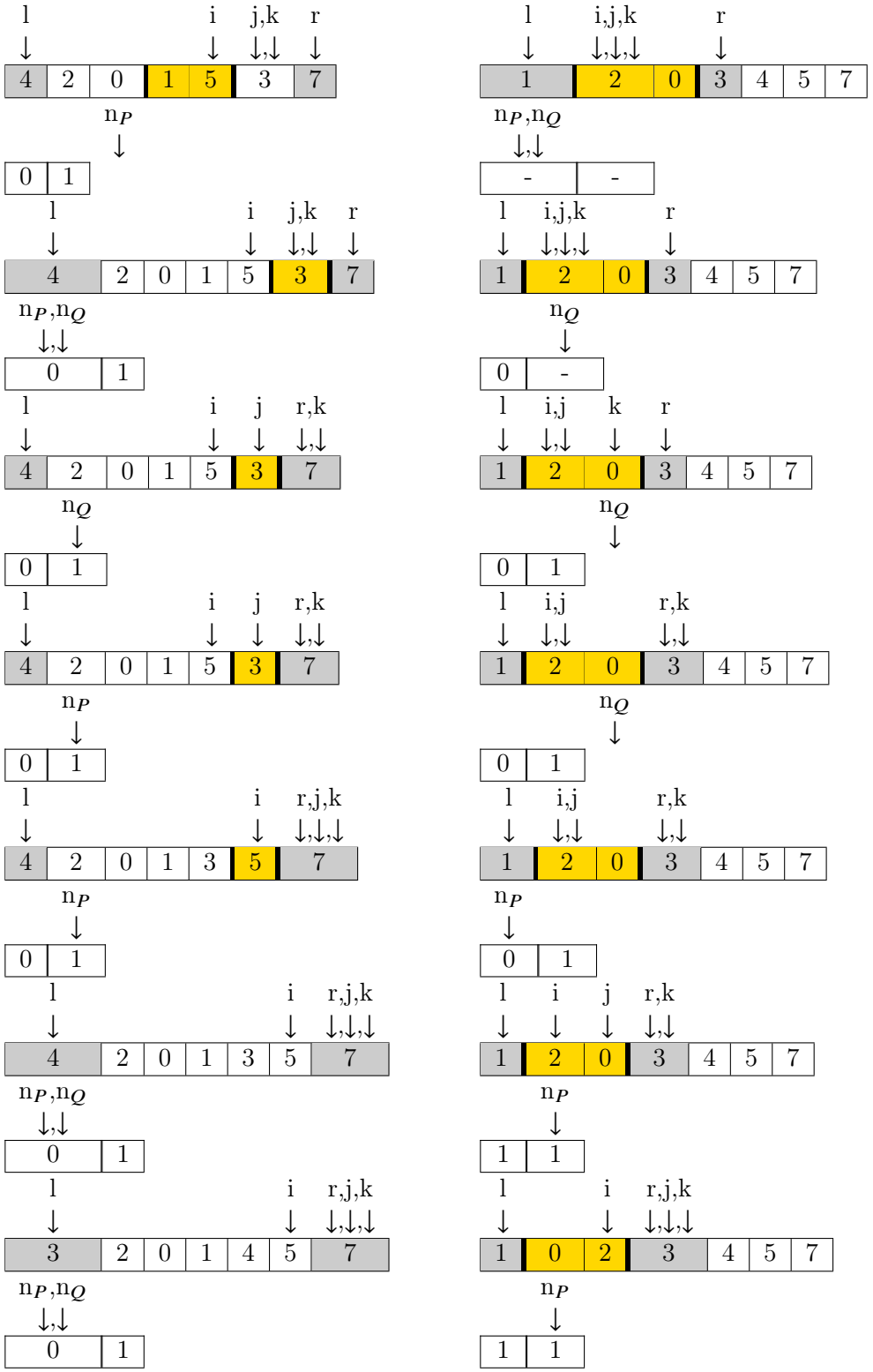
Abbildung 5.11 zeigt eine Beispielsequenz in Dual-Pivot BlockLomuto. Die aktuellen Pivot-Elemente sind grau hervorgehoben. Die klassifizierten Bereiche werden durch die Zeiger i, j und k begrenzt. Des Weiteren geben die Zeiger l und r die aktuellen Bereiche an und der aktuelle Pufferbereich ist gelb hervorgehoben. Ferner gibt der Zeiger n_Q an, wie viele Elemente im Pufferbereich gefunden wurden, welche einen kleineren Wert als das rechte (größere) Pivot-Element haben, während der Zeiger n_P angibt, wie viele Elemente von den durch n_Q angegebenen Elementen einen kleineren Wert als das linke (kleinere) Pivot-Element haben.

5.2.3. Weitere Anmerkungen

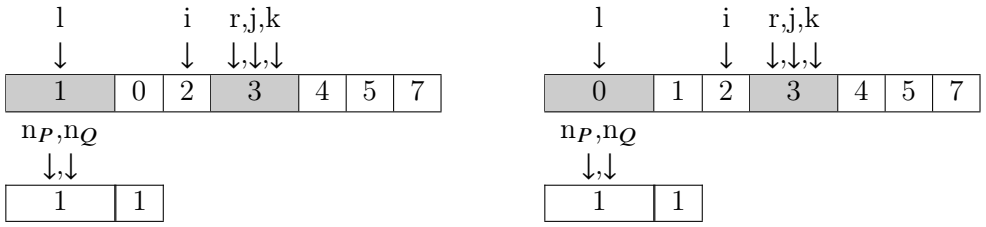
Wie wir auch schon in BlockQuicksort gesehen haben, ist die Wahl von Blockgröße und Pivot-Wahl entscheidend für die spätere Laufzeit in der Implementierung. Die erhaltene Laufzeitverbesserung für ansteigende Blockgrößen wird dabei immer kleiner und die Suche nach dem perfekten Wert ist wieder eine Tuning-Arbeit.


l i,j,k r
4 5 2 0 1 3 7
nP,nQ
- -
l i,j,k r
4 5 2 0 1 3 7
nQ
0 -
l i,j k r
4 5 2 0 1 3 7
nQ
0 1
l i,j k r
4 5 2 0 1 3 7
nQ
0 1
l i,j k r
4 5 2 0 1 3 7
nP
0 1
l i j k r
4 5 2 0 1 3 7
nP
1 1
l i j,k r
4 2 5 0 1 3 7
nP
1 1
l i j,k r
4 2 5 0 1 3 7
nP,nQ
1 1
l i j,k r
4 2 5 0 1 3 7
nQ
0 1
l i j k r
4 2 5 0 1 3 7
nQ
0 1
l i j k r
4 2 5 0 1 3 7
nQ
0 1
l i j k r
4 2 5 0 1 3 7
nP
0 1
l i j k r
4 2 5 0 1 3 7
nP
0 1
l i j,k r
4 2 0 5 1 3 7
nP
0 1

(a) Seite 1


l i j,k r
4 2 0 1 5 3 7
n_P
0 1
l i j,k r
4 2 0 1 5 3 7
n_P,n_Q
0 1
l i j r,k
4 2 0 1 5 3 7
n_Q
0 1
l i j r,k
4 2 0 1 5 3 7
n_P
0 1
l i r,j,k
4 2 0 1 3 5 7
n_P
0 1
l i r,j,k
4 2 0 1 3 5 7
n_P,n_Q
0 1
l i r,j,k
3 2 0 1 4 5 7
n_P,n_Q
0 1
l i,j,k r
1 2 0 3 4 5 7
n_P,n_Q
- -
l i,j,k r
1 2 0 3 4 5 7
n_Q
0 -
l i,j k r
1 2 0 3 4 5 7
n_Q
0 1
l i,j r,k
1 2 0 3 4 5 7
n_Q
0 1
l i,j r,k
1 2 0 3 4 5 7
n_P
0 1
l i j r,k
1 2 0 3 4 5 7
n_P
1 1
l i r,j,k
1 0 2 3 4 5 7
n_P
1 1

(b) Seite 2



(c) Seite 3

Abbildung 5.11.: Eine Beispielsequenz in Dual-Pivot BlockLomuto

Die beste Pivot-Wahl Strategie

Die BlockQuicksort-Implementation von Axtmann Weiß, und Hass [26] ändert die Pivot-Wahl Strategie je nach Größe der Eingabesequenz. Nach Aussage der Autoren ist dies jedoch kein großer Unterschied zur Pivot-Wahl Strategie, welche den vorgestellt wird.

Wähle fünf Samples der Größe fünf und daraus das jeweilige Median Element. Die beiden Pivot-Elemente werden anschließend auf das erste und dritte Element der fünf Median Elemente gesetzt.

Weitere Erkenntnisse zu denen Aumüller und Hass in ihrer Arbeit [4] gekommen sind werden im Folgenden angeführt.

- In Dual-Pivot BlockLomuto führten skewed Pivots (siehe Abschnitt 4.1.3) zu besseren Ergebnissen.
- Die Optimierung: Loop-Unrolling konnte die Laufzeit nicht verbessern. Dies liegt vermutlich an dem vereinfachten Code, welcher nun vom Compiler selbst entrollt werden kann.
- Zyklische Rotation wie in BlockQuicksort haben die Laufzeit sogar erhöht.
- Eine 3-Pivot Element Variante erreichte keine Verbesserungen, genauso wie eine Dual-Pivot Variante des originären BlockQuicksort-Verfahrens.

5.3. Nebenläufige BlockQuicksort Variante

Die Verwendung von mehreren Threads gestattet es, einzelne abgekapselte Funktionalitäten auf geteiltem Speicher durchzuführen. Die einzelnen Funktionalitäten sind dabei nichts weiter als eine *Subroutine*[5] bzw. eine Folge davon. Quicksort bedient sich dem Konzept von Divide-and-conquer und ist daher gut zu parallelisieren. Die entstehenden Teilprobleme können den Threads zugewiesen werden, vorausgesetzt, das Teilproblem ist hinreichend groß und der Prozessor hält noch einen freien Thread vor. Im Folgenden (5.16) ist der High-Level Pseudocode einer zu 5.2 ähnlichen Optimierung von BlockQuicksort dargestellt. Diese wurde wurde von Christof Kaser

[5]Eine mögliche Übersetzung lautet: „Unterprogram“

in einem Blog-Post [18] veröffentlicht. Im Fokus soll hierbei die Idee der Threads stehen, daher wurde unter anderem auf die Heapsort-Optimierung verzichtet. Eine Auffälligkeit erkennt man in der deutlich höheren Grenze für Insertionsort. Diese lässt sich auf die explizite Optimierung auf große Eingaben mit 50 Millionen Elementen und der Bearbeitung auf einem 4-Thread-System zurückführen.

5.3.1. Algorithmus

Algorithmus 5.16 Thread BlockQuicksort

```
1:  function SORT_THREAD(Stack)
2:      hole left und right vom Stack
3:      while true do
4:          if (left − right) < 50 then
5:              Insertionsort(left, right)
6:              if Stack ist leer then
7:                  gebe Thread frei und beende Methode
8:              else
9:                  hole left und right vom Stack
10:         else
11:             cut ← partition(left, right)
12:             Verwalte die kleinere Hälfte mit left, right und die größere mit ℓ, r
13:             if ((r − ℓ) > 100000) und noch ein Thread frei ist then
14:                 Starte einen Thread der den Bereich ℓ, r behandelt.
15:             else
16:                 lege ℓ, r auf dem Stack ab.
```

Der Algorithmus arbeitet mit einem Stack und holt sich in jeder Iteration die neuen Werte für die Zeiger ℓ und r. Dabei benutzt das Verfahren die uns bereits bekannte Optimierung mit dem Basis-Fall Insertionsort (siehe Zeile 4f). Der Unterschied zu den anderen Verfahren ist die Verwaltung von Threads. Anstatt wie bei BlockQuicksort die Zeiger des größeren Arbeitsbereiches direkt auf dem Stack abzuspeichern (siehe 5.2 Zeile 8ff oder 12ff) wird mit diesen hier, falls der entsprechende Arbeitsbereich hinreichend groß ist und noch mindestens ein freier Thread existiert, das Teilproblem auf einem neuen Thread gestartet und dort direkt weiterverarbeitet (vgl. Zeile 13,14).

5.3.2. Laufzeit

Das Thread BlockQuicksort-Verfahren wurde mit der gcc Implementierung std::sort und BlockQuicksort auf einem Intel Celeron Zwei-Kern Prozessor mit ingesamt 4 Threads verglichen. Dabei wurde für alle Verfahren eine Eingabegröße von 50 Millionen gewählt. Die Ergebnisse, dargestellt in Tabelle 5.1, zeigen eine deutliche Verbesserung gegenüber der sequentiellen Variante.

Verfahren	Sekunden
std::sort	5.98
BlockQuicksort	4.57
Thread BlockQuicksort	2.42

Tabelle 5.1.: Laufzeit-Vergleich verschiedener Verfahren auf dem Intel Celeron. Die Daten stammen von Kaser [18].

Trotz der guten Laufzeit sollten unbedingt zwei Dinge klar gestellt werden. Erstens lohnt sich diese Art der Optimierung nur für sehr große Eingaben. Wie auch in Zeile 13 dargestellt, werden neue Threads nur für Teilsequenzen mit mehr als 100.000 Elementen gestartet. Zweitens und damit wahrscheinlich sogar der wichtigere Punkt, werden für diese Optimierung mehr Ressourcen benötigt. Die große Stärke von Quicksort, ein Sortierverfahren für den Allgemeingebrauch zu sein, könnte damit stark beschnitten werden. Falls jedoch alle Anforderungen an Eingabe und Ressourcen-Verfügbarkeit erfüllt sind oder die Verwendung von Thread BlockQuicksort dynamisch anhand der Verfügbarkeit von Ressourcen und der Struktur der Eingaben entschieden wird, erhalten wir eine potentiell hohe Leistungssteigerung.

6. In-Place Parallel Super Scalar Samplesort

In Kapitel 5 haben wir bereits Sortieralgorithmen gefunden, welche Sprungfehler vermeiden. In diesem Kapitel sollen erneut Algorithmen vorgestellt werden, bei denen die Vermeidung von Sprungfehlern im Mittelpunkt steht. Wiederum soll bei der Entwicklung neuer Ansätze Quicksort als Ausgangspunkt dienen. Die bisherigen Verfahren haben entweder ein Pivot-Element (siehe BlockQuicksort und BlockLomuto) oder zwei Pivot-Elemente (siehe Dual-Pivot BlockLomuto) benutzt. Wie wir jedoch in Unterabschnitt 4.2.4 bereits gesehen haben, kann die Einführung weiterer Pivot-Elemente zu einer Verringerung der Datenzugriffe führen. Erhöht man die Anzahl der erzeugten Teilprobleme durch die Nutzung vieler Pivot-Elemente, erhält man Samplesort. Dieses Verfahren ist im Gegensatz zu den bisherigen Quicksort-Varianten weder In-Place, noch wird die Vermeidung von Sprungfehlern betrachtet. Dafür ist Samplesort jedoch gut für Parallelisierung geeignet und soll uns als Grundlage dienen um erst die Vermeidung von Sprungfehlern einzuführen und danach die In-Place Eigenschaft von Quicksort wieder zu erlangen.

Dafür werden in diesem Kapitel neben Samplesort zwei weitere Verfahren vorgestellt, bei denen die Vermeidung von Sprungfehlern im Vordergrund steht. Die Grundidee ist wieder, die Vergleiche von den Sprüngen zu entkoppeln. Es werden die folgenden drei Verfahren betrachtet:

- Samplesort
- Super Scalar Samplesort(S^3)
- In-Place Parallel Super Scalar Samplesort(IPS^4o)

Alle drei Ansätze bauen jeweils aufeinander auf, dabei werden wir am Ende trotz des zusätzlichen Speicherbedarfs von Samplesort einen Algorithmus erhalten, welcher die In-Place Eigenschaft mit Puffern konstanter Größe erfüllt, ähnlich den Puffern in BlockQuicksortund damit wieder In-Place ist. Zusätzlich wird ein hohes Maß an Instruction-level parallelism, Cache-Effizienz und asymptotisch optimale Laufzeit erreicht.

6.1. Samplesort

In diesem Abschnitt werden wir uns das Verfahren von Quicksort das erste Mal mithilfe eines Binärbaumes vorstellen. Dabei stellen die einzelnen Knoten die Pivot-Elemente der gleichen Rekursionsstufe dar. Wie in Abbildung 6.1 dargestellt, wird die 12 als Erstes Pivot-Element gewählt. Die Eingabesequenz wird nun wie gewohnt in zwei Bereiche aufgeteilt. Für den kleineren Bereich wird 5 als Pivot-Element gewählt und immer so weiter. Nach den zwei angezeigten Rekursionsstufen ist die Eingabesequenz in acht Bereiche aufgeteilt (siehe Abbildung 6.2).

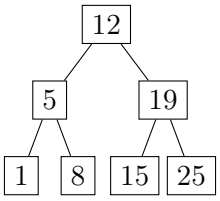


Abbildung 6.1.: Binärbaum der Pivot-Elemente im Quicksort-Verfahren

Die Idee von Samplesort, einem erstmals von Frazer und McKellar im Jahre 1970 vorgestellten Verfahren, ist sehr ähnlich zu der von Quicksort. Anstatt immer nur ein Pivot-Element zu wählen und die Eingabesequenz anhand von diesem aufzuteilen, wählt Samplesort als erstes ein Sample aus der Eingabesequenz und sortiert dieses. Aus dem sortierten Sample ist es dann möglich, einen Binärbaum zu erstellen, welcher aussehen könnte wie jener aus Abbildung 6.1. Es ist nun möglich, die Eingabesequenz mit diesem Binärbaum direkt in 8 Bereiche aufzuteilen und damit die zwei Rekursionsstufen, welche Quicksort benötigen würde, zu überspringen.

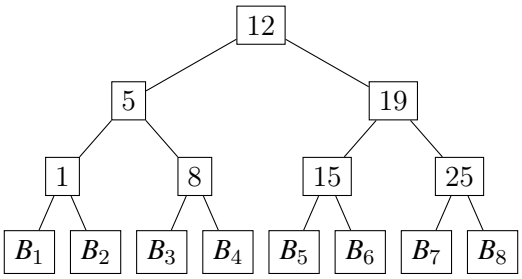


Abbildung 6.2.: Binärbaum der Pivot-Elemente im Quicksort-Verfahren mit den 8 entstehenden Bereichen B_1 bis B_8

6.1.1. Sampling, Klassifizeriung und Aufteilung

Um die zuvor skizzierte Vorgehensweise von Samplesort etwas detaillierter darzustellen, teilen wir das Verfahren in 2 Teile:

1. Sampling
2. Klassifizierung und Aufteilung

Sampling

Definition 6.1. (Sample, Splitters, Buckets)
Die zufällige Wahl einer Menge von Elementen $s_1, ..., s_{k-1}$ aus der Eingabesequenz wird *Sample* genannt. Sortiert man das Sample und fügt die Grenzen $S_0 = -\infty$ und $S_k = \infty$ hinzu, so erhalten wir k verschiedene Bereiche zwischen: S_i und S_{i+1} für $0 \leq i < k$, welche *Buckets* genannt werden. Um Elemente in die Buckets zu klassifizieren, erstellen wir aus dem sortierten Sample einen Binärbaum. Das zum Binärbaum zugehörige Feld wird *Splitters* genannt.

Der erste Schritt des Samplings ist die Wahl des Samples. Da die entstehenden Buckets durch den Binärbaum definiert werden, wählen wir die Bucket Anzahl k als Potenz von 2: $k = 2^i$ für ein $i > 0$, $i \in \mathbb{N}$. Man sieht leicht, dass wir für k Buckets ein Sample der Größe $k - 1$ benötigen. Ein Beispiel Sample wird in Abbildung 6.3 dargestellt.

s_1	s_2	s_3	s_4	s_5	s_6	s_7
19	8	1	12	5	15	25

Abbildung 6.3.: Ein Beispiel Sample der Größe $k - 1 = 7$ für Samplesort

Anschließend muss das Sample sortiert werden, damit wir daraus einen Binärbaum erstellen können (Abbildung 6.4).

s_1	s_2	s_3	s_4	s_5	s_6	s_7
1	5	8	12	15	19	25

Abbildung 6.4.: Das sortierte Beispiel Sample der Größe $k = 7$ für Samplesort

Der Binärbaum wird nun wie folgt aus dem sortierten Sample erstellt: Der Median des sortierten Samples bildet die Wurzel des Binärbaumes, weiter werden die Wurzeln der nächsten Stufe als das obere bzw. untere Quartil des Samples gewählt. Auf diese Weise fahren wir fort, bis keine weiteren dieser Quantile vorhanden sind, an welchem Punkt der Binärbaum fertig ist (siehe Abbildung 6.5). Im Fall, dass die Samplegröße eine Potenz der Zahl 2 ist, ist dies der Fall, sobald alle Elemente des Samples benutzt werden.

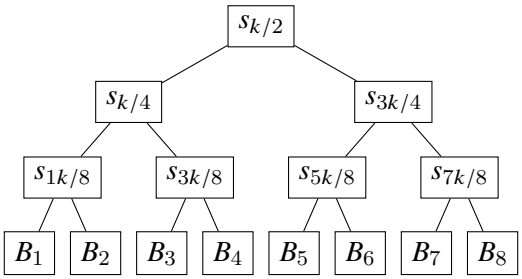


Abbildung 6.5.: Erstellung eines Binärbaumes aus einem sortierten Sample: Der Median bildet die Wurzel, gefolgt von dem oberen und unteren Quartil und immer so weiter. Die Abbildung stammt von Sanders et. al. [33]

Da die Erstellung des Binärbaumes und die Erstellung eines Splitter-Feldes äquivalent sind, werden wir andersherum vorgehen und ein Splitter-Feld erstellen (siehe Abbildung 6.6) um daraus den Binärbaum zu erhalten.

Mithilfe des Binärbaumes können wir nun Elemente in die einzelnen Buckets klassifizieren. Indem wir $S_0 = -\infty$ und $S_k = \infty$ hinzufügen, können wir die Bucket Intervalle wie folgt definieren: Bucket j besteht aus allen Elementen e für die gilt: $S_{j-1} < e \leq S_j$. Die Grenzen $S_0, ..., S_k$ sind in Abbildung 6.7 dargestellt. Die daraus

12	5	19	1	8	15	25

Abbildung 6.6.: Splitter-Feld aus dem der Binärbaum erstellt wird

erstellten Intervalle der Buckets sind in Abbildung 6.8 zu sehen.

S_0	S_1	S_2	S_3	S_4	S_5	S_6	S_7	S_8
$-\infty$	1	5	8	12	15	19	25	∞

Abbildung 6.7.: Sortiertes Sample mit Grenzen $-\infty$ und ∞

B_1	B_2	B_3	B_4	B_5	B_6	B_7	B_8
$[-\infty, 1]$	$[1, 5]$	$[5, 8]$	$[8, 12]$	$[12, 15]$	$[15, 19]$	$[19, 25]$	$[25, \infty]$

Abbildung 6.8.: Die einzelnen Intervalle der Buckets

Definition 6.2. (Der *Oversampling-Faktor*)
Ein Oversampling-Faktor α mit $\alpha \in \mathbb{N}$, $\alpha \geq 1$ gibt an, wie viele Samples der Größe $k-1$ aus der Eingabesequenz erstellt werden.

Durch den Oversampling-Faktor α wird es möglich, den Binärbaum robuster zu gestalten. Für eine Bucket-Anzahl von k wird ein Sample der Größe $(k-1)\cdot\alpha$ gewählt. Nach dem Sortieren können die Vielfachen von α benutzt werden, um das Splitter-Array zu erhalten. Für größere α erhöhen wir zwar die Kosten für das Sampling, können aber die Verteilung der Aufteilung in die Bereiche optimieren.

Klassifizierung und Aufteilung

Die Eingabesequenz durchläuft nun mithilfe von binärer Suche den Binärbaum und wird komplett klassifiziert. Das Vorgehen ist in 6.3 dargestellt. Während der Klassifizierung werden die Elemente in den Buckets gespeichert, diese müssen jeweils vorgehalten werden, weswegen Samplesort auch nicht In-Place arbeitet. Nachdem alle Elemente klassifiziert wurden, können die Inhalte der Buckets (in Reihenfolge der Bucket-Nummern) die Eingabesequenz überschreiben. Das Verfahren wird nun auf die einzelnen Buckets in der Eingabesequenz rekursiv angewandt und die Ergebnisse wieder konkateniert. Wichtig ist dabei, immer die kleinste Teilsequenz zuerst zu bearbeiten, um die Größe des Hilfsspeichers auf $\log N$ zu beschränken.

Algorithmus 6.3 Find Bucket (a.k.a. binäre Suche) aus der Super Scalar Samplesort-Implementation von Axtmann Weiß und Hass [26].

```
1: function FINDBUCKET(array splitters,integer key)
2:     integer i ← 1
3:     while i ≤ SAMPLESIZE do
4:         i ← 2 · i + (key > splitters[i − 1])        ▷ i fängt bei 1 an, splitters bei 0.
5:     return i − BUCKETCOUNT        ▷ i liegt zwischen BUCKETCOUNT und
   2 · BUCKETCOUNT − 1
```

6.1.2. Algorithmus

Algorithmus 6.4 Sample Sort

```
1: function SAMPLESORT(Array A[ℓ,..,r], integer k)
2:     if (r−ℓ+1)/k ist 'klein' then
3:         return smallSort(A[ℓ,...,r])
4:     sei (S_1,...,S_{ak−1}) ein zufälliges Sample aus A[ℓ,..,r]
5:     sortiere S
6:     (s_0, s_1, s_2, ..., s_{k−1}, s_k) ← (−∞, S_a, S_{2a}, ..., S_{(k−1)a}, ∞)
7:     for i ← l to r − ℓ + 1 do
8:         finde j ∈ {1,...,k} such that s_{j−1} < A[i] ≤ s_j
9:         platziere A[i] in Bucket b_j
10:    return concatenate(Samplesort(b_1, k), ..., Samplesort(b_k, k))
```

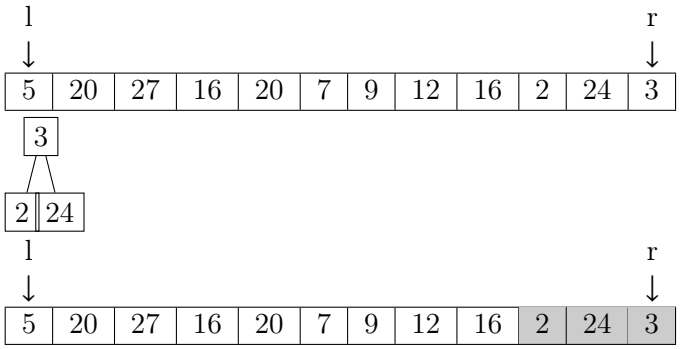
Der Algorithmus 6.4 basiert auf dem Pseudocode, welcher 2004 von Sanders und Winkel in ihrer Arbeit „Super Scalar Sample Sort“ [33] vorgestellt wurde und stellt das im vorherigen Abschnitt eingeführte Verfahren dar. Kleine Eingaben werden mit einem anderen Algorithmus wie zum Beispiel Quicksort oder Insertionsort sortiert, dies dient unter anderem als Rekursionsanker (Zeile 2-3). Danach wird ein Sample der Größe $s = \alpha k$ gewählt und sortiert, hierfür kann der selbe Algorithmus wie in Zeile 3 benutzt werden (Zeile 4-5). α stellt dabei den Oversampling-Faktor dar, die Vielfachen von α sowie die Grenzen $-\infty$ und ∞ werden genutzt, um das für die Klassifizierung benötigte Splitter-Feld zu erhalten (Zeile 6). Im Weiteren wird jedes Element der Eingabe klassifiziert und im entsprechenden Bucket gespeichert (Zeile 7-9). Anschließend werden die Bereiche rekursiv sortiert und konkateniert (Zeile 10).

Abbildung 6.9 zeigt eine Beispielsequenz in Samplesort. Das aktuelle Sample ist grau hervorgehoben mit dem Splitter-Feld dargestellt als Binärbaum. Der Schreib- und Lesevorgang in bzw. aus einem Bucket ist gelb hervorgehoben. Des Weiteren geben die Zeiger l und r die aktuellen Bereiche an.

6.1.3. Analyse

Nach der Publikation „Samplesort: Ein Sampling Ansatz zu minimalem Speicher-Baum Sortieren“ Frazer et. al. [12] liegt die erwartete Anzahl an Vergleichen für Samplesort bei: $2(N+1)\sum_{i=1}^{N}\frac{1}{i+1} - 2N = 1.39(N+1)\log N - 2.85N + 2.15$.

CONSTANTS
BASECASESIZE: 4 SAMPLESIZE: 3 MAXBUCKETS: 4

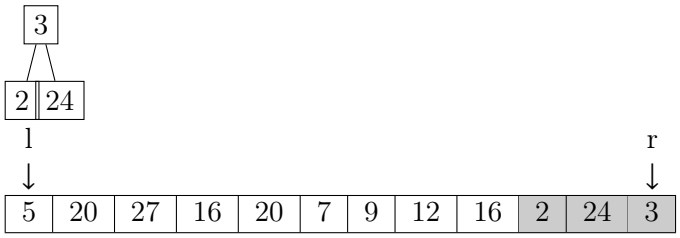


Bucket 0: leer

Bucket 1: leer

Bucket 2: leer

Bucket 3: leer

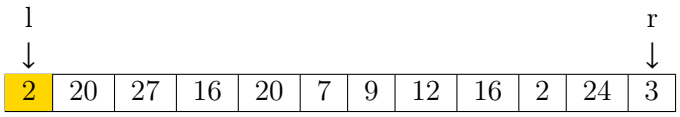


Bucket 0: 2

Bucket 1: 3

Bucket 2: 5 20 16 20 7 9 12 16 24

Bucket 3: 27

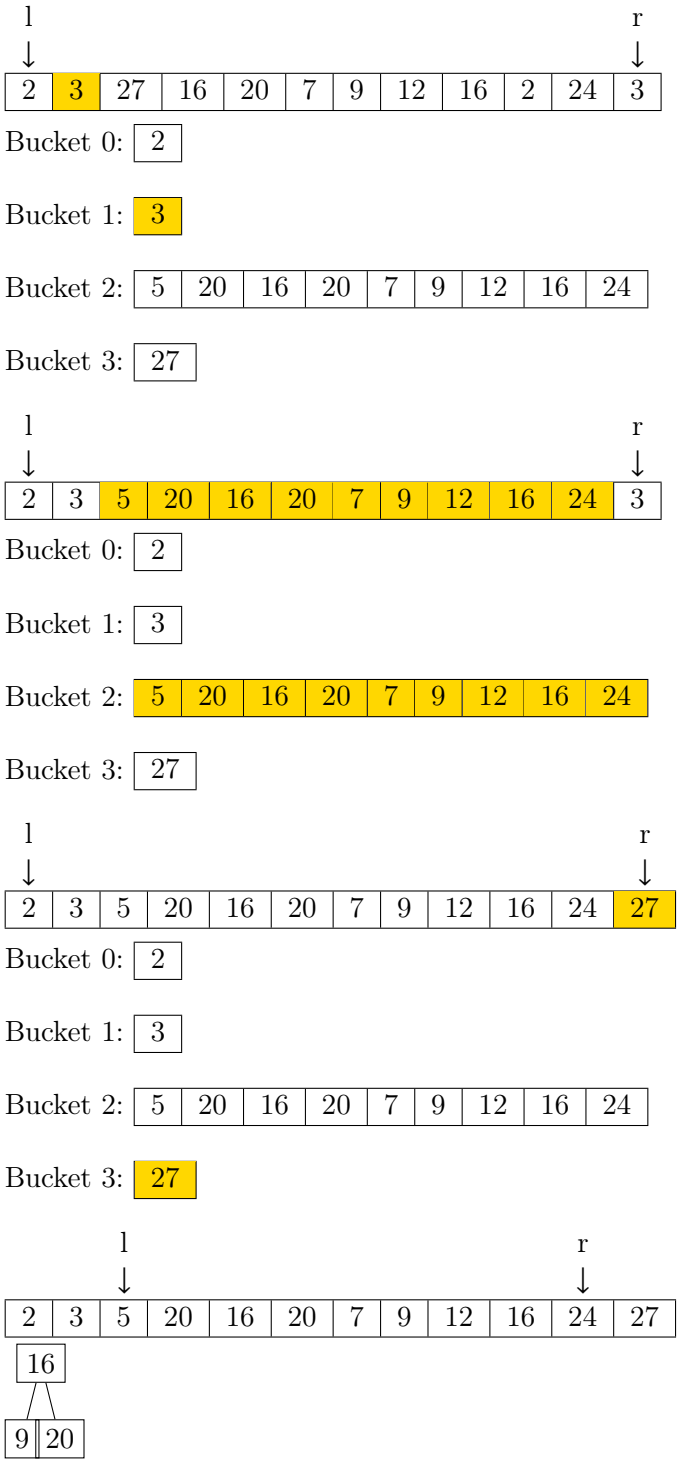


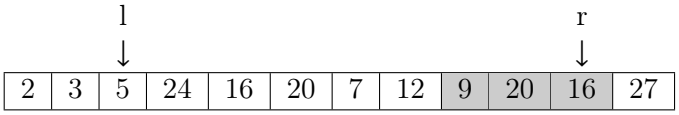
Bucket 0: 2

Bucket 1: 3

Bucket 2: 5 20 16 20 7 9 12 16 24

Bucket 3: 27


l
r
2 3 27 16 20 7 9 12 16 2 24 3
Bucket 0: 2
Bucket 1: 3
Bucket 2: 5 20 16 20 7 9 12 16 24
Bucket 3: 27
l
r
2 3 5 20 16 20 7 9 12 16 24 3
Bucket 0: 2
Bucket 1: 3
Bucket 2: 5 20 16 20 7 9 12 16 24
Bucket 3: 27
l
r
2 3 5 20 16 20 7 9 12 16 24 27
Bucket 0: 2
Bucket 1: 3
Bucket 2: 5 20 16 20 7 9 12 16 24
Bucket 3: 27
l
r
2 3 5 20 16 20 7 9 12 16 24 27
16
9 20

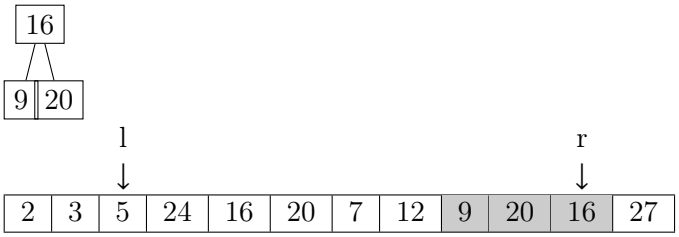


Bucket 0: leer

Bucket 1: leer

Bucket 2: leer

Bucket 3: leer

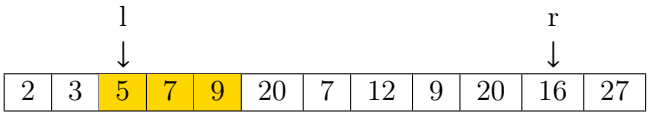


Bucket 0: 5 7 9

Bucket 1: 16 12 16

Bucket 2: 20 20

Bucket 3: 24

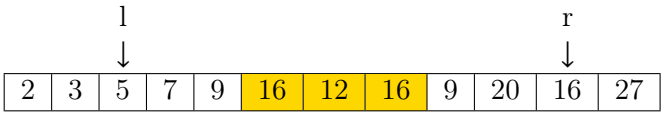


Bucket 0: 5 7 9

Bucket 1: 16 12 16

Bucket 2: 20 20

Bucket 3: 24



Bucket 0: 5 7 9

Bucket 1: 16 12 16

Bucket 2: 20 20

Bucket 3: 24

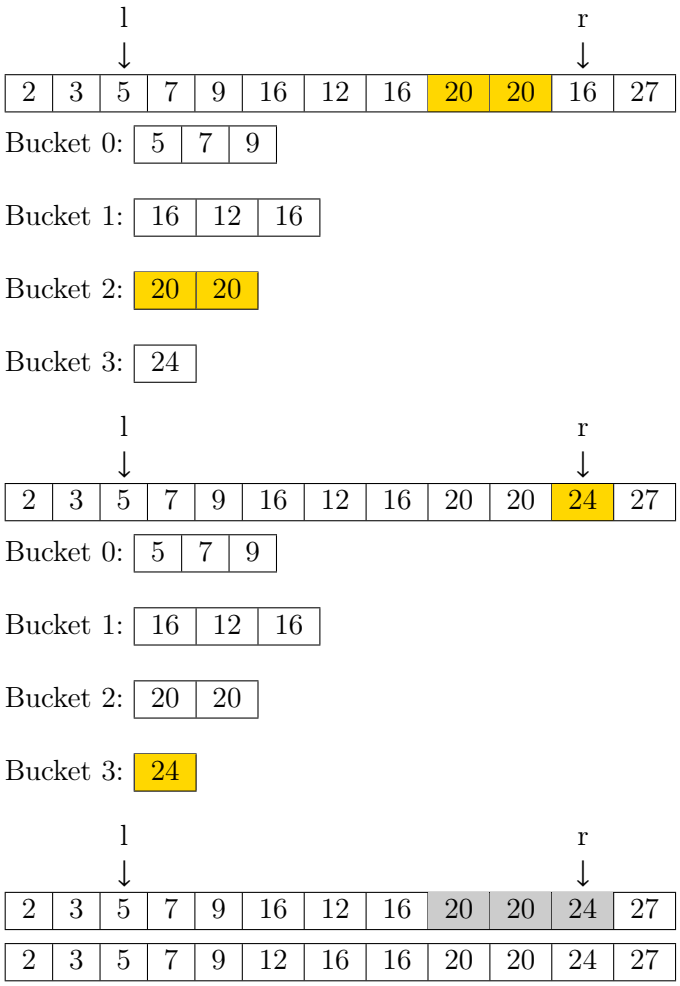


Abbildung 6.9.: Eine Beispielsequenz in Samplesort

Sprungfehler

Auch wenn die direkte Abfolge von Klassifizierung und Aufteilung Sprungfehler hervorruft, so ist die Klassifizierung, isoliert betrachtet, frei von Sprungfehlern. Dies wird durch das Benutzen eines Binärbaumes, bei dem die Indizes im Prozess der Traversierung nur mit vorherbestimmten Instruktionen erhöht werden, erreicht. Explizit wird hier die Instruktion *SETcc* benutzt (Umwandlung einer booleschen Variable zu einer Zahl, siehe 6.3 Zeile 4).

Cache-Effizienz

Durch die Nutzung eines Samples der Größe $k-1$ ist Samplesort Cache effizient: anstelle der $\log N$ Bewegungen in Quicksort werden nur $\log_k N$ Elementbewegungen benötigt.

6.2. Super Scalar Samplesort

Super Scalar Samplesort(S^3) ist eine Variante von Samplesort, die versucht, moderne Aspekte der Prozessorarchitektur zu berücksichtigen und Sprungfehler vermeidet. Mit einem Blick auf den vorherigen Algorithmus Samplesort (6.4) ist die Trennung der Klassifizierung in Zeile 8 von der Aufteilung in Zeile 9 die zentrale Idee in S^3. Somit wird ein hohes Maß an Instruction-level parallelism erreicht, da Datenabhängigkeiten und mögliche Sprungfehler vermieden werden. Das Vorgehen hat dabei folgenden Ablauf:

1. Sampling
2. Klassifizierung
3. Aufteilung

6.2.1. Das Verfahren

Wir brauchen ein zusätzliches Feld *out* der Größe N als temporären Speicher. Der Datenfluss zwischen der Eingabe und diesem Feld alterniert in den verschiedenen Stufen der Rekursion. Falls die finale Stufe ungerade ist, muss die derzeitige Eingabe mit den Werten aus *out* überschrieben werden. Des Weiteren brauchen wir ein Hilfsfeld o nach folgender Definition:

Definition 6.5. (Oracle)
Sei N die Länge der Eingabe mit $N = r - \ell + 1$. Definiere ein Hilfsfeld o der Größe N welches *Oracles* speichert. Dabei ist jedes Oracle mit Index i, also $o(i)$, ein Element welches den Bucket Index für das i-te Element in der Eingabe speichert.

Durch das Trennen der Zeilen 8 und 9 erhalten wir einen zweistufigen Ansatz. In der ersten Stufe, der Klassifizierung, werden die Elemente in der Eingabe nicht

verändert. Anstatt das Element direkt in einem Bucket zu speichern, wie es bei Samplesort der Fall war, wird im Feld o das Oracle für dieses Element gesetzt und die (virtuelle) Größe des jeweiligen Buckets erhöht.

Im zweiten Durchlauf können wir durch die mitgezählten Bucketgrößen die Bereiche der Buckets im alternierenden Feld festlegen. Danach werden die Elemente der Eingabe erneut durchlaufen, Element e_i wird in dem Bereich von Bucket $b_{o(i)}$ platziert.

Dieser zweistufige Ansatz, die Eingabesequenz erst komplett zu klassifizieren bevor man sie aufteilt, erzeugt zusätzliche Laufzeitkosten (Elemente müssen zweimal in den Speicher geladen werden), sowie Speicherkosten (Oracles müssen gespeichert werden). Auf der anderen Seite wird durch diesen Ansatz nach Sanders et. al. [33] nebenbei die Optimierung der *Schleifenspaltung* durchgeführt. Darunter versteht man die Spaltung einer Schleife in mehrere Schleifen mit gleichem Index-Intervall, wir erhalten dadurch erhöhten Instruction-level parallelism sowie weniger Cache-Störungen.

6.2.2. Algorithmus

Algorithmus 6.6 Klassifizierung

```
1: function Classification(Array A[ℓ,...,r], array splitters)
2:     for i ← ℓ; i < r; i++ do
3:         j ← 1
4:         while j ≤ SAMPLESIZE do
5:             j = 2j + (A[i] > splitters[j − 1])
6:         j = j − BUCKETCOUNT
7:         |b_j| ++
8:         o(i) = j
```

Der dargestellte Algorithmus stammt von Sanders und Winkel [33] und soll den Unterschied zu Zeile 8 im Samplesort-Algorithmus (6.4) verdeutlichen. Anstatt den Bucket-Index zurückzugeben, wird er in dem entsprechenden Oracle gespeichert. Zusätzlich sieht man die Erhöhung der Elemente in dem (virtuellen) Bucket.

Angenommen $B[j]$ wird mit $\sum_{i<j} |b_i|$ initialisiert, dann ist die Aufteilung durch die vorherige Klassifizierung in die Oracles nach Sanders und Winkel [33] wie folgt möglich:

```
1: for integer i ← ℓ; i < r; i++ do
2:     a'_{B[o(i)]++} = A[i]
```

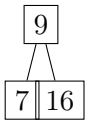
Abbildung 6.10 zeigt eine Beispielsequenz in S^3. Das aktuelle Sample ist grau hervorgehoben mit dem Splitter-Feld dargestellt als Binärbaum. Neben der Eingabesequenz wird das Feld *out*, mit welchem die Eingabe alterniert, sowie das Oracle-Feld dargestellt. Des Weiteren geben die Zeiger l und r die aktuellen Bereiche an.

CONSTANTS
BASECASESIZE: 4 SAMPLESIZE: 3 MAXBUCKETS: 4

beginIsHome: false

l ↓ r ↓

5	3	27	24	20	7	9	12	16	20	16	2
-	-	-	-	-	-	-	-	-	-	-	-
0	0	0	0	0	0	0	0	0	0	0	0



beginIsHome: false

l ↓ r ↓

16	9	7	24	20	27	3	12	5	20	16	2
-	-	-	-	-	-	-	-	-	-	-	-
2	1	0	3	3	3	0	2	0	3	2	0

beginIsHome: false

l ↓ r ↓

16	9	7	24	20	27	3	12	5	20	16	2
7	3	5	2	9	16	12	16	24	20	27	20
2	1	0	3	3	3	0	2	0	3	2	0

beginIsHome: false

l ↓ r ↓

16	9	7	24	20	27	3	12	5	20	16	2
7	3	5	2	9	16	12	16	24	20	27	20
2	1	0	3	3	3	0	2	0	3	2	0

beginIsHome: false

l ↓ r ↓

16	9	7	24	20	27	3	12	5	20	16	2
2	3	5	7	9	16	12	16	24	20	27	20
2	1	0	3	3	3	0	2	0	3	2	0

beginIsHome: false

l ↓ r ↓

16	9	7	24	20	27	3	12	5	20	16	2
2	3	5	7	9	16	12	16	24	20	27	20
2	1	0	3	3	3	0	2	0	3	2	0

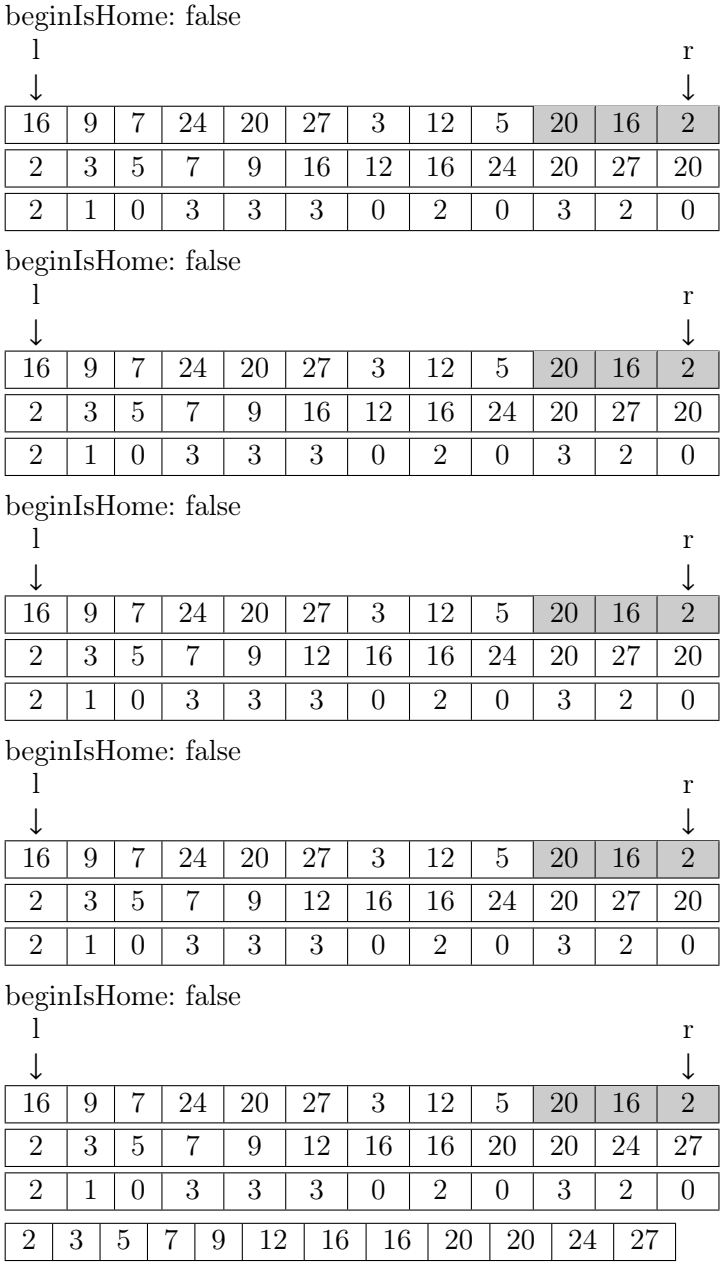


Abbildung 6.10.: Eine Beispielsequenz in S^3

6.2.3. Analyse

Nach der bereits erwähnten Abhandlung von Sanders und Winkel [33] liegen die erwarteten Vergleiche des Verfahrens S^3, mit einem zufälligen Sample der Größe $\Theta(\sqrt(N))$ bei $O(N \log N) + O(N)$. Für eine weiterführende Erklärung wird auf die entsprechende Arbeit verwiesen.

6.2.4. Abschluss

S^3ist zwar nicht In-Place, es gibt aber Optimierungen, die dies ermöglichen. Mit der Einführung von IPS^{4}o wird in dieser Arbeit bereits eine verbesserte Version vorgestellt die auch In-place ist, weswegen auf solche Optimierungen nicht eingegangen wird. In der Tat ist es jedoch so, dass eine In-Place Optimierung von S^3 die Inspiration für IPS^{4}o war.

6.3. In-Place Parallel Super Scalar Samplesort

6.3.1. Einleitung

Das Verfahren **I**n-place **P**arallel **S**uper **S**calar **S**ample**so**rt(IPS^{4}o) bildet den Zenit dieses Kapitels. Wir haben den Weg von der Grundidee, die Eingabesequenz nach Hoare in zwei Bereiche aufzuteilen, über die Idee, mehrere Buckets zu verwenden, bis hin zu der Idee, Puffer konstanter Größe zu benutzen, verfolgt und werden nun einen Algorithmus vorstellen, welcher

- Sprungfehler vermeidet,
- die In-Place Eigenschaft erfüllt,
- Cache-effizient ist,
- einen hohen Instruction-level parallelism ermöglicht und
- asymptotisch optimale Laufzeit hat.

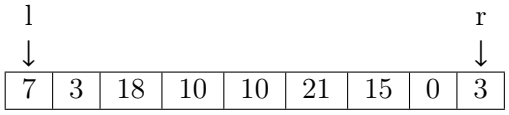
An Stelle von Oracles und Hilfsfeldern werden Puffer konstanter Größe benutzt. Die Idee ist dabei ähnlich zu der des besten Konkurrenten: BlockQuicksort (Kapitel 5). Wie in den vorherigen Abschnitten werden wir das Vorgehen wieder aufteilen. Dabei erhalten wir vier Phasen des Partitionionsvorganges, die im Folgenden dargestellt sind:

1. **Sampling**: Bestimme die Bucket Grenzen.
2. **Local classification**: Gruppiere die Eingabe in Blöcke sodass jedes Element eines Blockes zum selben Bucket gehört.
3. **Block permutation**: Bringe die Blöcke in die global richtige Reihenfolge.
4. **Cleanup**: Bereinige die Bucket Grenzen.

Auch wenn das Verfahren auf Parallelität optimiert ist und die Arbeit der einzelnen Phasen auf mehrere Threads verteilt werden kann, wird in dieser Arbeit nur die sequentielle Variante vorgestellt. Diese ist hinreichend, um ein gewisses Verständnis für den Algorithmus zu entwickeln. Für einen detaillierten Einblick in die parallele Variante wird auf die Arbeit „In-Place Parallel Super Scalar Samplesort“ von Axtmann et. al. [5] verwiesen. In Abbildung 6.14 ist die Idee hinter der Thread-Aufteilung für die lokale Klassifizierung kurz dargestellt.) Die im Folgenden abgebildeten Algorithmen wurden aus der im praktischen Teil dieser Arbeit vorgenommenen Implementierung entnommen. Diese basiert wiederum auf der Implementation von Sascha Witt [47]. Für die Laufzeiten gilt: $O(r-\ell)$ kann durch $max_{0\leq\ell\leq r\leq N}\{r-\ell\}$ mit $O(N)$ abgeschätzt und alle Konstanten können mit $O(1)$ abgeschätzt werden.

Die einzelnen Phasen werden in den nächsten Abschnitten jeweils vorgestellt und dabei durch ein begleitendes Beispiel unterstützt. Die Parameter und Eingabesequenz sind im Folgenden dargestellt.

CONSTANTS
BASECASESIZE: 4 SAMPLESIZE: 3 MAXBUCKETS: 4 PUFFERSIZE: 3



BUILD TREE

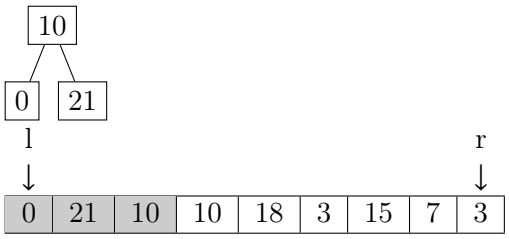


Abbildung 6.11.: Das Sampling in IPS⁴o

Abbildung 6.11 zeigt die erste Phase der Partitionierung, das Sampling. Diese ist analog zu dem Sampling der vorherigen Verfahren und soll die Grenzen der Buckets bestimmen. Das Vorgehen kann den vorherigen Abschnitten entnommen werden.

6.3.2. Klassifikation, Block-Vertauschung, Bereinigung der Grenzen

Klassifikation

In der zweiten Phase erstellen wir für jeden Bucket einen Puffer konstanter Größe. Jedes mal, wenn ein Element in einen Puffer geschrieben wird, d.h. in diesen Bucket klassifiziert wurde, zählen wir einen Zähler für den gleichen Bucket hoch. Außerdem führen wir einen Zeiger *write* ein, welcher am Anfang auf den linken Rand zeigt und angibt, in welchen Bereich der Eingabesequenz wir schreiben können. Die Eingabesequenz wird nun sequentiell durchlaufen, dabei werden die Elemente klassifiziert und in die jeweiligen Puffer geschrieben. Falls ein Puffer voll ist, wird er in die Eingabesequenz an die Position von *write* geschrieben. Danach wird *write* um die Puffergröße erhöht. Sobald wir alle Elemente durchlaufen haben, sind wir fertig. Dabei können klassifizierte Elemente teilweise noch in den Puffern geschrieben stehen, da diese noch nicht voll waren. Dies führt dazu, dass hinten in der Eingabesequenz „freie“ [1] Plätze bleiben. Die Eingabesequenz besteht abschließend aus Blöcken mit gleicher Größe wie die Puffer und jeder Block enthält nur Elemente eines Buckets, zusätzlich wissen wir, wie groß jeder Bucket ist. Durch das Vorhalten des *write*-Zeigers können wir zusätzlich auf den Beginn der freien Blöcke schließen. Mit all dem können wir nun für jeden Bucket die Zeiger d_i, w_i und r_i initialisieren. Der Zeiger d_i zeigt auf den Anfang von Block i, zusammen mit diesem geben die Zeiger w_i und r_i für jeden Bucket b_i folgende Invariante an:

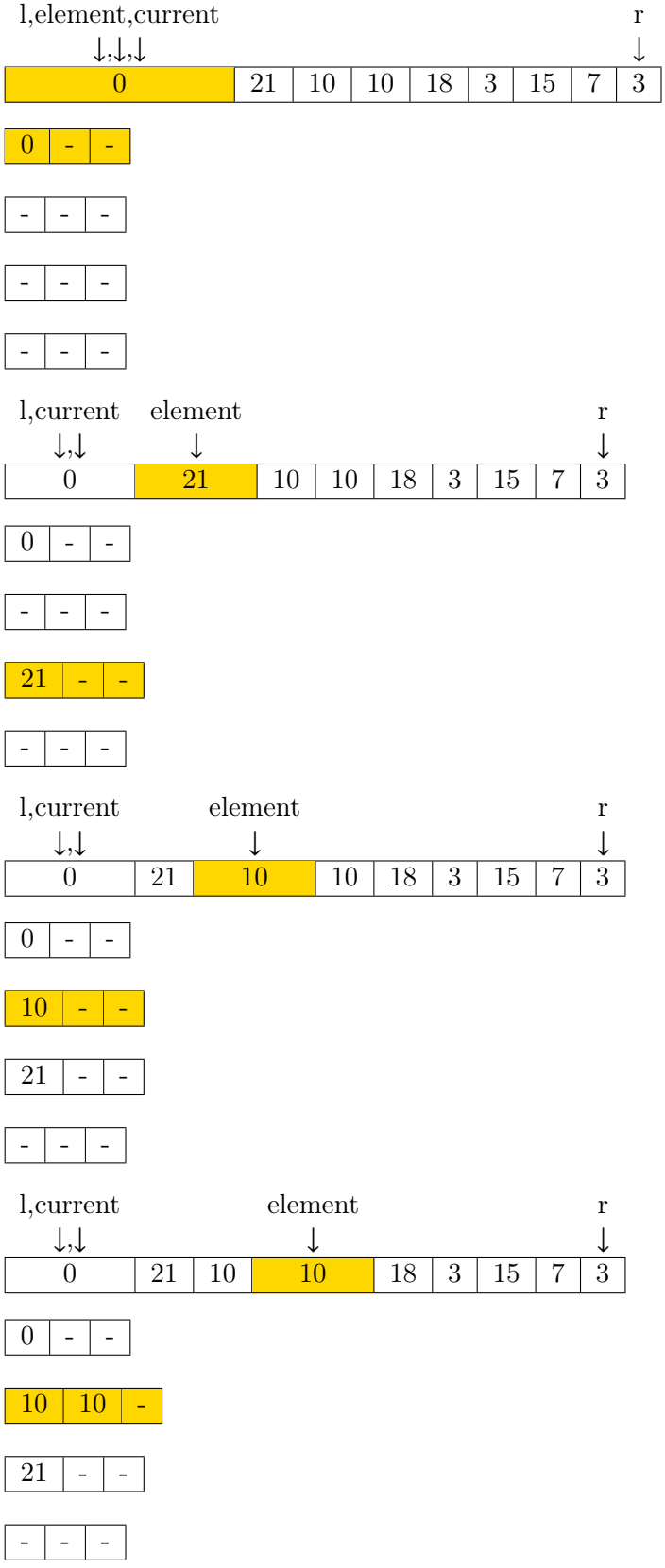
- Blöcke links von w_i (exklusiv) sind bereits korrekt platziert. Sie enthalten nur Elemente, welche zu b_i gehören: $[d_i, w_i)$.
- Blöcke zwischen w_i und r_i (inklusive) sind noch nicht bearbeitet, müssen also gegebenenfalls noch vertauscht werden: $[w_i, r_i]$.
- Blöcke rechts von $max(w_i, r_i + 1)$ (inklusive) sind leer: $(max(w_i, r_i + 1), d_{i+1})$.

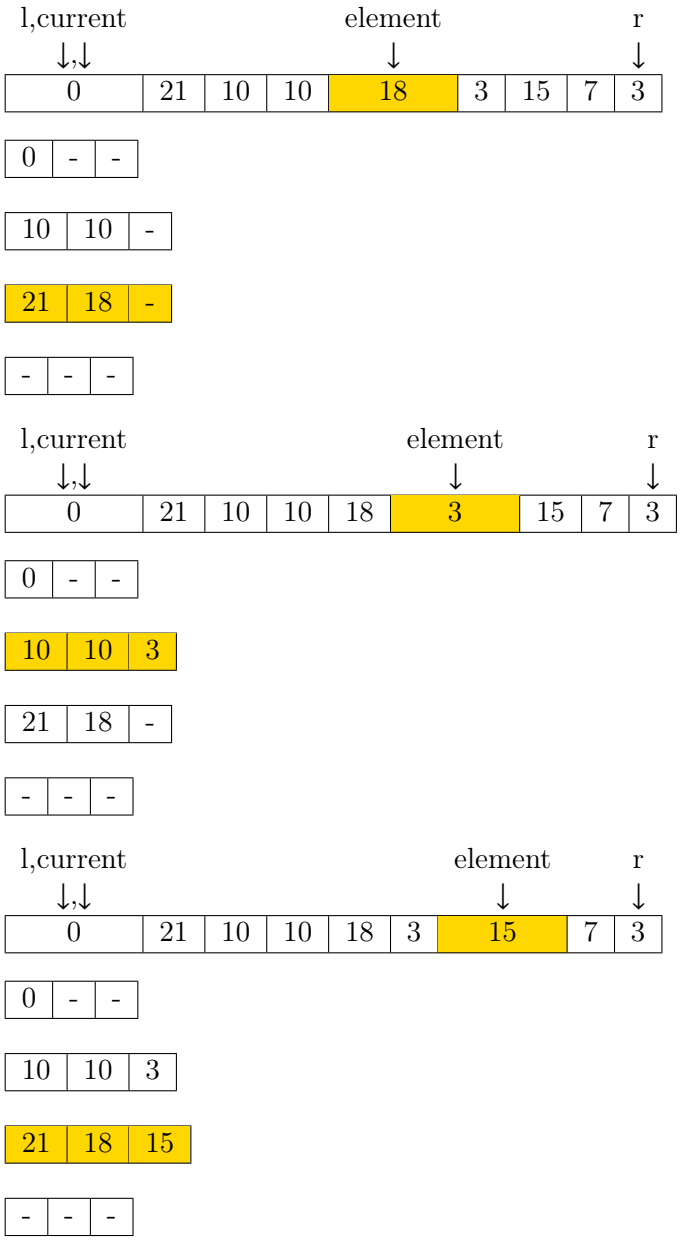
Der grobe Ablauf der Klassifikation ist in Abbildung 6.13, der Pseudocode in 6.7 und 6.8 dargestellt.

Abbildung 6.12 zeigt die Klassifikationsphase von IPS⁴o. Die Speicherung des aktuellen Elementes (angegeben durch den Zeiger *element*) im entsprechenden Puffer ist gelb hervorgehoben. Falls ein Puffer voll ist (rot hervorgehoben) muss dieser an die erste freie Position im Feld geschrieben werden (blau hervorgehoben). Diese Position wird durch den Zeiger *current* markiert. Der zu betrachtende Bereich der Sequenz ist wie gewohnt mit den Zeigern l und r begrenzt.

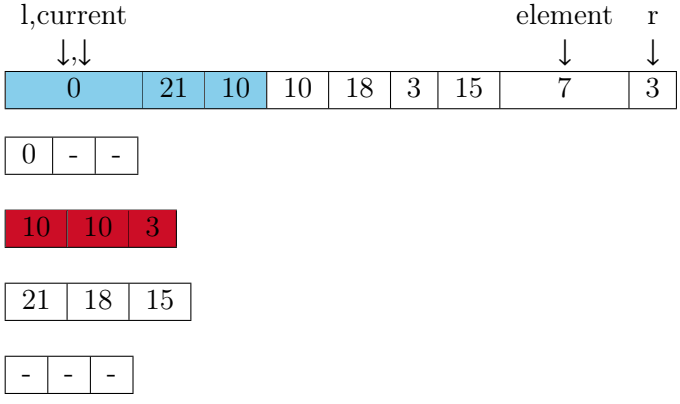
[1] Die Plätze enthalten natürlich noch die ursprünglichen Elemente, werden für uns aber als frei definiert und können überschrieben werden.

LOCAL CLASSIFICATION





LOCAL CLASSIFICATION FULL BUFFER



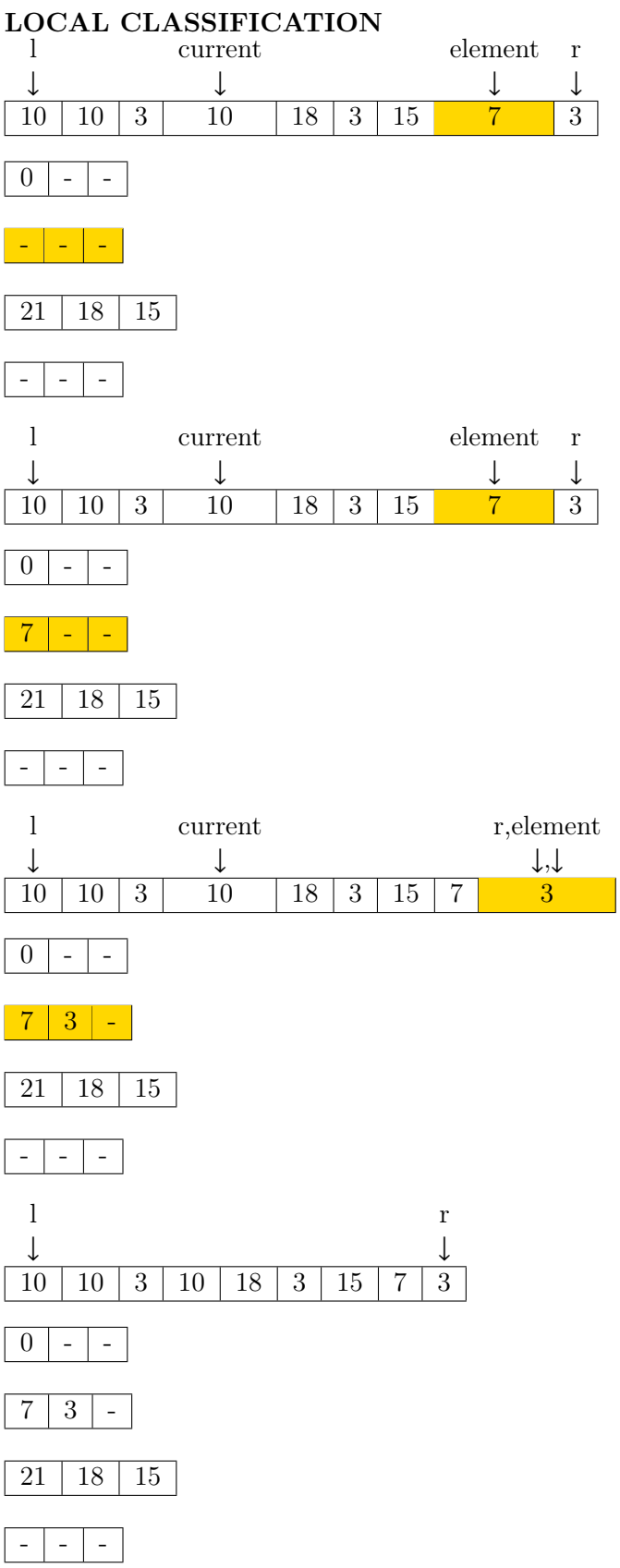


Abbildung 6.12.: Die Klassifikationsphase in IPS^{4}o

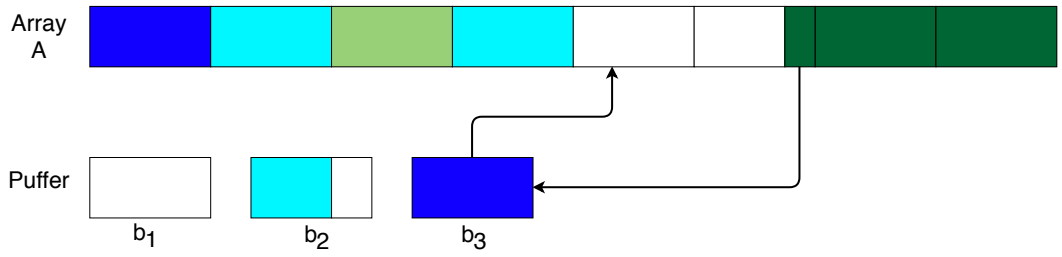


Abbildung 6.13.: Klassifikation in die drei Buckets (dargestellt durch Blau, hellblau, hellgrün), Weiße Blöcke sind leer und Grüne Blöcke sind noch unbearbeitet.

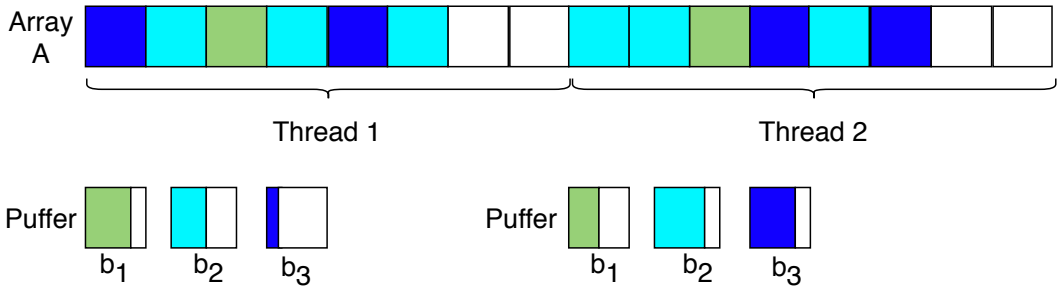


Abbildung 6.14.: Ausgang der lokalen Klassifikation: Dargestellt in der parallelen Variante, jeder Thread-Abschnitt besteht aus klassifizierten Blöcken, gefolgt von leeren Blöcken und hält für jeden Bucket einen eigenen Puffer vor.

Algorithmus 6.7 Gruppiere jeweils *PUFFERSIZE* Elemente mit gleicher Bucket Zugehörigkeit. Initialisiere die Bucket-Zeiger d_i, w_i und r_i

1: **function** LOCAL CLASSIFICATION($A[\ell,...,r]$)
2: $myFirstEmptyBlock \leftarrow classify(A[\ell,...,r])$
3: setze die d_i's auf die Anfänge der Buckets und speichere diese in *bucketStart*
4: setze die w_i's und r_i's, jeweils auf den nächsten Block aufgerundet und speichere diese in *bucketPointers*

Abbildung 6.15.: Instruktionen in *localClassification*

Zeile	*Kosten der Instruktionen*
2	$O(r-\ell)$
3	$O(numBuckets) = O(1)$
4	$O(numBuckets) = O(1)$

Insgesamt: $2 \cdot O(1) + O(r-\ell) = O(N)$

Algorithmus 6.8 Gruppiere jeweils *PUFFERSIZE* Elemente mit gleicher Bucket-Zugehörigkeit.

```
1:  function CLASSIFY(A[ℓ,...,r])
2:      int write ← ℓ
3:      for int i ← ℓ; i < r; i++ do
4:          int bucket ← (Classifier → A[i])
5:          if Puffer[bucket] ist voll then
6:              Schreibe Puffer[bucket] in Eingabesequenz (an Index write)
7:              Erhöhe bktsize[bucket] um die PUFFERSIZE
8:          Füge A[i] in Puffer[bucket] ein
9:      for int i ← 0; i < numBuckets; i++ do
10:         Erhöhe bktsize[i] um die Anzahl der verbleibenden Elemente in Puffers[i]
11:     return write − l
```

Abbildung 6.16.: Instruktionen in *classify*

Zeile	*Kosten der Instruktionen*
2	$O(1)$
3	$O(r-\ell)$
$\cdot$ 4	$O(\log(BUCKETCOUNT) = O(1)$
$\cdot$ 5	$O(1)$
$\cdot$ 6	$O(PUFFERSIZE) = O(1)$
$\cdot$ 7	$O(1)$
$\cdot$ 8	$O(1)$
9	$O(numBuckets) = O(1)$
10	$O(1)$

Insgesamt: $3 \cdot O(1) + O(r-\ell) \cdot O(1) = O(N)$

Block-Vertauschung

In der dritten Phase werden die in der lokalen Klassifikation erstellten Blöcke in die global richtige Reihenfolge gebracht. Dafür müssen wir zwei *Tauschpuffer* einführen. Diese haben die gleiche Größe wie die vorherigen Puffer und sorgen dafür, dass wir Blöcke tauschen können ohne Daten zu überschreiben, welche wir noch brauchen. Als Erstes wird ein Start-Bucket festgelegt, aus dem wir unseren ersten Block bekommen. Sollte der Start-Bucket im Laufe des Algorithmus keine weiteren freien Blöcke mehr vorhalten, so müssen wir zu dem nächsten Bucket rotieren. Dies geschieht genau dann, wenn im jeweiligen Bucket b_i für die Zeiger $read_i$ und $write_i$: $read_i < write_i$ gilt. Den so betrachteten Bucket nennen wir *Rotationsbucket*. In den anderen Fällen hat der Start-Bucket noch einen weiteren Block, also einen Block zwischen $write_i$ und $read_i$, welcher von uns in den ersten freien Tauschpuffer gelesen wird. Da aus diesem Bucket nun ein Block gelesen wurde, muss der Zeiger $read_i$ um die Puffergröße

erhöht werden. Dies ist dargestellt in Schritt 3 und 4 von Abbildung 6.19. Der in den Tauschpuffer geladene Block muss nun erneut klassifiziert werden, da wir, um die In-Place Eigenschaft zu erfüllen, nicht gespeichert hatten, welche Blöcke zu welchem Bucket gehören. Diese Klassifizierung des Blocks wird *Zielbucket* genannt. Für diesen gibt es nun zwei Möglichkeiten:

1. Der Zielbucket hat noch mindestens einen nicht betrachteten Block, d.h. $read \geq write$.
2. Der Zielbucket hat nur noch korrekte und leere Blöcke, d.h. $read < write$.

Eine andere Möglichkeit gibt es nicht, da wir die exakten Bucket Größen wissen und die w_i bzw. r_i danach initialisiert haben. Als Erstes erhöhen wir für den Zielbucket den *write* Zeiger, da wir in jedem Fall in diesen Bucket schreiben.

Im ersten Fall (dargestellt in Abbildung 6.18) können wir den ersten noch nicht betrachteten Block in den zweiten Tauschpuffer lesen. Dadurch können wir diesen Block ohne Bedenken mit dem Block aus dem ersten Tauschpuffer überschreiben (vgl. 6.10). Im zweiten Fall (dargestellt in Abbildung 6.19) können wir zwar den Block im Tauschpuffer ohne Probleme in die Eingabesequenz schreiben, da wir dort nur leere Blöcke überschreiben, wir erhalten jedoch im Austausch keinen Neuen Block in unserem Tauschpuffer, wodurch wir uns einen neuen holen müssen. Dafür dient uns der derzeitige Rotationsbucket, mithilfe dessen wir - wie am Anfang - einen neuen Block in den Tauschpuffer bekommen und damit auch einen neuen Zielpuffer (vgl. 6.11). Anschließend wiederholen wir unser Vorgehen mit dem neuen Block im Tauschpuffer und dem neuen Zielbucket.

Abbildung 6.17 zeigt die (gekürzte) Block-Vertauschungsphase in IPS^{4}o. Die Zeiger r_i, w_i, d_i geben die einzelnen Grenzen der Invariante (siehe Abschnitt 6.3.2) an. Der zu betrachtende Bereich der Sequenz ist wie gewohnt mit den Zeigern l und r begrenzt.

Der Zeiger *bRead* gibt den aktuellen Bucket an, aus welchem als nächstes in den Tauschpuffer geladen werden kann. Dieser Ladevorgang ist in gelb hervorgehoben. Der Zeiger *bDest* zeigt wiederum den Bucket an, in welchen der Block im Tauschpuffer geschrieben werden kann. Auch hier ist, diese Mal der Schreibvorgang, in gelb hervorgehoben. Dabei werden die Grenzen der einzelnen Buckets mit deutlicheren Linien markiert.

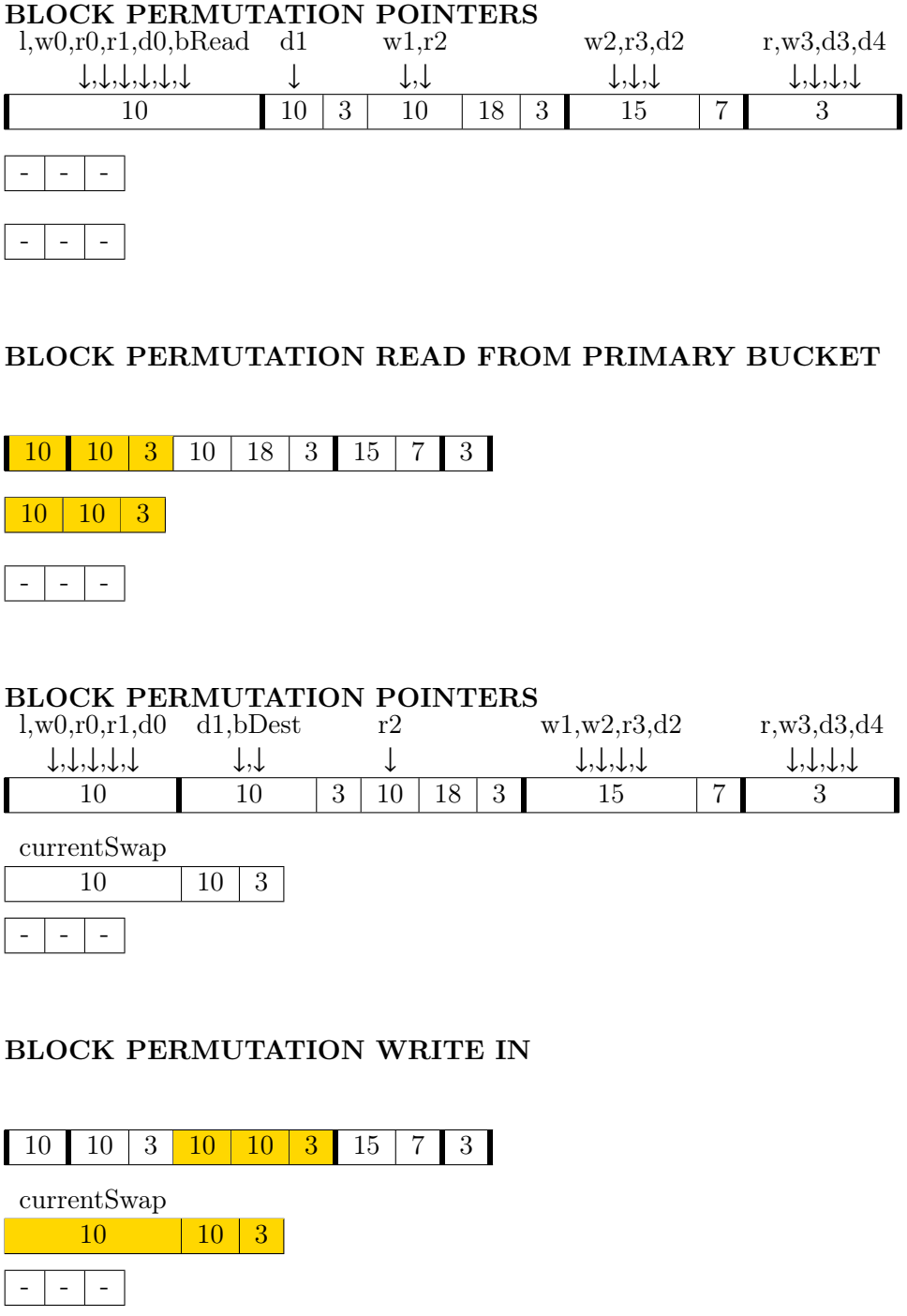


Abbildung 6.17.: Die (gekürzte) Block-Vertauschung in IPS⁴o

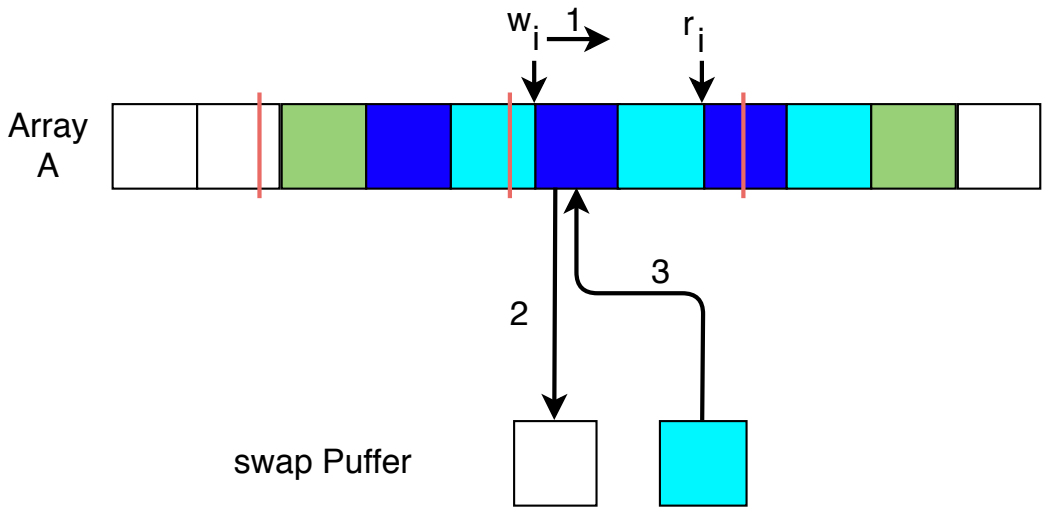


Abbildung 6.18.: Erste Möglichkeit im Zielbucket: es existiert noch mindestens ein nicht betrachteter Block, d.h. $read \geq write$. Die Abbildung stammt aus der bereits erwähnten Abhandlung von Axtmann et. al. [5].

Algorithmus 6.9 Vertausche die Blöcke in die global richtige Reihenfolge

```
1: function BLOCK PERMUTATION(A[ℓ,...,r])
2:     readBucket ← 0
3:     int maxOff ← erster Index, welcher einen Overflow hervorrufen würde
4:     for int count ← numBuckets to 0 do
5:         int destBucket
6:         while destBucket ← classifyAndReadBlock(A[ℓ,...,r],readBucket) not
   −1 do
7:             int currentSwap ← 0
8:             while destBucket ← swapBlock(A[ℓ,...,r],maxOff,destBucket,currentSwap)
   not −1 do
9:                 currentSwap ← (currentSwap + 1)%2;
10:        readBucket ← (readBucket + 1)%numBuckets
```

Algorithmus 6.10 Lade einen neuen Block in den Tauschpuffer und erhalte einen neuen Zielbucket

```
1: function CLASSIFYANDREADBLOCK(A[ℓ,...,r],readBucket)
2:     hole write und read von bucketPointers[readBucket]
3:     verringere read in bucketPointers[readBucket] um SAMPLESIZE      ▷ das
   initialisierte read wird nicht verringert.
4:     if read < write then
5:         return -1
6:     Lese die Eingabesequenz (an Index read + begin) in den ersten swapPuffer
7:     return (Classifier → das erste Element im ersten swapPuffer)
```

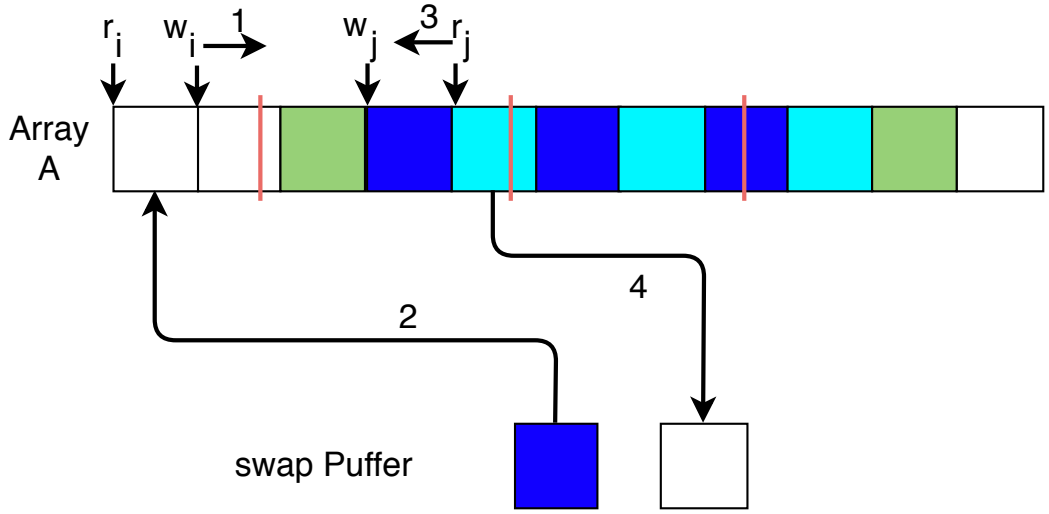


Abbildung 6.19.: Zweite Möglichkeit im Zielbucket: ex existieren nur noch korrekte und leere Blöcke, d.h. $read < write$. Die Abbildung stammt aus der bereits erwähnten Abhandlung von Axtmann et. al. [5].

Algorithmus 6.11 Füge den Block im Tauschpuffer in den Zielbucket ein, erhalte gegebenenfalls einen neuen Block

```
1:  function SWAPBLOCK(A[ℓ,...,r], maxOff, destBucket, currentSwap)
2:      do
3:          hole write und read von bucketPointers[readBucket]
4:          erhöhe write in bucketPointers[readBucket] um SAMPLESIZE      ▷ das
    initialisierte write wird nicht erhöht.
5:          if write > read then
6:              if write ≥ maxOff then
7:                  schreibe den aktuellen swapPuffer in den overflowPuffer
8:                  return -1
9:              schreibe den aktuellen swapPuffer in die Eingabesequenz (an Index
    write + ℓ)
10:             return -1
11:         newDestBucket ← (Classifier → A[ℓ + write])      ▷ Prüfe ob der Block
    bewegt werden muss
12:     while newDestBucket equals destBucket
13:     schreibe die Eingabesequenz (an Index ℓ + write) in den nicht aktuellen
    swapPuffer
14:     schreibe den aktuellen swapPuffer in die Eingabesequenz (an Index ℓ+write)
15:     return newDestBucket
```

Bereinigung der Grenzen

In dieser Phase werden die Buckets um ihre Grenzen herum aufgeräumt. Hier betrachten wir die Möglichkeit eines Overflows (siehe 6.11: Zeile 7 und 6.13: Zeile 5) und fügen die restlichen Elemente der Puffer in die entsprechenden Buckets. Des Weiteren,

Abbildung 6.20.: Instruktionen in *BlockPermutation*

Zeile	*Kosten der Instruktionen*
2	$O(1)$
3	$O(1)$
4	$O(numBuckets)$
	$\cdot \left\{ \begin{array}{l\|l} 5 & O(1) \\ & \#classifyAndRead \cdot \left\{ \begin{array}{l\|l} 6 & O(1) \\ 7 & O(1) \\ & \#swapBlock \cdot \left\{ \begin{array}{l\|l} 8 & O(1) \\ 9 & O(1) \end{array} \right\} \\ 10 & O(1) \end{array} \right\} \end{array} \right\}$

Insgesamt:
$3 \cdot O(1) + O(numBuckets) \cdot (O(1) + \#classifyAndRead \cdot (O(1) + \#swapBlock \cdot (O(1))))$

Abbildung 6.21.: Instruktionen in *classifyAndRead*

Zeile	*Kosten der Instruktionen*
2	$O(1)$
3	$O(1)$
4	$O(1)$
5	$O(1)$
6	$O(PUFFERSIZE)$
7	$O(\log(numBuckets))$

Insgesamt: $4 \cdot O(1) + O(PUFFERSIZE) + O(\log(numBuckets)) = O(1)$

da die Bucketgrößen nicht immer ein Vielfaches der Blockgrößen sind, werden hier die Elemente am Anfang des nächsten Buckets (genannt *head*) betrachtet, diese könnten noch zum aktuellen Bucket gehören. Dies ist in Abbildung 6.24 dargestellt.

Abbildung 6.23 zeigt die (vereinfachte) Darstellung der Bereinigung der Grenzen in IPS^{4}o. Der zu betrachtende Bereich der Sequenz ist wie gewohnt mit den Zeigern l und r begrenzt. Dabei werden die Grenzen der einzelnen Buckets mit deutlicheren Linien markiert.

Abbildung 6.22.: Instruktionen in *swapBlock*

Zeile	*Kosten der Instruktionen*
	$\#sameBucket \cdot \{ \boxed{2-13} \; \boxed{O(max(PUFFERSIZE, \log(numBuckets)))} \}$
14	$O(PUFFERSIZE)$
15	$O(PUFFERSIZE)$
16	$O(1)$

Insgesamt: $3 \cdot O(1) + \#sameBucket \cdot (O(1))$

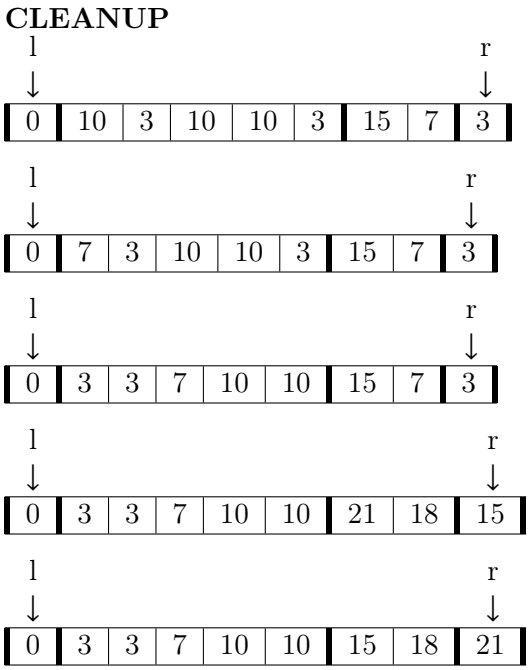


Abbildung 6.23.: Die Bereinigung der Grenzen in IPS⁴o

Algorithmus 6.12 Bereinige die Bucket Grenzen um Overflows, Elemente im Head des nächstens Buckets und füge die verbleibenden Puffer-Elemente ein.

```
1: function CLEANUP(A[ℓ,...,r], firstBucket, lastBucket, overflowBucket)
2:     for int i ← 0; i < numBuckets; i++ do
3:         if Bucket i hat Overflow then
4:             behandle Elemente im Overflow-Puffer
5:         else if zu Bucket i gehören Elemente von dem Head in Bucket i + 1 then
6:             behandle Elemente in Head_{i+1}
7:         if Bucket i hat noch Elemente in Puffer i gespeichert then
8:             behandle Elemente in Puffer_i
```

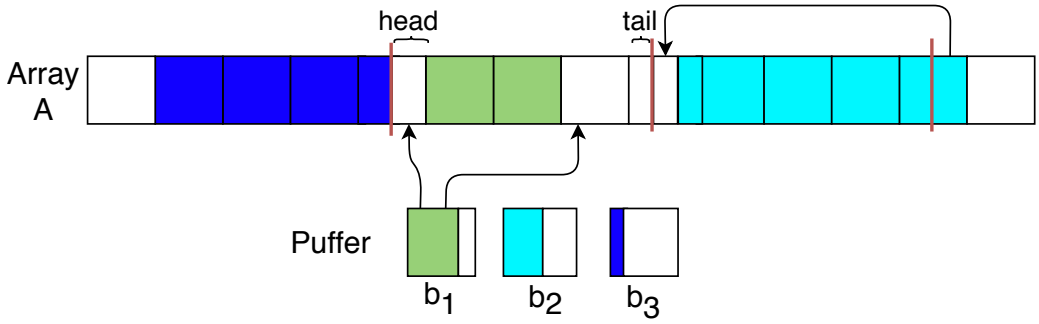


Abbildung 6.24.: Bereinigung der Bucket-Grenzen um Elemente im Head des nächsten Buckets und Einfügung der verbleibenden Puffer-Elemente.

Abbildung 6.25.: Instruktionen in *cleanup*

Zeile	*Kosten der Instruktionen*
2	$O(numBuckets)$
.	3: $O(1)$
	4: $O(PUFFERSIZE)$
	5: $O(1)$
	6: $O(PUFFERSIZE)$
	7: $O(1)$
	8: $O(PUFFERSIZE)$

Insgesamt: $numBuckets \cdot (3 \cdot O(1) + 3 \cdot O(PUFFERSIZE)) = O(1)$

6.3.3. IPS^{4}o

Nachdem wir nun die entsprechenden Schritte eingeführt haben, werden wir diese in der Methode *partition* zusammenfassen und uns die Hauptmethode anschauen.

Algorithmus 6.13 Aufteilung der Eingabesequenz in rekursiv bearbeitbare Teilsequenzen

1: **function** PARTITION($A[\ell, ..., r]$)
2: $numBuckets \leftarrow buildClassifier(A[\ell, ..., r], SAMPLESIZE, MAXBUCKETS)$
3: $localClassification(A[\ell, .., r])$
4: $blockPermutation(A[\ell, .., r])$
5: int $overflowBucket \leftarrow$ erster Bucket welcher einen Overflow hervorrufen könnte.
6: $cleanup(A[\ell, .., r], 0, numBuckets, overflowBucket)$

Die Methode *buildClassifier* ist (hier) eine Blackbox, sie wählt das Sample aus der Sequenz und erstellt daraus den Binärbaum. Zurückgegeben wird dann die Anzahl an Buckets. Im Folgenden kann der erstellte Classifier benutzt werden, um ein Element zu klassifizieren, z.B. int $bucket \leftarrow (Classifier \rightarrow A[i])$, um den Bucket für das Element an Position i zu finden und in $bucket$ zu speichern ($0 \leq bucket < MAXBUCKETS$).

Den Abschluss bildet die Methode IPS^{4}o. Dies ist die Methode, welche so lange rekursiv mit den einzelnen Bucket-Grenzen aufgerufen wird, bis die Eingabesequenz sortiert ist. Dabei arbeitet sie immer auf dem selben Feld, als Rekursionsanker dient ein Basis-Sortierverfahren, in diesem Fall Insertionsort.

Algorithmus 6.14 (sequentiell) In-Place Parallel Super Scalar Samplesort

```
1:  function IPS⁴o(A[ℓ,...,r])
2:      n ← r − ℓ + 1;
3:      if n ≤ 2 · BASECASESIZE then
4:          Insertionsort(A[ℓ,...,r]);
5:          return
6:      numBuckets ← partition(A[ℓ,...,r]);
7:      for int i = 0; i < numBuckets; i++ do
8:          start ← bucketStart[i]
9:          stop ← bucketStart[i + 1] − 1
10:         if stop − start > 2BASECASESIZE then
11:             IPS⁴o(A[ℓ + start,...,ℓ + stop])
```

6.3.4. Analyse

Laufzeit IPS^{4}o

Die Laufzeit von IPS^{4}o liegt nach Axtmann et. al. [5] für zufällige Eingaben mit hoher Wahrscheinlichkeit bei $O(N \log N)$. Auf den Beweis wird im Rahmen dieser Arbeit verzichtet.

Laufzeit Partitionierung

Theorem 6.15. *(Laufzeit der Partitionierung)*
Sei N die Länge der Eingabe mit $N = r - \ell + 1$. Die Laufzeit der Partitionierung liegt dann in $O(N)$.

Für den Beweis führen wir zuerst folgendes Hilfslemma ein:

Lemma 6.16. *(Laufzeit der Block-Vertauschungen)*
Sei N die Länge der Eingabe mit $N = r - \ell + 1$. Die Laufzeit der Block-Vertauschungen kann dann durch $O(N)$ abgeschätzt werden.

Beweis: Die Schleifen in den Zeilen 6 und 8 der Methode *BlockPermutation* terminieren, sobald die jeweilige Methode *classifyAndReadBlock* oder *swapBlock* -1 zurückgibt. Dies geschieht, falls für den jeweiligen Bucket b_i gilt: $write_i > read_i$ (Zeile 6 in *swapBlock*, Zeile 4 in *classifyAndReadBlock*). Dabei werden die jeweiligen *write* Zeiger nur erhöht und die *read* Zeiger nur verringert (Zeile 5 in *swapBlock*, Zeile 3 in *classifyAndReadBlock*). Dies geschieht mindestens einmal pro jeweiligem Schleifendurchlauf. Man kann nach oben abschätzen, dass alle $write_i$ mit 0 initialisiert

werden und alle $read_i$ mit r. Die Anzahl der Schleifendurchläufe von *swapBucket* und $classifyAndRead$ zusammen kann nun mit:
$O(numBuckets \cdot \frac{r-\ell+1}{SAMPLESIZE}) = O(r-\ell+1)$ begrenzt werden. Die Schleifendurchläufe in der Methode *swapBlock* (Zeile 2-13) wird auch ohne die eigentliche Bedingung, nur durch die Inkrementierung von $write_i$ und den *return* Statements automatisch begrenzt.

Es folgt für die Block-Vertauschungen:
$3 \cdot O(1) + O(numBuckets) \cdot (O(1) + \#classifyAndRead \cdot (O(1) + \#swapBlock \cdot (\#sameBucket \cdot (O(1))))) = O(r-\ell) = O(N)$ □
(Achtung: z.B. $\#classifyAndRead = \frac{r-\ell+1}{2}$ bedeutet nicht, dass die Methode $classifyAndRead$ in jeder Iteration auch $\frac{r-\ell+1}{2}$ mal ausgeführt werden kann, sondern ingesamt über alle Iterationen $\frac{r-\ell+1}{2}$ mal ausgeführt werden kann.)

Für die Laufzeit der Partitionierung folgt mit 6.16 und den vorherigen Laufzeitabschätzungen:

Abbildung 6.26.: Instruktionen in *partition*

Zeile	*Kosten der Instruktionen*
2	$O(1)$
3	$O(N)$
4	$O(N)$
5	$O(1)$
6	$O(1)$

Insgesamt: $3 \cdot O(1) + 2 \cdot O(N) = O(N)$ □

Korrektheit IPS⁴o

Theorem 6.17. *(Korrektheit IPS⁴o)*
Der Algorithmus IPS⁴o ist korrekt.

Auf den Beweis wird im Rahmen dieser Arbeit verzichtet.

Korrektheit der Partitionierung

Theorem 6.18. *(Korrektheit Partitionierung)*
Die Partitionierung von IPS⁴o ist korrekt.

Auf den Beweis wird im Rahmen dieser Arbeit verzichtet.

6.4. Überblick

Samplesort:

- Ideal für Parallele Systeme und GPUs
- Basiert auf dem Quicksort-Verfahren
- Cache-effizient durch $\log_k N$ Elementbewegungen
- Vorherbestimmte Instruktionen in der Klassifizierung
- Benutzung eines Binärbaumes mit Buckets, Oversampling-Faktor sorgt für Trade-Off zwischen Mehraufwand und Genauigkeit
- Asymptotisch optimale Laufzeit
- Extra Speicher:
  - $O(N)$ (Buckets)
  - $O(2 \cdot SAMPLESIZE)$ (Vorhalten des Samples und des Binärbaums (Splitter-Feld))
  - $O(BUCKETCOUNT)$ (Vorhalten der Bucket-Größen)
  - $O(\log N)$ (Rekursionsstack)
- $SAMPLESIZE = 2^i - 1$ für ein $i > 0$, $i \in \mathbb{N}$ wobei gilt: $BUCKETCOUNT = 2^i$

S^3:

- „Gut durchdachte Implementierung für Standard CPUs“
- Basiert auf dem Samplesort-Verfahren
- Cache-effizient durch $\log_k N$ Elementbewegungen
- Zweistufiger Ansatz → Schleifenspaltung, Vermeidung von Sprungfehlern
- Benutzung eines Binärbaumes mit Buckets, Oversampling-Faktor sorgt für Trade-Off zwischen Mehraufwand und Genauigkeit
- Oracles → Vergleiche und Sprünge werden entkoppelt
- Asymptotisch optimale Laufzeit
- Extra Speicher:
  - $O(N)$ (Hilfsarray (alterniert mit Eingabefeld))
  - $O(N)$ (Oracles)
  - $O(2 \cdot SAMPLESIZE)$ (Vorhalten des Samples und des Binärbaums (Splitter-Feld))
  - $O(BUCKETCOUNT)$ (Vorhalten der Bucket-Größen)
  - $O(\log N)$ (Rekursionsstack)

- $SAMPLESIZE = 2^i - 1$ für ein $i > 0$, $i \in \mathbb{N}$ wobei gilt: $BUCKETCOUNT = 2^i$

IPS^{4}o:

- Basiert auf dem S^3-Verfahren
- Puffer konstanter Größe (ähnliche Idee wie in BlockQuicksort)
- Vermeidung von Sprungfehlern
- Cache-effizient
- Hoher Instruction-level parallelism
- Aufteilung auf mehrere Threads möglich
- Asymptotisch optimale Laufzeit
- Extra Speicher:
  - In-Place

7. Vergleich von Sprungfehler-optimierten Sortierverfahren

In dieser Arbeit wurden mittlerweile viele Ansätze und Verfahren vorgestellt, um Quicksort zu optimieren. Vermehrt wurde sich dabei darauf konzentriert, Sprungfehler zu vermeiden. Die vielseitig benutzte Quicksort-Optimierung std::sort ist nicht im Hinblick auf Sprungfehler optimiert und wird im Folgenden als Maßstab zu den vorgestellten Verfahren BlockQuicksort (siehe Abschnitt 5.1), BlockLomuto (siehe Unterabschnitt 5.2.1), Dual-Pivot BlockLomuto (siehe Unterabschnitt 5.2.2) und IPS^{4}o (siehe Abschnitt 6.3) agieren. Die Aufgabe für den praktischen Teil dieser Arbeit war die Erstellung eines Programmes zur didaktischen Darstellung verschiedener Sortierprozesse. In der Entwicklung stand dabei die Nutzererfahrung im Fokus, um ein bestmögliches Verständnis für die Verfahren zu ermöglichen. Durch diese Anforderungen sind die Implementierungen für realistische Laufzeitvergleiche untauglich. Um die Verfahren in der Praxis trotzdem realistisch vergleichen zu können stützt sich dieses Kapitel hauptsächlich auf die Analyse im BlockQuicksort-Paper von Aumüller und Hass [4] sowie jene im BlockQuicksort-Paper von Edelkamp und Weiß [10]. Edelkamp und Weiß benutzen dabei eine Intel Core i5-2500K CPU mit 3.30 GHz und 4 Kernen, dazu ein Ubuntu Betriebssystem, Aumüller und Hass benutzen einen Xeon Prozessor mit zwei 14-core Intel Xeon E5-2690 v4 CPUs getaktet auf 2.6 GHz und auch wieder das Ubuntu Betriebssystem. Während sich Aumüller und Hass in ihren Experimenten auf 64b *Integers* beschränken, benutzen Edelkamp und Weiß die Datentypen *int*, *Vector* und *Record*. Diese sind im Folgenden dargestellt.

- **int**: vorzeichenbehafteter 32b *Integer*
- **Vector** 10-dimensionales Feld aus 64b *Doubles*. Die Reihenfolge ist mithilfe der Euklidischen Norm bestimmt - für jeden Vergleich wird die Summe der Quadrate der einzelnen Komponenten berechnet und dann verglichen.
- **Record** 21-dimensionales Feld aus 32b *Integers*. Nur die erste Komponente wird verglichen.

7.1. Auswirkung verschiedener Blockgrößen auf die Laufzeit

Für eine Eingabegröße von ca. 67 Millionen sehen wir in BlockLomuto und Dual-Pivot BlockLomuto einen stetigen Abfall der Laufzeit bei Vergrößerung der Blockgröße bis $B = 2^{10} = 1024$ (Abbildung 7.1). Danach wird das Cache-Verhalten zu ineffizient um den Vorteil der erhöhten Blockgröße auszugleichen. In der Arbeit haben wir meistens die Blockgröße $B = 2^7 = 128$ benutzt, wir sehen in dem Graphen, dass der Unterschied in dieser Größenordnung marginal ist. Dies deckt sich mit den Erkenntnissen aus BlockQuicksort, dort sehen wir (Abbildung 7.2) jedoch einen Unterschied für komplexere Datentypen. Für den Datentyp Vector und Record sorgt

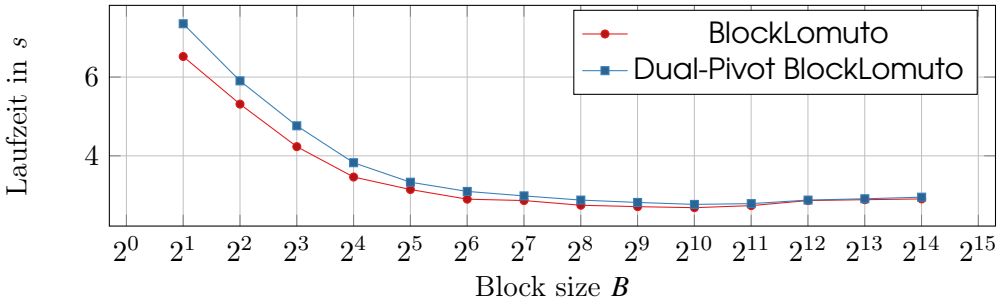


Abbildung 7.1.: Ein Laufzeitvergleich von BlockLomuto(1) und Dual-Pivot BlockLomuto(2) auf einer Eingabe der Größe $N = 2^{26}$ für verschiedene Blockgrößen $2 \leq B \leq 2^{14}$. Die Daten stammen aus den Experimenten von Aumüller und Hass [4].

bereits eine Blockgröße von $B = 2^3 = 8$ für eine gute Laufzeit, diese fängt bei weiterer Erhöhung der Blockgröße an zu steigen.

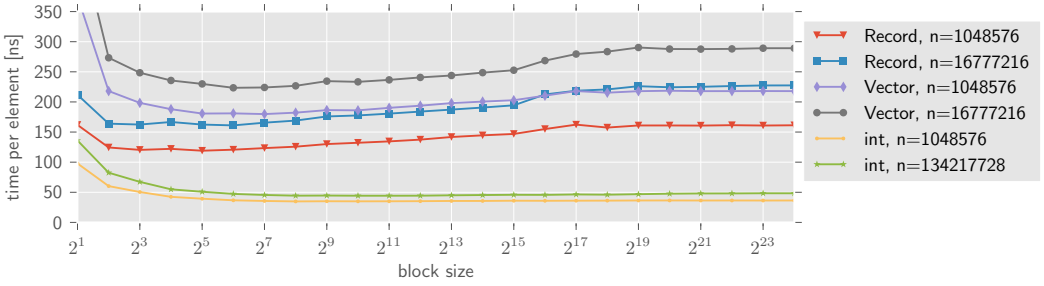


Abbildung 7.2.: Ein Laufzeitvergleich verschiedener Sortierverfahren für steigende Blockgrößen. Die Daten stammen aus den Experimenten von Edelkamp und Weiß [10].

7.2. Auswirkung verschiedener Pivot-Wahl Strategien auf die Laufzeit

Die besten Strategien
(schnell → langsam)
für BlockLomuto

1. 3 aus 5*
2. 3 aus 5
3. 2 aus 3
4. direkt

Die besten Strategien
(schnell → langsam)
für Dual-Pivot BlockLomuto

1. 1,3 aus 5*
2. 1,3 aus 5
3. 1,2 aus 3
4. 2,4 aus 5
5. direkt

Die Strategie der Pivot-Wahl hat einen großen Einfluss auf die Laufzeit (siehe Abbildung 7.3). Die Beste der dargestellten Strategien ist dabei die Wahl aus einer Menge von Medianen, welche aus 5-elementigen Samples gezogen wurden.

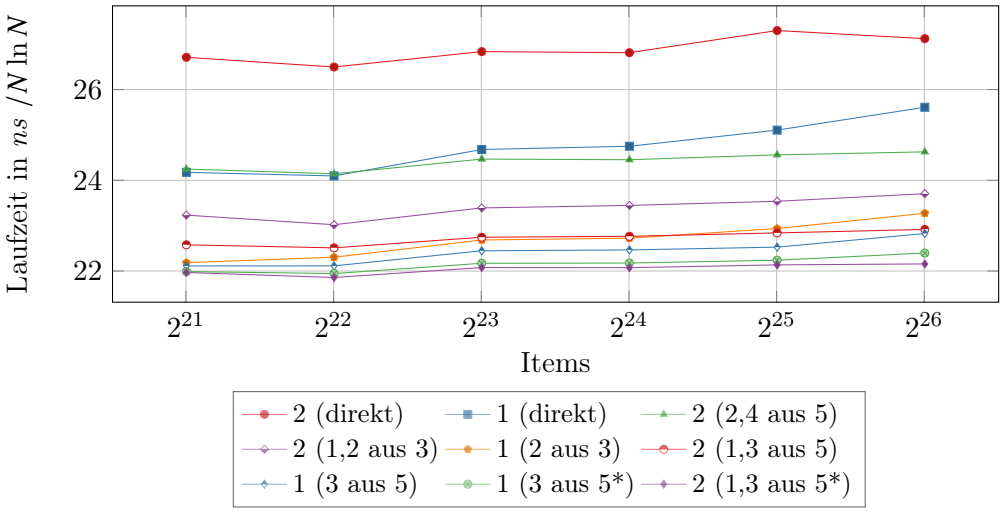


Abbildung 7.3.: Ein Laufzeitvergleich von BlockLomuto und Dual-Pivot BlockLomuto auf verschiedenen Pivot-Wahl Strategien, die Sternchen geben das Median-Of-Medians Sampling an. Die Daten stammen aus den Experimenten von Aumüller und Hass [4].

Die schlechteste Strategie ist dabei sehr eindeutig die direkte Wahl aus der Eingabe. Eine interessante Beobachtung ist die Bestätigung der vorher aufgestellten Behauptung, alle Verfahren basierend auf Multi-Pivot Quicksort arbeiten besser mit skewed Pivot-Elementen. Aumüller und Hass [4] benutzen eine adaptive Pivot-Wahl Strategie, welche sich je nach Eingabegröße anpasst. In den Experimenten von Edelkamp und Weiß [10] kann man erkennen (siehe Abbildung 7.4), dass die Benutzung von skewed Pivot-Elementen in BlockQuicksort, einem Algorithmus, welcher nur mit **einem** Pivot-Element arbeitet, zu einer Erhöhung der Laufzeit führt.

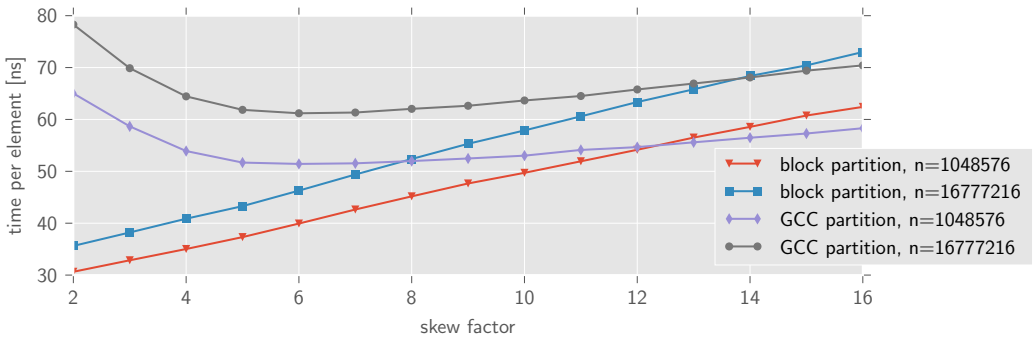


Abbildung 7.4.: Ein skew-Faktor k bedeutet, dass das $\lfloor \frac{N}{k} \rfloor$-te Element als Pivot-Element einer Eingabe der Größe N gewählt wird. Die Daten stammen aus den Experimenten von Edelkamp und Weiß [10].

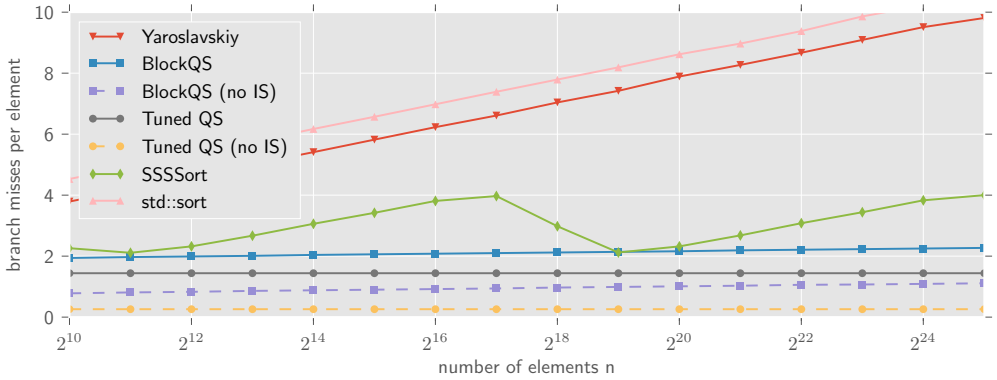


Abbildung 7.5.: Anzahl an Sprungfehlern für verschiedene Sortierverfahren. Die Daten stammen aus den Experimenten von Edelkamp und Weiß [10].

7.3. Sprungfehler in verschiedenen Sortierverfahren

In Abbildung 7.5 sehen wir, dass std::sort und Yaroslavskiys Dual-Pivot Quicksort Sprungfehler in der Dimension von $\Theta(N \log N)$ hervorrufen. Dies lässt sich direkt auf die 50/50 Sprünge der Vergleiche von Elementen mit dem Pivot-Element zurückführen. Während S^3 immer wieder Peaks in beide Richtung hat (Edelkamp und Weiß vermuten dahinter das Erreichen der Schranke zu Insertionsort mit unterschiedlich großen Sequenzen), liegen BlockQuicksort und dessen Varianten alle konstant zwischen 0.2 und 2 Sprungfehlern pro Element.

7.4. Laufzeitvergleich verschiedener Sortierverfahren

In diesem Abschnitt wollen wir uns mithilfe der Experimente von Edelkamp und Weiß sowie Aumüller und Hass die Laufzeit zufällig permutierter Eingaben für verschiedene Sortierverfahren (ohne IPS^{4}o und mit IPS^{4}o) anschauen. Anschließend werfen wir einen Blick auf die Laufzeit bei verschiedenen Eingabemustern unter Berücksichtigung von IPS^{4}o, BlockQuicksort, Dual-Pivot BlockLomuto, std::sort und BlockLomuto. Im Folgenden werden die verschiedenen Eingabemuster eingeführt.

7.4.1. Einführung der verschiedenen Eingabemuster

- Permutation: Wählt die Eingabesequenz als zufällige Permutation der Menge $\{1, ..., N\}$
- Sägezahn: Setzt $A[i] = i \ mod \ \sqrt{N}$
- ZufälligeDups: Setzt $A[i] = uniform(N) \ mod \ \sqrt{N}$
- Sortiert: Setzt $A[i] = i$
- Verkehrt: Setzt $A[i] = N - i - 1$
- Gleich: Setzt $A[i] = 1$
- AchtDups: Setzt $A[i] = i^8 + \frac{N}{2} \ mod \ N$

7.4.2. Vergleich

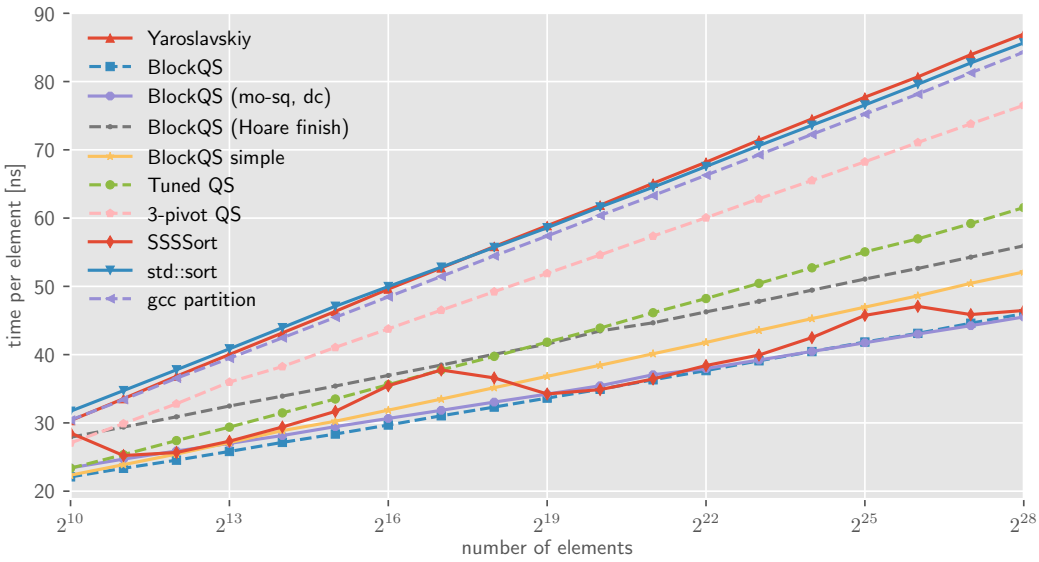


Abbildung 7.6.: Die Laufzeit verschiedener Sortierverfahren (ohne IPS⁴o) auf einer zufällig permutierten Eingabe. Die Daten stammen aus den Experimenten von Edelkamp und Weiß [10].

In dem Experiment von Edelkamp und Weiß (Abbildung 7.6) sieht man die Dominanz der Sprungfehler freien Sortierverfahren BlockQuicksort, S³ und Tuned Quicksort. Letzteres Verfahren fällt für größere Eingaben ab, ein Grund, warum es in dieser Arbeit eher als Ansatz und nicht als Verfahren dargestellt wurde. Die restlichen Verfahren, welche nicht auf Sprungfehler-optimiert wurden, haben die von uns bereits angemerkte Reihenfolge in der Laufzeit (schnell → langsam): 3-Pivot Multi-Pivot Quicksort, Yaroslavskiys Dual-Pivot Quicksort und std::sort.

Aumüller und Hass haben in ihren Experimenten auch das zuletzt vorgestellte Verfahren IPS⁴o einbezogen. In Abbildung 7.7 sieht man dessen Überlegenheit gegenüber den anderen Sprungfehler freien Sortierverfahren.

Reihenfolge der Laufzeit (schnell → langsam).

1. Faktor 1: IPS⁴o
2. Faktor 1.15: BlockLomuto
2. Faktor 1.15: Dual-Pivot BlockLomuto
3. Faktor 1.20: BlockQuicksort
4. Faktor 2.22: std::sort

Wie erwartet ist std::sort mit Abstand am langsamsten, danach folgt BlockQuicksort. BlockLomuto und Dual-Pivot BlockLomuto arbeiten auf einer zufällig permutierten Eingabe gleich schnell, am schnellsten ist IPS⁴o.

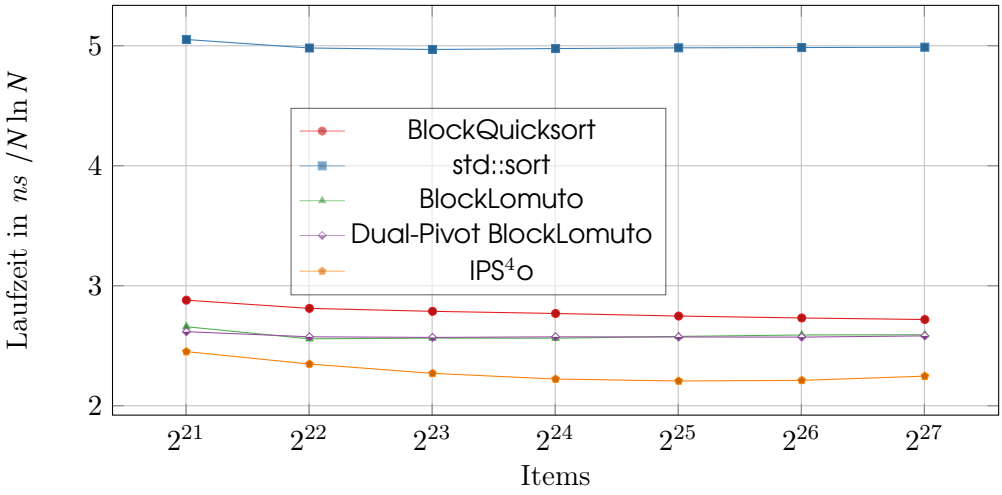


Abbildung 7.7.: Laufzeitvergleich verschiedener Sortierverfahren auf einer zufällig permutierten Eingabe. Die Daten stammen aus den Experimenten von Aumüller und Hass [4].

Für viele Duplikate (Sägezahn, ZufälligeDups, AchtDups, Gleich) ist BlockLomuto viel schlechter. Wir hatten dies in Unterabschnitt 4.2.1 bereits für Algorithmen angemerkt, welche die Partitionierung nach Lomuto benutzen. Dort wurde außerdem erwähnt, dass dies mit der Hinzunahme eines weiteren Pivot-Elementes robuster gestaltet werden könne. Dies ist hier im Praxistest zu sehen. Um vernünftige Achsen zu bekommen, wurde BlockLomuto aus den jeweiligen Darstellungen ausgeklammert.

Weiter sieht man, dass Dual-Pivot BlockLomuto zwar auf Eingaben mit Duplikaten gut läuft, jedoch nicht auf Eingaben welche (fast) sortiert oder (fast) verkehrt sortiert sind. Dafür erreicht std::sort auf diesen Eingaben die beste Laufzeit. Abschließend fällt auf, dass BlockQuicksort für eine Eingabe der Form $A[i] = 1$ sehr schlechte Laufzeiten erreicht.

Zum Vergleich befinden sich diese Experimente, erneut auf einem i7 Prozessor durchgeführt, im Abschnitt A.2 des Anhangs.

7.5. Gesamtvergleich

In der Tabelle 7.1 ist der deutliche Performance-Kontrast von IPS⁴o nach oben und std::sort nach unten zu sehen, mit BlockLomuto, Dual-Pivot BlockLomuto und BlockQuicksort in der Mitte. IPS⁴o dominiert durch wenige Instruktionen und L2 Cache-Fehler. Das Problem von std::sort hingegen sind die vielen nicht fertiggestellten Zyklen, welche vor allem durch Sprungfehler auftreten. In den vorherigen Vergleichen konnten wir sehen, dass BlockLomuto und Dual-Pivot BlockLomuto in der Regel besser als BlockQuicksort performen, obwohl dies durch unsere theoretische Analyse nicht vermutet wurde. Wir sehen in der Tabelle bei diesen Verfahren jedoch weniger unvollständige Zyklen, sowie Instruktionen insgesamt. Dies lässt sich teilweise auf die simplere Struktur zurückführen, welche von Prozessor und Compiler benutzt werden kann.

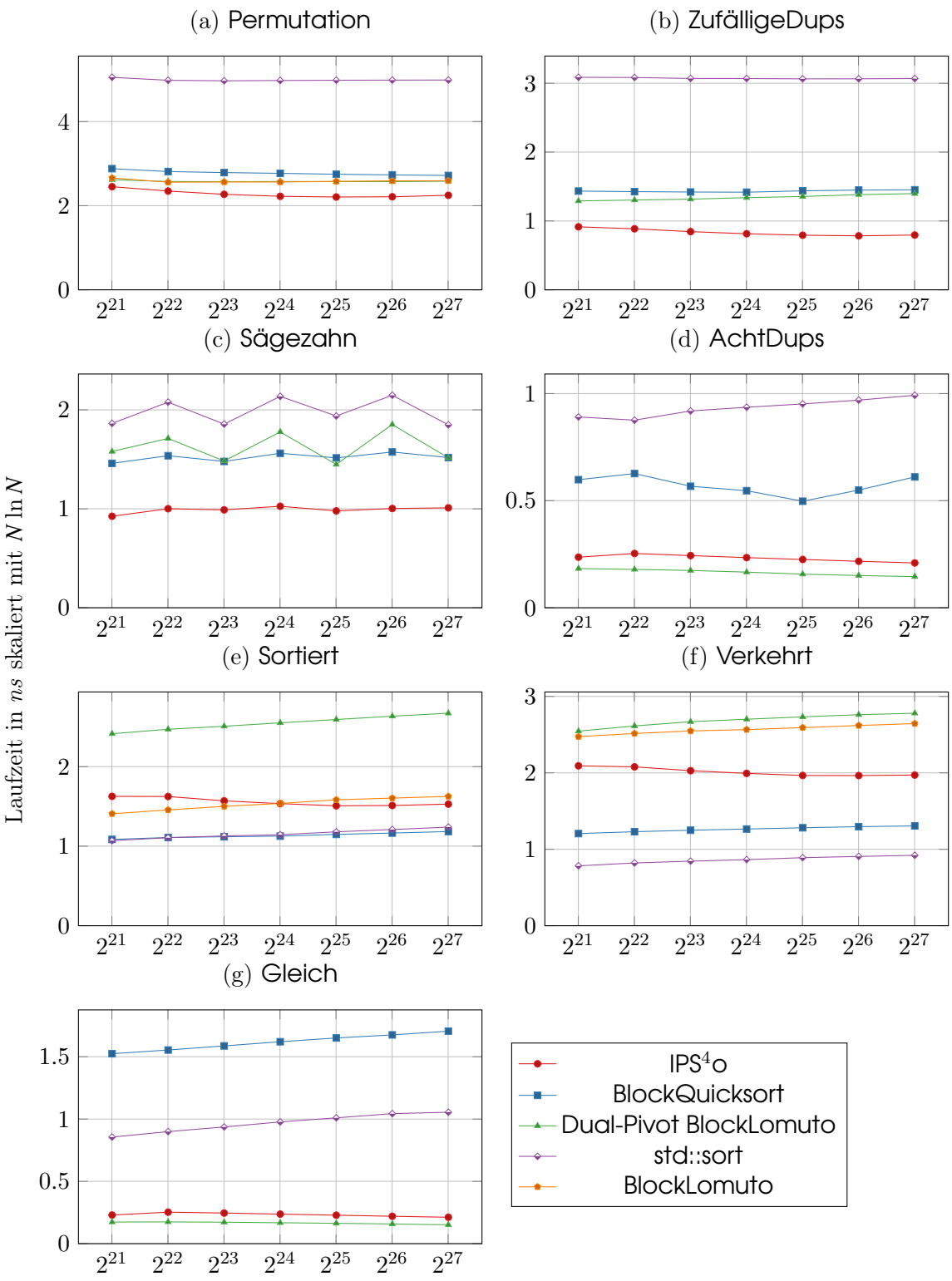


Abbildung 7.8.: Laufzeitvergleiche auf dem `Xeon` Prozessor. Die x-Achse stellt die Anzahl an Elementen dar, die y-Achse die Laufzeit in Nanosekunden, skaliert mit $N \ln N$. Die Daten stammen aus den Experimenten von Aumüller und Hass [4].

Algorithmus	L1/L2 CM	CB	INS	WOF (%, Zyklen)
IPS^{4}o	.147 / .085	1.080	12.870	2.921 (41.3%, 6.834)
BlockLomuto	.169 / 1.43	0.926	16.456	3001 (38.7%, 7.758)
Dual-Pivot BlockLomuto	.144 / .121	0.884	17.467	2.579 (32.7%, 7.878)
BlockQuicksort	.125 / .102	0.877	17.566	3.240 (38.7%, 8.354)
std::sort	.137 / .115	2.128	11.769	11.509 (74%, 15.550)

Tabelle 7.1.: Tabelle stammt aus dem BlockQuicksort-Paper von Aumüller und Hass [4]: CPU Zählspezifikationen für $N = 2^{27}$ Elemente auf einer zufällig permutierten Eingabe. **CM** = Sprungfehler, **CB** = Entscheidungssprung Instruktionen, **INS** = Instruktionen und **WOF** = Zyklen ohne fertige Instruktion. Alle Werte sind Normalisiert auf $N \ln N$.

7.6. Abschluss

Experimente haben ergeben, dass in den meisten Fällen eine Blockgröße zwischen 2^7 bis 2^{10} optimal ist. Weiter wird in jedem Fall empfohlen, das Pivot-Element nicht direkt, sondern durch ein Sample zu wählen. Für die Laufzeiten im generellen Fall sehen wir eine klare Rangfolge von (schnell → langsam): IPS^{4}o, BlockLomuto und Dual-Pivot BlockLomuto, BlockQuicksort, std::sort. Da Quicksort jedoch ein Sortierverfahren für den allgemeinen Gebrauch ist, sollten bei der Wahl des Sortierverfahrens auch die weiteren Eingabemuster berücksichtigt werden.

8. Anwendungsbeschreibung

Ziel des praktischen Teils dieser Arbeit war die Erstellung eines Programmes zur didaktischen Darstellung verschiedener Sortierprozesse. Dabei wurde in Java unter anderem eine Benutzeroberfläche entwickelt, in welcher der Nutzer einen parametrisierten Sortierprozess starten kann. Der abgeschlossene Sortierprozess lässt sich anschließend durch eine in LaTeX erstellte PDF-Ausgabe nachvollziehen.

8.1. Parametrisierung

Die Parametrisierung verläuft wie folgt:
<Eingabesequenz> <Sortierverfahren> <Variante/Pivot-Wahl> <Datentyp>
Dabei lauten die möglichen Optionen der einzelnen Parameter:

- Eingabesequenz
  - Eingabe
  - Zufall
  - verschiedene Beispiel-Eingaben
- Sortierverfahren:
  - Insertionsort
  - Heapsort
  - Quicksort
  - Lomutosort
  - BlockQuicksort
  - BlockLomuto
  - Dual-Pivot BlockLomuto
  - Samplesort
  - Super Scalar Samplesort
  - In-Place Parallel Super Scalar Samplesort
- Variante / Pivot-Wahl
  - Standard
- Datentyp
  - Integer
  - String

8.2. Ausgabe

Sobald der Sortierprozess abgeschlossen ist, kann man sich die Zwischenschritte erst vom Programm selbst skizzieren lassen und sich dann anschließend mittels *Erstelle PDF* als PDF ausgeben lassen. Alternativ kann man sich direkt den LaTeX Quellcode der Sortierschritte mittels *Erstelle LaTeX* und/oder das LaTeX-Template mittels *Erstelle Template* anzeigen lassen.

Das erstellte Programm wurde im theoretischen Teil dieser Arbeit genutzt, um die vorgestellten Sortierverfahren beispielhaft darstellen zu können. Ein weiteres Anwendungsgebiet befindet sich im Lehrbereich. Das Programm kann dort benutzt werden, um die Abläufe verschiedener Verfahren sowie Optimierungen und Varianten dieser besser zu erklären. Es sei hier darauf hingewiesen, dass die Implementation unter dem zentralen Aspekt stattgefunden hat, alle Schritte so transparent wie möglich zu gestalten. Dabei wurde nicht immer die effizienteste Implementation gewählt.

8.3. Softwarearchitektur

Das in dieser Arbeit entwickelte Programm bedient sich in abgeänderter Form dem *Model-View-Controller* (MVC) Entwurfsmuster. Dieses ist in Abbildung 8.1 dargestellt und durch Müller-Olm [29] wie folgt definiert:

Definition 8.1. (Das *MVC* Entwurfsmuster)
Durch die Kombination von Entwurfsmustern erhält man die MVC-Struktur. Diese bildet sich aus den drei Komponenten Model, View und Controller. Die jeweiligen Aufgaben sind dabei:

Model: Datenhaltung sowie zugehörige Methoden

View: Anzeige von Model-Zustand und Bedienelementen

Controller: Übersetzung von Ereignisfolgen auf die Bedienelemente der View

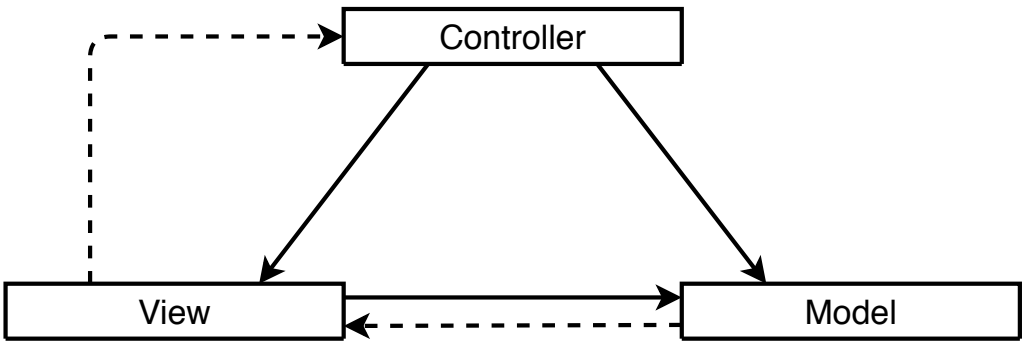


Abbildung 8.1.: Die Abhängigkeiten im Enwurfsmuster: MVC

Das Model-View-Controller Konzept wird laut Müller-Olm in erster Linie benutzt um „Erweiterbarkeit und Änderbarkeit eigener Software zu erreichen“ [29].

In dieser Arbeit ist die Benutzung von diesem Ansatz sinnvoll, um das Programm ohne große Umstände auf die unterschiedlichsten Anforderungen im Anwendungsgebiet des Sortierens flexibel erweitern zu können. Dabei kann der entstehende Trade-off zwischen Flexibilität und Overhead [29] in Kauf genommen werden, da das Programm unter dem Fokus der didaktischen Darstellung und nicht der Effizienz steht. Die Konzepte sind in der Implementation wie folgt auf die Klassen abgebildet (vergleiche Abbildung 8.2):

- Model → *SortingState*
- View → *View*
- Controller → *Controller*

Das Model bedient sich dabei den Black Boxen *Sorter* und *SortingStep*. Vergleicht man Abbildung 8.2 und Abbildung 8.1 erkennt man mehrere Unterschiede, wie zum Beispiel in den Relationen der einzelnen Klassen.

Ein Sortierschritt speichert dabei den aktuellen Zustand aller Variablen einschließlich der zu sortierenden Sequenz. Die Variablen werde dabei klassifiziert in *arrayListMap* (speichert die Sequenzen des generischen Datentyps *I*), *indexMap* (speichert die Zeiger) und *indexArrayMap* (speichert *Integer* Sequenzen, wie zum Beispiel Puffer).

8.4. Erweiterbarkeit

Zusätzlich zu der Ergänzung des Programmes mit weiteren Sortierverfahren sind bereits weitere Erweiterungen möglich. Damit diese und weitere Verbesserungen im Programm möglichst einfach hinzugefügt werden können, wurde sich im Vorfeld Gedanken über die Struktur gemacht. Diese Gedanken werden im Folgenden vorgestellt.

Das generisches Interface *Sorter* sorgt dafür, dass neue Sortierverfahren die nötigen Methoden implementieren müssen, um mit dem Programm kompatibel zu sein. Weiter werden die benötigten Hilfsmittel und Methoden, um aus den aus dem Sortierprozess entstehenden Sortierschritten einen LaTeX-Code zu erstellen, durch die Abstrakte Klasse *TexStringGenerator* bereitgestellt (siehe Abbildung 8.3).

Das behandeln der Sortierverfahren und Sortierschritte als Black Box sorgt dafür, dass einzelne Sortierverfahren einfach austauschbar sind und die gespeicherten Attribute in einem Sortierschritt flexibel änderbar sind. Zudem ermöglichen generische Typen das Hinzufügen weiterer Datentypen ohne die Kompatibilität, vor allem der Sortierverfahren, mit dem Programm zu verlieren (vgl. Abbildung 8.2).

Außerdem kann auch die View weiter ausgebaut werden. So können in der initialen Parameterwahl mehrere Datentypen, sowie Varianten des Sortierverfahrens und verschiedene Pivot-Wahlen hinzugefügt werden. Ferner könnte die Ausgabe einerseits erweitert werden, um verschiedene Schritte im Sortierverfahren eventuell detaillierter

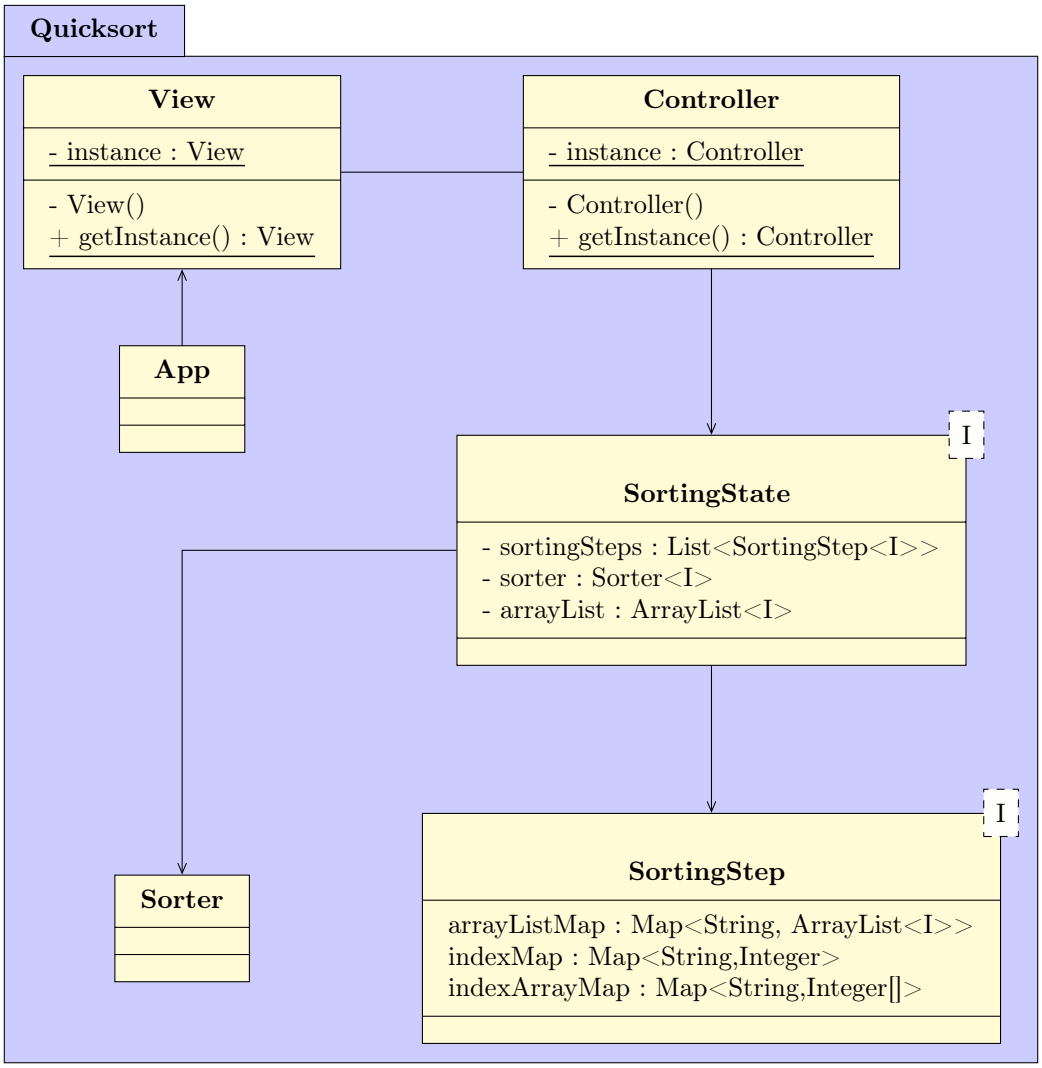


Abbildung 8.2.: UML-Klassendiagramm der Implementierten MVC-Struktur

darzustellen oder zu streichen (*Zwischenschritte*, *Pivot-Berechnung*, *Basisfall*, *Worst-Case Sicherung*). Die benötigte Infrastruktur dafür ist bereits in der Implementation enthalten, ein Beispiel für die Unabhängigkeit der einzelnen Komponenten innerhalb des MVCs. Abschließend kann die Option hinzugefügt werden, die Ausgabe als Beamer-PDF zu erhalten (*Erstelle Beamer*).

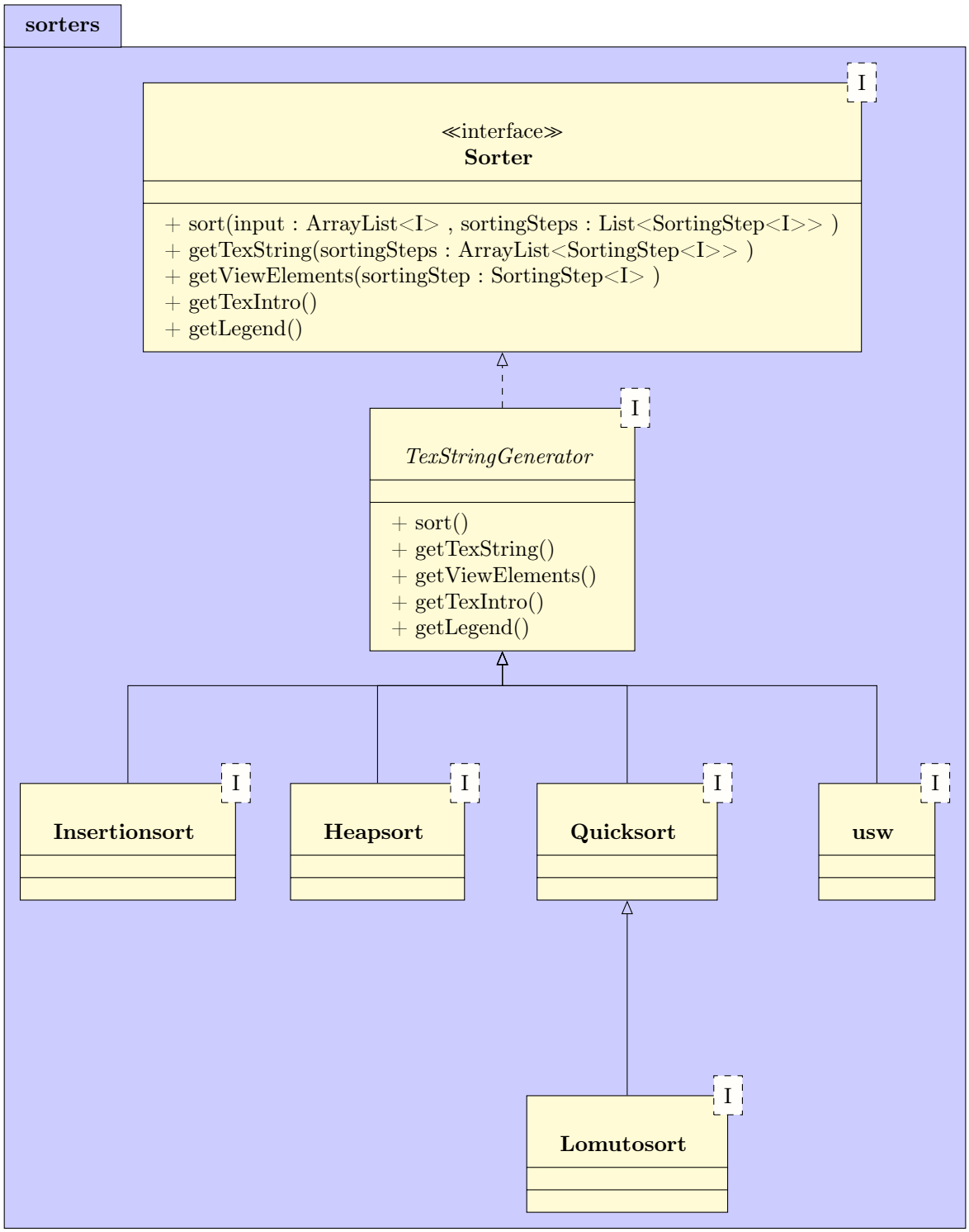


Abbildung 8.3.: UML-Klassendiagramm der Implementierten Struktur der Sortierverfahren

9. Ausblick

9.1. Sprungfehlervermeidung für Verfahren außerhalb des Sortierens

Die in Kapitel 2 aufgearbeiteten Ansätze zur Vermeidung von falschen Sprungvorhersagen können auch in anderen Gebieten außerhalb des Sortierens benutzt werden. Ein Beispiel dafür liefert die Abhandlung „Architektonische Unterstützung für probabilistische Zweige“ von Adileh et. al. [1]. Dort wird ein Verfahren gegen Fehlvorhersagen behandelt, welche durch Zufalls-Sprünge auftreten. Einer der dort auftretenden Algorithmen sieht wie folgt aus:

```
1: function MUTATE(arrray bits)
2:     for (int i ← 0; i <bits.length(); i++) do
3:         if RANDOM_NUM < MUTATION_RATE then
4:             if bits[i] = 1 then
5:                 bits[i] ← 0
6:             else
7:                 bits[i] ← 1
```

Es gibt keinen Grund, warum wir die Erkenntnisse, welche wir aus unserer Sprungfehler Optimierung ziehen, nicht auch in anderen Gebieten benutzen sollten. So könnten wir den vorgestellten Beispiel-Algorithmus unabhängig von dem von Adileh et. al. vorgestellten Verfahren wie folgt optimieren:

```
1: function MUTATE(array bits)
2:     for (int i ← 0; i <bits.length(); i++) do
3:         bits[i] ← (bits[i] + (RANDOM_NUM < MUTATION_RATE) )% 2
```

Dies ist nur ein Beispiel zur Illustration, es wird angenommen, dass bits[i] $\in \{0,1\}$.

9.2. Sprungfehler in kommenden Optimierungen der Sortierverfahren

Die Berücksichtigung von Sprungfehlern in Sortierverfahren ist ein (in der Zeitspanne der Sortierverfahren) relativ neues Gebiet. Da die Paper und Arbeiten zu diesen Themen alle sehr aktuell sind, stehen die Chancen gut, dass in der nächsten Zeit weitere Verbesserungen in diesem Bereich erzielt werden können. Allgemein liegt großes Verbesserungspotential in der Laufzeit der Sortierverfahren, sollten die modernen Prozessoraspekte mehr berücksichtigt werden.

10. Fazit

Ziel dieser Arbeit war es, aus dem Schatten teilweise Jahrzehnte benutzter Sortierverfahren herauszuschreiten und Algorithmen vorzustellen, welche modernen Hardwareeigenschaften gerecht werden. Dazu sollten Verfahren vorgestellt werden, welche die Implementierungen in heutigen Standard-Bibliotheken ersetzen könnten. Zusätzlich sollte dabei der Fokus auf der Vermeidung von Sprungfehlern liegen, um somit in der Praxis eine deutliche Laufzeitverbesserung zu ermöglichen. Im Verlaufe der Arbeit wurden insgesamt sechs vielversprechende Algorithmen vorgestellt, welche alle auf Quicksort basieren. Zu diesen gehören BlockQuicksort, BlockLomuto, Dual-Pivot BlockLomuto, Samplesort, Super Scalar Samplesort und In-Place Parallel Super Scalar Samplesort. Während die Sortierverfahren Samplesort und S^3 beide an der Erfüllung der In-Place Eigenschaft scheiterten, dienten sie uns als Grundlage für das IPS^{4}o-Verfahren, welches im späteren Verlauf der Arbeit vorgestelltem Praxis-Vergleich sehr erfolgreich war. Im selben Vergleich mussten wir feststellen, dass BlockLomuto nicht robust genug gegenüber Duplikaten war, wodurch das Verfahren als Kandidat für eine Bibliothekts-Implementierung ausschied. Auch hier wurde jedoch, dieses Mal mit Dual-Pivot BlockLomuto, ein Nachfolger vorgestellt, welcher die vorherige Schwäche ausbessern konnte. Am Ende konnten wir mit BlockQuicksort, Dual-Pivot BlockLomuto und IPS^{4}o insgesamt drei Verfahren vorstellen, welche die benötigten Eigenschaften erfüllen können, um einen Platz in den Standard-Bibliotheken zu erhalten. So haben alle Verfahren eine asymptotische Laufzeit in $O(N \log N)$. Weiter wurde bei allen Verfahren Speicherplatz benutzt, welcher sublinear in der Eingabegröße liegt und zuletzt konnten wir durch eine optimierte Wahl von Puffer, Bucket und Blockgrößen ein gutes Cache-Verhalten erzielen.

Außerhalb dieser drei Eigenschaften ist die Stärke dieser drei vergleichsbasierten Sortierverfahren Instruktionen auszuführen, ohne dabei unnötige und teure Entscheidungssprünge zu benutzen. Dabei werden die im Sortierprozess hervorgerufenen Sprungfehler stark reduziert. Es existieren zwar sogenannte schlanke Sortierverfahren, deren Anzahl Sprungfehler in $O(1)$ liegt, diese nehmen aber zu große Einschnitte in die Laufzeit in Kauf. Dadurch wurde sich in dieser Arbeit damit begnügt die Sprungfehler nur hinreichend klein zu halten. Zwar können wir nicht immer bestimmen, welche Technik moderne Prozessoren bei der Sprungvorhersage benutzen, so konnten wir jedoch trotzdem feststellen, dass die einfachen in dieser Arbeit vorgestellten Techniken zur Sprungvorhersage oft hinreichend sind, um die Effizienz der Sortierverfahren hinsichtlich Sprungfehler einschätzen zu können. Dies sieht man zum Beispiel in Kapitel 7, wo mehrere Sortierverfahren mit Blick auf die Sprungfehler verglichen wurden. Die Kosten für Sprungfehler pro Element blieben bei den Sprungfehler-optimierten Verfahren konstant, während sie bei Quicksort, Dual-Pivot Quicksort und Multi-Pivot Quicksort logarithmisch in der Eingabegröße anstiegen.

Weiter haben wir in der Arbeit ein gefestigtes Verständnis für den Quicksort-Algorithmus erarbeitet sowie viele Ideen zu möglichen Optimierungen, Erweiterungen und Varianten vorgestellt. Dabei haben wir mit der Hinzunahme von Insertionsort als Basis-Fall durchgehende Verbesserungen erzielen können, bei den Varianten von BlockQuicksort ist zusätzlich die Beschränkung für den schlechtesten Fall durch Heapsort nennenswert. Auch wenn wir das Thema der Pivot-Wahl in dieser Arbeit nicht im Detail besprechen konnten ist deutlich geworden, dass eine feste Pivot-Wahl sowohl zu deutlich schlechteren Ergebnissen führt und zusätzlich eine Sicherheitslücke darstellen kann. Im Gegensatz dazu konnten wir mit verschiedenen Pivot-Wahl Strategien, welche auf der Approximation des Medians basieren, gute Ergebnisse erzielen. Außer Frage steht, dass die in dieser Arbeit vorgestellten Ideen, Puffer zu benutzen, die Eingabe in Buckets zu klassifizieren oder sie als Blöcke zu gruppieren ganz sicher Optimierungen sind, welche in zukünftig entwickelten Sortierverfahren eine große Rolle spielen können.

Ferner konnten wir feststellen, dass sowohl die Wahl des besten Sortierverfahrens als auch die optimale Wahl der Parameter, von mehreren Eigenschaften wie zum Beispiel der verfügbaren Hardware oder der Struktur der Eingabesequenz abhängen kann. Dadurch kann es sinnvoll sein, benutzte Sortierverfahren, sowie dessen Parameter, dynamisch auszuwählen. Sortierverfahren wie IPS^{4}o oder die Multi-Thread Variante von BlockQuicksort sind bei Multi-Core-Prozessoren zu bevorzugen, bei Datentypen wie Vector oder Record führt eine kleinere Blockgröße zu besseren Ergebnissen. Letzten Enden sollte man jedoch nicht den Vorzug von Quicksort vergessen, dass es ein Verfahren für den allgemeinen Gebrauch ist. Dies ist auf die Verfahren BlockLomuto, Dual-Pivot BlockLomuto und IPS^{4}o übertragbar, weshalb diese Algorithmen auch ohne dynamische Auswahl unter den verschiedensten Umständen gute Laufzeiten erzielen.

Die Sprungfehleranalyse in dieser Arbeit ist ein Beispiel dafür, dass es nicht immer nur auf theoretische Analyse ankommt. Voraussetzung dafür ist natürlich, dass nicht alle Parameter exakt bekannt sind, welche die Laufzeit beeinflussen könnten. Dies sieht man auch außerhalb dieser Arbeit. So wurden laut Wild und Nebel auch in Yaroslavskiys Dual-Pivot Quicksort erst „umfangreiche empirische Leistungstests“ durchgeführt, bevor dieser als neuer Standard in Oracles Java 7 Bibliothek festgelegt werden konnte [45]. Mit immer besser werdenden Prozessoroptimierungen, welche in der theoretischen Analyse zu berücksichtigen sind, wird die Laufzeitanalyse durch das einfache Zählen der Instruktionen immer weniger aussagekräftig. Dies ist fest damit verbunden, dass bei der Entwicklung neuer Sortierverfahren immer mehr auf die Eigenschaften moderner Prozessoren, wie zum Beispiel dem Instruction-level parallelism, sowie der Sprungvorhersage eingegangen werden sollte. Dazu passt folgendes Zitat welches Kushagra et. al. 2014 in ihrer Abhandlung geschrieben haben: „Die Laufzeiten von Quicksort-Algorithmen werden am besten abgeschätzt, indem eine Kostenbemessung abhängig der Cache-Fehler der CPU benutzt wird“ [21].

Durch diese Arbeit wurde gezeigt, dass die theoretische Laufzeit-Analyse nicht nur auf Cache-Fehler erweitert werden sollte, sondern genauso auf Sprungfehler und eventuellen weiteren Eigenschaften, welche es zu finden gilt.

Literaturverzeichnis

[1] A. Adileh, D. J. Lilja, and L. Eeckhout. Architectural support for probabilistic branches. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 108–120, 2018.

[2] Martin Aumüller and Martin Dietzfelbinger. Optimal partitioning for dual pivot quicksort. In Fedor V. Fomin, Rūsiņš Freivalds, Marta Kwiatkowska, and David Peleg, editors, *Automata, Languages, and Programming*, pages 33–44, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[3] Martin Aumüller, Martin Dietzfelbinger, and Pascal Klaue. How good is multi-pivot quicksort? *ACM Trans. Algorithms*, 13(1), October 2016.

[4] Martin Aumüller and Nikolaj Hass. Simple and fast blockquicksort using lomuto's partitioning scheme, 2018.

[5] Michael Axtmann, Sascha Witt, Daniel Ferizovic, and Peter Sanders. In-Place Parallel Super Scalar Samplesort (IPSSSSo). In *25th Annual European Symposium on Algorithms (ESA 2017)*, volume 87 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 9:1–9:14. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2017.

[6] Gerth Stølting Brodal and Gabriel Moruz. Tradeoffs between branch mispredictions and comparisons for sorting algorithms. In Frank Dehne, Alejandro López-Ortiz, and Jörg-Rüdiger Sack, editors, *Algorithms and Data Structures*, pages 385–395, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[7] R. Chaudhuri and A. C. Dempster. A note on slowing quicksort. *SIGCSE Bull.*, 25(2):57–58, June 1993.

[8] Curtis R. Cook and Do Jin Kim. Best sorting algorithm for nearly sorted lists. *Commun. ACM*, 23(11):620–624, November 1980.

[9] T. H. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Introduction to Algorithms, Second Edition.* MIT Press, 2001.

[10] Stefan Edelkamp and Armin Weiß. Blockquicksort: Avoiding branch mispredictions in quicksort. *ACM J. Exp. Algorithmics*, 24(1), January 2019.

[11] A. Farcy, O. Temam, R. Espasa, and T. Juan. Dataflow analysis of branch mispredictions and its application to early resolution of branch outcomes. In *Proceedings. 31st Annual ACM/IEEE International Symposium on Microarchitecture*, pages 59–68, 1998.

[12] W. D. Frazer and A. C. McKellar. Samplesort: A sampling approach to minimal storage tree sorting. *J. ACM*, 17(3):496–507, July 1970.

[13] PASCAL HENNEQUIN. *Analyse en moyenne d'algorithmes, tri rapide et arbres de recherche*. PhD thesis, 1991. Thèse de doctorat dirigée par Steyaert, Jean-Marc Sciences appliquées Palaiseau, Ecole polytechnique 1991.

[14] John L. Hennessy and David A. Patterson. *Computer Architecture, Fifth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5th edition, 2011.

[15] Prof. Dr. Paula Herber. Vorlesung Informatik IV - Rechnerstrukturen [Kapitel 06]: Pipelining, Summer 2019.

[16] Intel. Intel 64 and ia-32 architecture optimization reference manual, 2016. Retrieved December 24, 2018 from http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf Order Number: 248966-032.

[17] Kanela Kaligosi and Peter Sanders. How branch mispredictions affect quicksort. In Yossi Azar and Thomas Erlebach, editors, *Algorithms – ESA 2006*, pages 780–791, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[18] Christof Kaser. Make quicksort fast by using threads and avoiding branch misprediction. `https://easylang.online/blog/qsort_c.html`. Accessed: 2020-08-12.

[19] Jyrki Katajainen. *Sorting programs executing fewer branches*. CPH STL Report. Department of Computer Science, University of Copenhagen, 2014.

[20] Donald E. Knuth. *The Art of Computer Programming, Volume 3: (2nd Ed.) Sorting and Searching*. Addison Wesley Longman Publishing Co., Inc., USA, 1998.

[21] Shrinu Kushagra, Alejandro López-Ortiz, J. Munro, and Aurick Qiao. Multi-pivot quicksort: Theory and experiments. *Proceedings of the Workshop on Algorithm Engineering and Experiments*, pages 47–60, 05 2014.

[22] Jeffrey S. Leon. Computer algorithms i : After cs_401 / mcs_401, Summer 2007.

[23] S. A. Mahlke, R. E. Hank, R. A. Bringmann, J. C. Gyllenhaal, D. M. Gallagher, and W. . W. Hwu. Characterizing the impact of predicated execution on branch prediction. In *Proceedings of MICRO-27. The 27th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 217–227, 1994.

[24] Conrado Martínez, Markus E. Nebel, and Sebastian Wild. Analysis of branch misses in quicksort. *2015 Proceedings of the Twelfth Workshop on Analytic Algorithmics and Combinatorics (ANALCO)*, Dec 2014.

[25] Colin McDiarmid and Ryan Hayward. Strong concentration for quicksort. In *Proceedings of the Third Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '92, page 414–421, USA, 1992. Society for Industrial and Applied Mathematics.

[26] Nikolaj Borg-Hass Michael Axtmann, Armin Weiß. Github: Blockquicksort.

[27] Dalia Motzkin. Pracniques: Meansort. *Commun. ACM*, 26(4):250–251, April 1983.

[28] David Musser. Introspective sorting and selection algorithms. *Software Practice and Experience*, 27:983–993, 1997.

[29] Prof. Dr. Müller-Olm. Vorlesung Software Engineering [Kapitel 06 Part 2]: Entwurfsmuster, Winter 2018-2019.

[30] David Patterson. Sequential laundry - electrical and computer engineering.

[31] David A. Patterson and John L. Hennessy. *Computer Organization and Design, Fifth Edition: The Hardware/Software Interface.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5th edition, 2013.

[32] Prof. Erik Demaine Prof. Charles Leiserson. Mit lecture 4: Quicksort, randomized algorithms, Fall 2005.

[33] Peter Sanders and Sebastian Winkel. Super scalar sample sort. In Susanne Albers and Tomasz Radzik, editors, *Algorithms – ESA 2004*, pages 784–796, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[34] R. S. Scowen. Algorithm 271: Quickersort. *Commun. ACM*, 8(11):669–670, November 1965.

[35] R. Sedgewick. Phd thesis: Quicksort, 1975.

[36] Robert Sedgewick. The analysis of quicksort programs. *Acta Inf.*, 7(4):327–355, December 1977.

[37] Robert Sedgewick. Implementing quicksort programs. *Commun. ACM*, 21(10):847–857, October 1978.

[38] Ottmann T. and Widmayer P. *Sortieren. In: Algorithmen und Datenstrukturen.* Springer Vieweg, Berlin, Heidelberg., 2017.

[39] Laszlo Toth, R. High, D. E. Knuth, R. L. Graham, and O. Patashnik. E3432. *The American Mathematical Monthly*, 99(7):684–685, 1992.

[40] Prof. Dr. Jan Vahrenhold. Vorlesung Informatik II - Datenstrukturen und Algorithmen [Kapitel 04]: Sortierverfahren, Summer 2018.

[41] Prof. Dr. Jan Vahrenhold. Vorlesung Informatik II - Datenstrukturen und Algorithmen [Kapitel 05]: Divide-And-Conquer, Summer 2018.

[42] Prof. Dr. Jan Vahrenhold. Vorlesung Informatik II - Datenstrukturen und Algorithmen [Kapitel 08]: Vorrangswarteschlangen, Summer 2018.

[43] Roger L. Wainwright. A class of sorting algorithms based on quicksort. *Commun. ACM*, 28(4):396–402, April 1985.

[44] Lutz Wegner. Quicksort for equal keys. *IEEE Trans. Computers*, 34:362–367, 04 1985.

[45] Sebastian Wild and Markus E. Nebel. Average case analysis of java 7's dual pivot quicksort. In Leah Epstein and Paolo Ferragina, editors, *Algorithms – ESA 2012*, pages 825–836, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[46] John W. J. Williams. Algorithm 232: Heapsort. *Commun. ACM*, 7(6):347–348, June 1964.

[47] Sascha Witt. Github: In-place parallel super scalar samplesort (ips4o).

A. Anhang

A.1. Alternative zur swap Instruktion

Algorithmus 1.1 Bewege die Elemente mithilfe von Rotationen um eine Partitionierung zu erhalten

Bewege die Elemente durch Zyklische Rotation um eine Partitionierung zu erhalten

1: **function** EXCHANGE$_k(A[1,...,n])$
2: $i \leftarrow k+1$; $j \leftarrow n$;
3: $m \leftarrow \lceil \frac{k+1}{2} \rceil$;
4: $b_1, ..., b_{m-1} \leftarrow i$;
5: $b_m, ..., b_{k-1} \leftarrow j$;
6: $p, q \leftarrow -1$; ▷ p und q halten die Gruppen Indices der Elemente indiziert durch i und j.
7: **while** $i < j$ **do**
8: **while** $A[i]$ gehört zur Gruppe A_p mit $p < m$ **do**
9: **if** p < m - 1 **then**
10: $rotate(i, b_{m-1}, ..., b_{p+1})$;
11: $b_{p+1}++, ..., b_{m-1}++$;
12: $i++$;
13: **while** $A[j]$ gehört zu Gruppe A_q mit $q \geq m$ **do**
14: **if** $q \geq m+1$ **then**
15: $rotate(j, b_m, ..., b_{q-1})$;
16: $b_{q-1}--, ..., b_m--$
17: $j--$;
18: **if** $i < j$ **then**
19: $rotate(i, b_{m-1}, ..., b_{q+1}, j, b_m, ..., b_{p-1})$;
20: $i++; b_{q+1}++, ..., b_{m-1}++$;
21: $j--; b_m--, ..., b_{p-1}--$;

A.2. Alternativer Laufzeitvergleich

Diese Abbildung soll die Ergebnisse der Experimente aus Kapitel 7 mithilfe eines weiteren Prozessors unterstützen. Dafür wird hier der `Xeon` Prozessor durch einen 4-Kernigen Intel Core `i7`-4790 Prozessor, getaktet auf 3.6 GHz ersetzt.

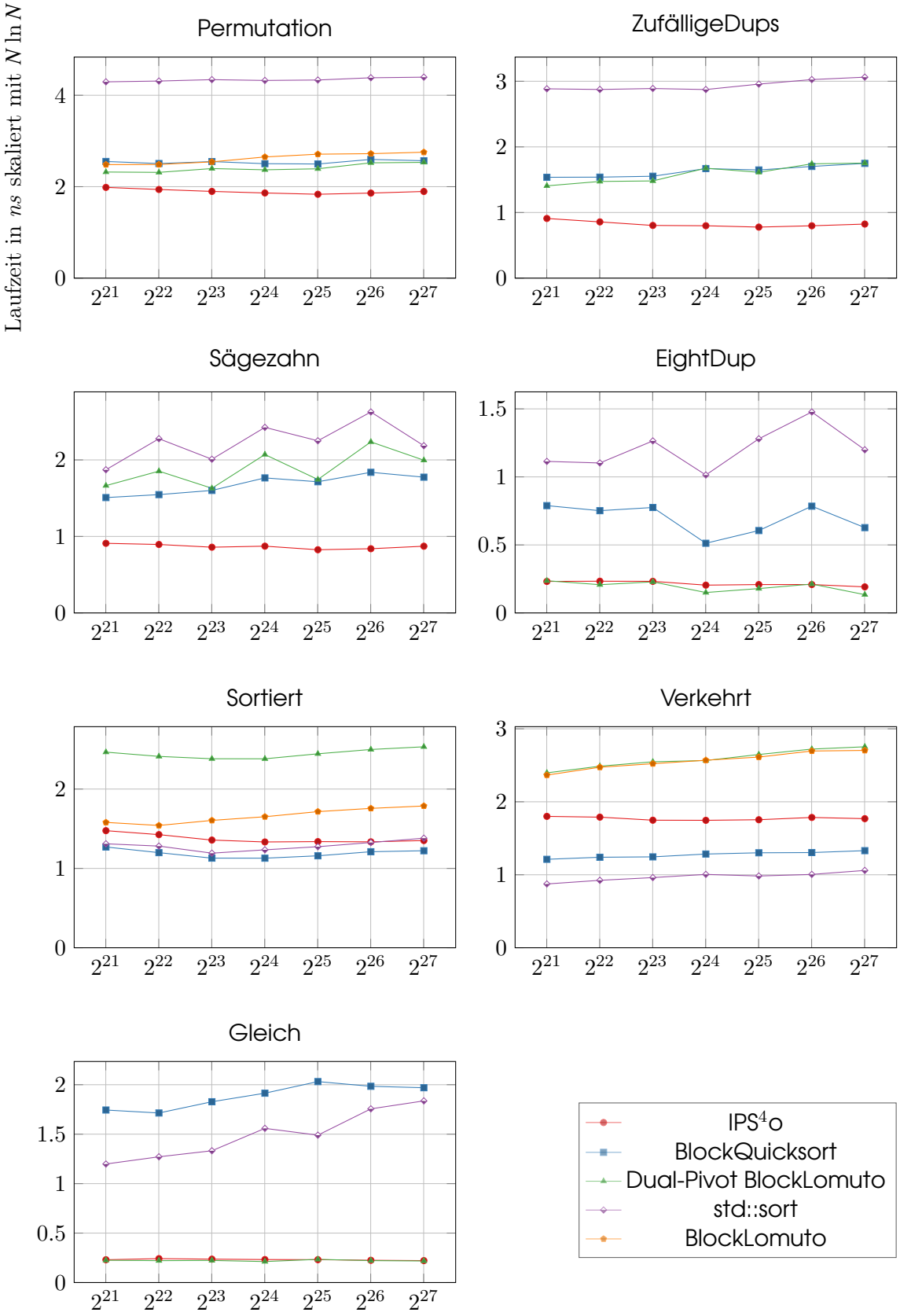


Abbildung A.1.: Laufzeitvergleiche auf dem `i7` Prozessor. Die x-Achse stellt die Anzahl an Elementen dar, die y-Achse die Laufzeit in Nanosekunden, skaliert mit $N \ln N$. Die Daten stammen aus den Experimenten von Aumüller und Hass [4].

Eidesstattliche Erklärung

Hiermit versichere ich, dass die vorliegende Arbeit über *„Optimierungsansätze für das Quicksort-Verfahren “* selbstständig verfasst worden ist, dass keine anderen Quellen und Hilfsmittel als die angegebenen benutzt worden sind und dass die Stellen der Arbeit, die anderen Werken – auch elektronischen Medien – dem Wortlaut oder Sinn nach entnommen wurden, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht worden sind.

Jonas Stübbe, Münster, 31. Oktober 2020

Ich erkläre mich mit einem Abgleich der Arbeit mit anderen Texten zwecks Auffindung von Übereinstimmungen sowie mit einer zu diesem Zweck vorzunehmenden Speicherung der Arbeit in eine Datenbank einverstanden.

Jonas Stübbe, Münster, 31. Oktober 2020