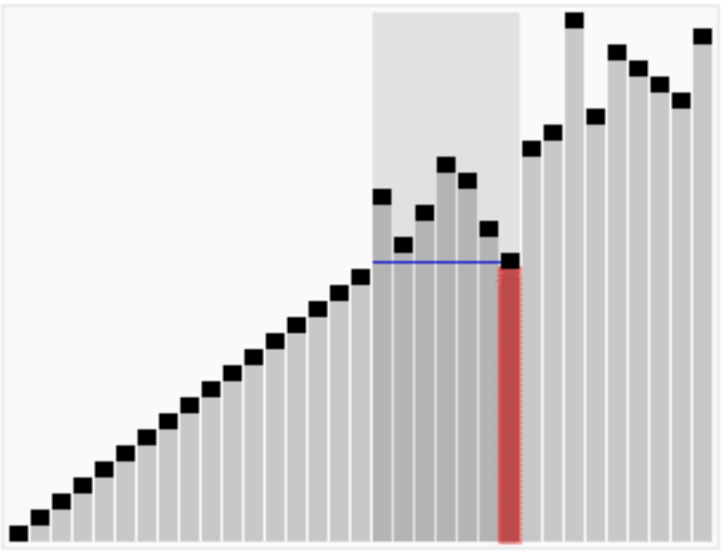
Hector Seminar

Sortieralgorithmen in Lean



1

Abschlussbericht der Kooperationsphase 2025/26

Durchgeführt in der Lean Focused Research Organization
Betreut durch Julia Markus Himmel

Johan Henkel und Elias Jungmann, Kurs KA 20

Inhaltsverzeichnis

1 Einleitung **2**

1.1 Eigenschaften von Lean . . . 2

1.1.1 Strikt . . . 2

1.1.2 Pur . . . 2

1.1.3 Funktional . . . 3

1.1.4 Dependent Types . . . 5

1.2 Theoretische Betrachtung des Sortierens . . . 6

1.2.1 Theoretische Grundlagen . . . 6

1.2.2 Verschiedene Algorithmen . . . 7

1.2.3 Hybridität . . . 10

1.3 Aufgabenstellung . . . 11

2 Implementierungen der Algorithmen **12**

2.1 Entscheidungsfindungsphase . . . 12

2.1.1 Mergesort . . . 12

2.1.2 Heapsort . . . 17

2.1.3 Quicksort . . . 17

2.2 Pull-Request: Quicksort mit *Dutch National Flag*-Algorithmus . . . 18

2.2.1 Insertionsort . . . 18

2.2.2 Auswählen des Pivot-Elements . . . 19

2.2.3 Partitionierung . . . 19

2.2.4 Gesamtablauf . . . 21

2.2.5 Wrapper . . . 22

3 Ergebnisse **23**

3.1 Benchmarken der Algorithmen . . . 23

3.2 Vergleich der Sortieralgorithmen . . . 25

3.3 Einfluss des Pull-Requests auf Leans „Radar“-Benchmark . . . 29

4 Diskussion **31**

4.1 Bewertung der Ergebnisse . . . 31

4.2 Bedeutung der Ergebnisse im Hinblick auf Lean . . . 33

4.3 Verbesserungsmöglichkeiten . . . 33

5 Appendix **37**

5.1 Listings . . . 37

Sortieralgorithmen in Lean © 2026 by Johan Henkel & Elias Jungmann is licensed under CC BY-SA 4.0. To view a copy of this license, visit `https://creativecommons.org/licenses/by-sa/4.0/`

Abstract

Lean is a strict, pure, functional programming language with dependent types. Using the *Curry-Howard Isomorphism*, it can also be used to formalise and verify mathematical proofs. The Lean standard library includes various sorting algorithms such as Merge Sort and Quick Sort. The goal of this work was to identify a way to improve the performance of sorting in Lean.

A comparative analysis of Merge Sort, Heap Sort, and Quick Sort indicated that a reimplementation of Quick Sort holds the most potential for Lean. The main drawback of its current version is that it takes $O(n^2)$ time to sort an array containing only one replicated value. To counter this, the new implementation uses Dijkstra's *Dutch National Flag* algorithm for three-way partitioning. Furthermore, an Insertion Sort cutoff was added to reduce recursion depth, turning the implementation into a hybrid sorting algorithm. While it now sorts replicated values efficiently, benchmarking shows longer runtime on randomly shuffled arrays. When a comparator function is provided instead of the internally used `Ord` typeclass, performance suffers even more. Nonetheless, this currently is the preferred way of calling `Array.qsort`, as its internal implementation relies on it. Consequently, further investigation on the topic is needed.

1 Einleitung

KI-basierte Systeme sind aktuell eine große Bedrohung für die Sicherheit von Software, da sie schneller Fehler finden und ausnutzen können, als menschliche Entwickler diese ausmerzen. Ein Lösungsansatz ist, Software so zu entwickeln, dass sie weniger Fehler aufweisen kann.

Eine Sprache, die das ermöglicht, ist Lean. Dadurch, dass sie gleichzeitig eine Programmiersprache und ein mathematischer Beweisassistent ist, ermöglicht sie eine formale Verifikation der in ihr implementierten Programme. So kann sicherheitsrelevanter Code mit hoher Vertrauenswürdigkeit produziert werden. Diese Vorzüge werden bereits von großen Organisationen wahrgenommen. Beispielsweise verwendet Amazon Web Services (AWS) Lean, um deren Berechtigungssprache *Cedar* zu verifizieren.[2]

1.1 Eigenschaften von Lean

Um effizienten Code zu schreiben, ist es nötig, die Konzepte der verwendeten Sprache gut zu kennen. Lean unterscheidet sich grundlegend von den meisten weit verbreiteten imperativen oder objektorientierten Programmiersprachen wie Python, Java oder C. Im Gegensatz zu diesen handelt es sich bei Lean um eine „strikte pure funktionale Sprache mit Dependent Types“.[3]

1.1.1 Strikt

Striktheit bedeutet, dass die Argumente einer Funktion vollständig berechnet werden, bevor ihre Anweisungen ausgeführt werden. Ein Beispiel hierfür ist die Funktion `plusFive`, die ein Argument `x` vom Typ `Nat` annimmt und einen Wert vom Typ `Nat` zurückgibt:

```
def plusFive (x : Nat) : Nat := x + 5
```

In diesem Fall würde der Aufruf

```
#eval plusFive (2 + 2)
```

vor der Ausführung in

```
#eval plusFive 4
```

umgeformt und dann erst der Wert `9` zurückgegeben werden. Während es sich um eine Eigenschaft aller genannten Sprachen (Python, Java, C) handelt, gibt es auch Sprachen wie Haskell, die nicht strikt sind. Diese Sprache verwendet sogenannte *lazy evaluation*,[4] was bedeutet, dass sie die Werte von Aussagen erst berechnet, wenn sie benötigt werden.

1.1.2 Pur

Pure Sprachen lassen zur Laufzeit keine Seiteneffekte, also Veränderungen am System außerhalb der Kapselung des gerade ausgewerteten Ausdrucks, zu. Da sie globale Variablen erlauben, ist keine der eingangs erwähnten Sprachen pur. In Lean sind Seiteneffekte nur unter speziellen Umständen möglich. In allen anderen Fällen müssen Funktionsaufrufe deterministisch sein, beispielsweise darf innerhalb

einer Funktion keine Nutzer-Eingabe eingeholt werden, da dies zu unterschiedlichen Rückgabewerten bei gleichen Argumenten führen kann.

Eine weitere Folge dieser Purität ist, dass Variablen ähnlich wie in der Mathematik für einen festen Wert stehen. Dieser kann unbekannt sein, darf sich aber nicht verändern. Das erschwert das Schreiben schnellen Codes, da der Wert einer Variable immer erst kopiert werden muss, um ihn zu verändern. In Ausnahmefällen ist es Lean möglich, diese Vorgabe zu umgehen.

Destruktive Updates Wie die meisten Sprachen muss Lean für seine Garbage Collection nachverfolgen, wie viele Objekte eine Referenz auf einen gewissen Wert speichern. Wenn nur die ändernde Instanz noch eine Referenz auf den zu ändernden Wert hat, wird dies erkannt und der Wert wird direkt verändert und an der betreffenden Speicherstelle belassen. Dieses Ändern eines Werts, ohne ihn vorher zu kopieren, nennt sich destruktives Update. Es verkürzt die Laufzeit von Programmen, da das Kopieren von Daten linear mit deren Größe zunehmende Zeit benötigt. Allerdings gibt es aktuell keine Hilfestellung seitens der Sprache, um sicherzustellen, dass ein destruktives Update stattfindet. Durch geschicktes Design muss die Sprache in Situationen gebracht werden, in denen sie effizient Werte modifizieren kann. Ein wichtiger Schritt bei der Optimierung der Laufzeit ist deswegen, zu prüfen, ob immer destruktive Updates durchgeführt werden.

1.1.3 Funktional

Die Funktionalität von Lean ist seine relevanteste Eigenschaft als Programmiersprache. So wie in den meisten Sprachen Variablen Werte erster Klasse sind, werden in Lean auch Funktionen behandelt. Das bedeutet, dass Funktionen Parameter für andere Funktionen sein können. Die anderen genannten Sprachen bieten diese Funktionalität zwar ebenfalls an, jedoch hat sie keinen so hohen Stellenwert wie in Lean. Ein Beispiel ist die folgende Funktion:

```
def doTwice (f : Nat → Nat) (x : Nat) : Nat :=
  f (f x)
```

Diese hat als Parameter eine Funktion `f`, die natürliche Zahlen auf natürliche Zahlen abbildet, und eine natürliche Zahl. Betrachte man einen Aufruf der Funktion[1]:

```
#eval doTwice plusFive 4
```

Dieser Aufruf wird von Lean in den folgenden Ausdruck umgeformt:

```
#eval plusFive (plusFive 4)
```

Die Funktion `plusFive` wird zweimal angewendet, was zu dem Ergebnis `14` führt.

Es ist auch eine deutlich allgemeinere Version von `doTwice` möglich: Eine Funktion mit einem Parameter beliebigen Typs, die denselben Typ auch zurückgibt, soll beliebig oft angewendet werden. Hierfür sind in Lean zwei Konzepte relevant: Polymorphie und Rekursion.

[1] Lean erwartet Argumente für eine Funktion nur durch Leerzeichen abgetrennt hinter dem Namen der Funktion. Die im Abschnitt 1.1.1 gesetzten Klammern waren erforderlich, da Lean den Ausdruck `plusFive 2 + 2` als `(plusFive 2) + 2` interpretiert.

Polymorphie Das Konzept der Polymorphie (Vielgestaltigkeit) erlaubt es, die Angabe von Typen durch eine Variable zu ersetzen, deren konkreten Wert Lean bei jedem Aufruf der Funktion automatisch ableitet. So kann eine allgemeine Funktion für viele Typen definiert werden. Dafür wird der Typ (meist ein griechischer Buchstabe) in geschweiften Klammern als sogenannter impliziter Parameter angegeben:

```
def doTwicePoly {α : Type} (f : α → α) (x : α) : α :=
  f (f x)
```

Wird diese Funktion jetzt mit den Argumenten `plusFive` und `4` aufgerufen, erkennt Lean automatisch, dass α in diesem Fall den Wert `Nat` hat.

Rekursion Das Aufrufen einer Funktion aus sich selbst heraus wird Rekursion genannt und ist in Lean der bevorzugte Weg, um eine Wiederholungsstruktur zu konstruieren. Dieser Ansatz folgt dem funktionalen Gedanken von Lean. Dabei muss beachtet werden, dass die Rekursion terminiert. In Lean hat das einen besonderen Stellenwert, da sonst die mathematische Korrektheit verletzt werden kann. Deswegen muss für jede Funktion, deren Definition nicht das Keyword `partial` vorangestellt wurde, ein Terminationsbeweis geführt werden. In der Regel ist Lean in der Lage, das automatisch durchzuführen. So auch in diesem Beispiel:

```
def doMultipleTimes {α : Type} (times : Nat) (f : α → α) (x : α) : α :=
  if times > 0 then doMultipleTimes (times - 1) f (f x)
  else x
```

In jedem rekursiven Aufruf gilt: Entweder wird `times` im nächsten Aufruf um 1 kleiner sein und kommt somit dem Abbruchkriterium näher, oder es wurde bereits erreicht und die Funktion hat terminiert. Folglich ist der Terminationsbeweis trivial. Da zudem in jedem Aufruf `f` einmal auf `x` angewendet wird, ist `doMultipleTimes` eine verallgemeinerte Funktion von `doTwice`.

Repräsentation mehrerer Parameter durch *Currying* Der Ausdruck `doMultipleTimes (α := Nat) 2` ist äquivalent zur ursprünglichen Funktion `doTwice`, da beide den Typ `(Nat → Nat) → Nat → Nat` haben. An dieser Notation erkennt man, wie Lean den Typ von Funktionen repräsentiert, die mehrere Parameter haben. Tatsächlich handelt es sich um Funktionen, die selber Funktionen zurückgeben. Dieses Prinzip nennt sich *Currying*. Das Einsetzen des ersten Arguments „entfernt“ dabei die am weitesten links stehende Funktion. `doMultipleTimes` hat den Typ `Nat → (α → α) → α → α`, `doMultipleTimes 2` nur noch den Typ `(α → α) → α → α`. Wird in diese resultierende Verkettung jetzt ein zweites Argument vom Typ `(α → α)` eingesetzt und danach noch ein drittes vom Typ α, bleibt nur der Rückgabewert vom Typ α übrig.

1.1.4 Dependent Types

Ähnlich wichtig wie die Funktionalität von Lean ist das System der Dependent Types, also Typen, die Ausdrücke enthalten, welche selber keine Typen sind. Ein anschauliches Beispiel hierfür ist der Typ `Vector`, der einem Array mit fester Größe entspricht:

```
Vector.{u} (α : Type u) (n : Nat) : Type u
```

Die Länge `n` des Arrays ist bereits im Typ gespeichert. Das erspart es beispielsweise, zu beweisen, dass sich die Größe eines Arrays durch einen Funktionsaufruf wie das Austauschen eines Wertes nicht ändert. Zudem bedeutet die „Abhängigkeit“ von Typen, dass nicht nur der Rückgabe**wert**, sondern auch der Rückgabe**typ** einer Funktion von ihrem Argument abhängen kann.

```
def negativeOrNot (neg : Bool) : if neg then Int else Nat :=
  match neg with
  | true => (-1 : Int)
  | false => (1 : Nat)
```

Falls das Argument `true` ist, wird die ganze Zahl `-1` zurückgegeben, andernfalls die natürliche Zahl `1`. Wie bereits in der Signatur der Funktion erkennbar, hängt der Rückgabetyp vom Argument der Funktion ab. Ein Nebeneffekt der Verwendung von Dependent Types ist, dass auch Typen zu Werten erster Klasse in Lean werden. So kann der `if`-Ausdruck Typen wie `Int` oder `Nat` zurückgeben. Auf ähnliche Weise können nicht nur Programme Typen berechnen, sondern auch Typen Programme enthalten.[5]

Einsatz Besonders relevant sind Dependent Types für die mathematische Beweisführung in Lean. Der *Modus ponens* besagt beispielsweise: Wenn Aussage A Aussage B impliziert und Aussage A wahr ist, dann ist Aussage B wahr. Dieses Theorem wird in Lean folgendermaßen dargestellt[2]:

```
theorem modusPonens {a b : Prop} (proofA : a) (proofAimpB : a → b) : b :=
  sorry
```

Diese Notation erscheint sehr ähnlich zur Definition einer Funktion. In der Tat verhalten sich das `theorem`- und `def`-Schlüsselwort in Lean identisch. Nach der *Curry-Howard-Korrespondenz* können in Lean Beweise durch Funktionen mit Dependent Types repräsentiert und geführt werden.[6] Diese stellt eine Verbindung zwischen der Struktur der Typen und Funktionen und der der logischen Aussagen und Beweise her.

Programme als Beweise Aussagen haben in Lean den Typ `Prop`, sind selber aber auch Typen. Sie enthalten gegebenenfalls ihre Beweise. Wenn also der Typ `a` leer ist, gibt es keinen Beweis für diese Aussage. Kann jedoch durch einen Term ein Element dieses Typs konstruiert werden, so gilt die Aussage als bewiesen. Dieser Ansatz wird als *programs-as-proofs*-Ansatz bezeichnet. Er lässt sich gut in einer Programmiersprache umsetzen, da zur Überprüfung eines Beweises der bereits vorhandene

[2] Das verwendete `sorry`-Axiom beweist jede Aussage, und kann verwendet werden, um vorerst nur die Aussage zu postulieren, dann weiterzuarbeiten und diese später zu beweisen.

Type-Checker eingesetzt wird. Ist der Typ der Rückgabeanweisung identisch mit dem Rückgabetyp, so wurde ausgehend von den Parametern ein Beweis für die gesuchte Aussage konstruiert.

Da Aussagen selber Typen sind, die ihre Beweise enthalten, konnte die Implikation $A \to B$ im oberen Beispiel als Funktion geschrieben werden. Demnach kann diese Funktion aus einem Beweis für A einen Beweis für B konstruieren. Folglich lautet der Beweis für den *Modus ponens*:

```
theorem modusPonens {a b : Prop} (proofA : a) (proofAimpB : a → b) : b :=
  proofAimpB proofA
```

Es reicht, den gegebenen Beweis von a als Argument der Implikationsfunktion einzusetzen, um ein Element vom Typ `b`, also einen Beweis für Aussage B, zu erhalten. Einen auf diese Art geführten Beweis nennt man in Lean „Termbeweis“.

Um nicht nur vom Nutzer geführte Beweise zu prüfen, sondern auch beim Schreiben dieser zu unterstützen, existiert in Lean der Taktikmodus. Er wird mit dem Schlüsselwort `by` eingeleitet und ermöglicht ein vorwärtsgerichtetes Beweisen, während man bei Termen vom Zieltyp ausgehen muss. Hier können zudem automatisierte Taktiken wie `grind` eingesetzt werden. Ein Beispiel für einen Taktikbeweis findet sich im Abschnitt 2.1.1. Der Taktikmodus basiert trotz seiner anderen Beweisführungsrichtung und Syntax auf Termbeweisen. Ähnlich, wie Java-Programme in Maschinencode kompiliert werden, werden Taktikbeweise in Termbeweise überführt.

1.2 Theoretische Betrachtung des Sortierens

1.2.1 Theoretische Grundlagen

Um eine Eingabe möglichst effizient algorithmisch zu sortieren, ist es nötig, die theoretischen Grundlagen der Sortieralgorithmen zu betrachten. Dafür gibt es Eigenschaften, die bei der Betrachtung von Algorithmen relevant sind.

Laufzeitkomplexität Eine dieser Eigenschaften ist die Laufzeitkomplexität. Sie gibt an, wie die Laufzeit eines Algorithmus mit der Anzahl der Elemente zunimmt. Man gibt diese in der *Landau-Notation* an, die die Laufzeitkomplexität in Abhängigkeit von der Größe der Eingabe n beschreibt. So bedeutet zum Beispiel $\mathrm{O}(n^2)$, dass die Laufzeit des Algorithmus quadratisch mit der Eingabegröße zunimmt. Algorithmen haben je nach Eingabe unterschiedliche Laufzeitkomplexitäten, wobei in der Regel auf den schlechtesten, durchschnittlichen und oft auch den besten Fall geachtet wird. Im Best-Case-Fall ist die Laufzeitkomplexität minimal, während sie im Worst-Case-Fall maximal ist. Es hängt vom Algorithmus ab, welche Eingabe welchen Fall darstellt. Um einen Algorithmus zu optimieren, ist es sinnvoll, die Eingaben zu betrachten, die schlechte Fälle darstellen, um Methoden zu finden, diese zu reduzieren.

Die theoretische Grenze für Sortieralgorithmen, die auf Vergleichen basieren, liegt bei $\mathrm{O}(n \log n)$. Algorithmen, die diese Grenze erreichen, werden als optimal bezeichnet.

Speicherkomplexität Eine weitere Eigenschaft ist die Speicherkomplexität. Sie gibt an, wie viel Speicherplatz ein Algorithmus benötigt, um seine Aufgabe zu erfüllen. Auch diese wird häufig in der

Landau-Notation angegeben. Ein Algorithmus mit einer Speicherkomplexität von $O(n)$ benötigt linear mit der Eingabegröße zunehmenden Speicherplatz zusätzlich zu der Eingabe.

Unter anderem für diese Eigenschaft ist es bedeutsam, ob ein Algorithmus in-place arbeitet. Das bedeutet, dass er nur konstant viel Speicherplatz über die Eingabe hinaus benötigt.

Stabilität Stabilität ist eine andere wichtige Eigenschaft, die beim Sortieren häufig betrachtet wird. Ein Algorithmus ist stabil, wenn er die Reihenfolge von Elementen mit demselben Wert beibehält. Das ist dann relevant, wenn man mit Elementen mit mehreren Eigenschaften arbeitet, denn diese können im Sinne des Sortieralgorithmus gleich sein, aber dennoch anderweitig unterscheidbar oder sortiert sein.

1.2.2 Verschiedene Algorithmen

Insertionsort Insertionsort ist dafür bekannt, dass es einfach zu implementieren und für kleine Eingaben oft sehr effizient ist. Dabei wird die Eingabe in eine sortierte und eine unsortierte Partition aufgeteilt. Das erste Element der unsortierten Partition wird genommen und in die sortierte Partition an der richtigen Stelle eingefügt. Dieser Vorgang wird wiederholt, bis die unsortierte Partition leer ist. Schematischer Ablauf:

1. Beginne mit dem zweiten Element der Eingabe. Der Bereich links von diesem Element ist die „sortierte“ Partition.
2. Schiebe das aktuelle Element nach links in die sortierte Partition, bis das Element links von ihm kleiner oder gleich ist.
3. Nimm das erste Element der unsortierten Partition und wiederhole Schritt 2, bis das Ende der Eingabe erreicht ist.

Es ergeben sich für Insertionsort folgende Eigenschaften:

- Im besten Fall, bei schon sortierten Daten, ist die Laufzeitkomplexität $O(n)$.
- Im schlechtesten Fall, bei umgekehrt sortierten Daten, ist die Laufzeitkomplexität $O(n^2)$.
- Die Speicherkomplexität ist $O(1)$, da der Algorithmus *in-place* arbeitet, also keinen zusätzlichen Speicherplatz benötigt.
- Insertionsort ist stabil, da Elemente mit gleichen Werten nicht aneinander vorbeigeschoben werden und so deren Reihenfolge beibehalten wird.

Insertionsort ist effizient bei kleinen und fast sortierten Eingaben. Bei großen, unsortierten Eingaben ist er allerdings sehr ineffizient. Dadurch, dass er *in-place* arbeitet und stabil ist, ist er dennoch weit verbreitet.

Mergesort Mergesort arbeitet nach dem Prinzip „Teile und herrsche“. Dabei wird die Eingabe in zwei Hälften geteilt, die rekursiv sortiert und dann zusammengefügt werden. Schematischer Ablauf:

1. Teile die Eingabe in zwei Hälften.
2. Teile diese Hälften wieder in zwei Hälften und das rekursiv immer weiter, bis die Hälften nur noch ein Element enthalten.
3. Beginnend bei den kleinsten Hälften, vergleiche jeweils die ersten Elemente der beiden Hälften und füge sie so zusammen, dass eine sortierte Liste entsteht.
4. Wiederhole das Vorgehen bei allen Hälften dieser Größe, dann mit der nächstgrößeren Ebene, bis die gesamte Eingabe sortiert ist.

Es ergeben sich für Mergesort folgende Eigenschaften:

- Die Laufzeitkomplexität ist im besten und schlechtesten Fall $O(n \log n)$.
- Die Speicherkomplexität kann theoretisch auf $O(1)$ reduziert werden, wenn der Algorithmus *in-place* implementiert wird. Das ist allerdings sehr komplex und wird in der Praxis selten umgesetzt. In der Standardimplementierung ist die Speicherkomplexität $O(n)$, da zusätzliche Listen für die sortierten Hälften erstellt werden.
- Mergesort ist stabil, weil beim Aufbauen der sortierten Elemente die Reihenfolge beibehalten wird, wenn Elemente mit demselben Wert verglichen werden.

Heapsort Heapsort ist ein Vergleichssortieralgorithmus, der auf der Datenstruktur Binary Heap basiert. Ein Binary Heap ist ein binärer Baum, bei dem jeder Elternteil entweder kleiner oder gleich seinen Kindern ist (Min-Heap) oder größer oder gleich seinen Kindern ist (Max-Heap). Bei dem Max-Heap ist das größte Element die Wurzel des Baums, bei dem Min-Heap diese das kleinste Element.

Schematischer Ablauf:

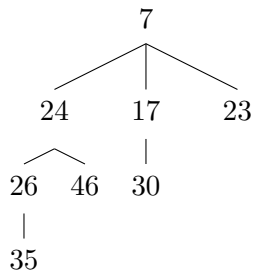


Abbildung 1: Min-Heap

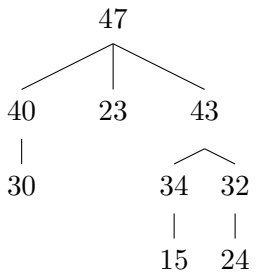


Abbildung 2: Max-Heap

Heapsort funktioniert in zwei Phasen: Zunächst wird die Eingabe in einen Heap umgewandelt (in der Regel einen Max-Heap), und dann werden die Elemente des Heaps in sortierter Reihenfolge extrahiert. Da diese im Heap schon sortiert sind, gewinnt man an Effizienz.

Schematischer Ablauf:

1. Wandle die Eingabe in einen Heap um, indem man das Array als binären Baum interpretiert.
2. Beginnend bei den letzten Elternknoten, tausche diese jeweils mit ihrem größeren Kind, falls dieses größer als der Elternteil ist, und arbeite so bis zur Wurzel durch. Dabei ist jedes Elternelement schließlich größer oder gleich seinen Kindern, sodass ein Max-Heap entsteht.
3. Tausche die Wurzel mit dem letzten Element des Heaps. Das größte Element steht damit an seiner endgültigen Position und wird nicht mehr in den Heap aufgenommen. Das ehemals letzte Element ist nun die neue Wurzel.
4. Stelle die Heap-Eigenschaft wieder her, indem die neue Wurzel mit ihrem größeren Kind getauscht wird, bis die Heap-Eigenschaft wieder erfüllt ist.
5. Wiederhole die Schritte 3 und 4, bis nur noch ein Element im Heap verbleibt.

Es ergeben sich für Heapsort folgende Eigenschaften:

- Die Laufzeitkomplexität ist in der Theorie im besten und schlechtesten Fall $O(n \log n)$. Durch geschickte Implementierung kann die Laufzeit in der Praxis aber bei gleichen Werten auf bis zu $O(n)$ reduziert werden, da die Operation zur Umwandlung in einen Heap in diesem Fall nur einmal pro Element durchgeführt werden muss.
- Die Speicherkomplexität ist $O(1)$, da der Algorithmus in-place arbeitet.
- Heapsort ist aber nicht stabil, da die Reihenfolge von Elementen mit demselben Wert beim Umwandeln in einen Heap verloren gehen kann.

Quicksort Quicksort ist ein weiterer Vergleichssortieralgorithmus, der das Prinzip „Teile und herrsche“ verwendet. Dabei wird nicht wie bei Mergesort die Eingabe in der Mitte geteilt, sondern ein sogenannter Pivot-Wert gewählt, anhand dessen die Eingabe in zwei Teile aufgeteilt wird: Den Teil mit Werten kleiner als der Pivot und den Teil mit Werten größer und gleich dem Pivot (inklusive dem Pivot). Anschließend wird der Prozess rekursiv für die beiden Teile wiederholt. Schematischer Ablauf:

1. Wähle einen Pivot-Wert aus der Eingabe.
2. Teile die Eingabe in Werte kleiner als der Pivot und Werte größer oder gleich dem Pivot (inkl. dem Pivot).
3. Setze die kleineren Werte nach links und die größer-gleichen Werte rechts zusammen.
4. Wiederhole dies für beide Teile rekursiv, bis sie nur noch ein Element enthalten.

Es ergeben sich für Quicksort folgende Eigenschaften:

- Die Laufzeitkomplexität ist im besten Fall $O(n \log n)$, wenn der Pivot-Wert die Eingabe in zwei gleich große Teile teilt. Im schlechtesten Fall, wenn der Pivot-Wert das größte oder kleinste Element ist, beträgt die Laufzeitkomplexität $O(n^2)$.
- Die Speicherkomplexität kann bis zu $O(\log n)$ betragen, weil der Algorithmus theoretisch in-place arbeiten kann und nur den Stack für die Rekursion benötigt. In der Praxis wird allerdings oft $O(n)$ Speicher benötigt, da zusätzliche Listen für die Partitionierung erstellt werden.
- Quicksort ist nicht stabil, denn die Reihenfolge von Elementen mit demselben Wert beim Partitionieren kann verloren gehen.
- Durch die Wahl des Pivot-Werts kann die Leistung von Quicksort stark beeinflusst werden. Eine gute Wahl in der Praxis ist es, den Median des ersten, mittleren und letzten Elements der Eingabe als Pivot zu wählen, um eine gleichmäßige Aufteilung zu gewährleisten. Wenn man mehr Werte in Betracht zieht, riskiert man, bei der Auswahl mehr Aufwand zu erzeugen, als gewonnen wird.
- Quicksort ist in der Praxis oft schneller als andere $O(n \log n)$-Algorithmen, da es direkt im ursprünglichen Array arbeitet und dadurch weniger zusätzlichen Speicher benötigt und weniger Kopiervorgänge notwendig sind. So ist der Grundwert (auch Konstante genannt) der Laufzeit geringer.

1.2.3 Hybridität

Als hybriden Sortieralgorithmus bezeichnet man einen Algorithmus, der mehrere Sortieralgorithmen kombiniert, um die Vorteile der einzelnen Algorithmen optimal zu nutzen und die Nachteile zu minimieren.

Bei Quicksort ist es üblich, für kleine Eingaben auf Insertionsort zurückzugreifen, da die Aufteilung der Eingabe in kleine Teile aufwendig und somit die Rekursion bei kleinen Eingaben ineffizient ist. Insertionsort ist für kleine Eingaben oft schneller, da es weniger Overhead hat und die Laufzeitkomplexität im besten Fall $O(n)$ beträgt. Schematischer Ablauf:

1. Wähle einen Pivot-Wert aus der Eingabe und teile die Eingabe danach in zwei Teile.
2. Wiederhole Schritt 1 rekursiv für beide Teile, bis die Teile eine bestimmte Größe unterschreiten.
3. Sortiere die kleinen Teile mit Insertionsort.

Es gilt dabei zu beachten, dass die Bedingungen, die bestimmen, wann welcher Algorithmus verwendet wird, selbst auch eine Laufzeitkomplexität haben, wodurch umfangreiche hybride Algorithmen in der Praxis nicht sinnvoll sind.

1.3 Aufgabenstellung

Ein wichtiges Element der Standardbibliothek einer Programmiersprache sind Sortieralgorithmen. Ist ein Datensatz sortiert, lassen sich beispielsweise schneller bestimmte Elemente finden. In Lean benötigt zudem die automatisierte Beweistaktik `grind`, ein wesentlicher Bestandteil der Sprache, eine Möglichkeit zum Sortieren.

In der Standardbibliothek von Lean befinden sich im Juli 2026 unter anderem Implementierungen von Quicksort, Mergesort, Insertionsort und Heapsort. Die bisherige Implementierung von Quicksort weist das Problem auf, dass sie ein Array, das nur einen Wert enthält, also zum Beispiel `#[1,1,1,1,1]`, nicht in einem Durchgang sortiert. Stattdessen gerät der Algorithmus hier in einen seiner Worst Cases und benötigt eine quadratisch mit der Array-Länge ansteigende Laufzeit.

Dies zeigt exemplarisch, dass die in Lean bislang enthaltenen Sortieralgorithmen Verbesserungspotenzial aufweisen. Das Ziel des Projekts war es, die Eigenschaften von Lean kennenzulernen und zu prüfen, wie durch optimierte Sortieralgorithmen die Standardbibliothek von Lean verbessert werden kann. Neben einer größeren Performanz sollten die entwickelten Algorithmen formal verifiziert werden und hybride Ansätze an das Sortieren aufgreifen.

2 Implementierungen der Algorithmen

Das Überführen der in Abschnitt 1.2.2 aufgeführten Algorithmen in Lean birgt einige Herausforderungen, die im vorherigen Kapitel schon angedeutet wurden. Darunter sind:

- Lean führt destruktive Updates nur aus, wenn keine andere Referenz auf einen Wert existiert. Ansonsten wird der Wert kopiert, was bei langen Arrays viel Laufzeit benötigt.
- Statt Schleifen wird in Lean Rekursion verwendet. Die Notation von Wiederholungsstrukturen ist also im Vergleich zu anderen gängigen Sprachen ungewohnt. Zudem sind Überlegungen zur Stackgröße erforderlich, da Funktionen im Gegensatz zu Schleifen Speicher im Stack belegen können.
- Lean benötigt an vielen Stellen Beweise, zum Beispiel, dass ein Array-Zugriff tatsächlich in bounds ist oder dass eine Funktion terminiert.

Außerdem muss unabhängig von der Sprache eine möglichst gute Implementierung gewählt werden, die zum Beispiel sehr speichereffizient ist.

2.1 Entscheidungsfindungsphase

In Vorversuchen wurden verschiedene Algorithmen in Lean implementiert und bezüglich ihres Verbesserungspotenzials beurteilt. Auf Basis dieser Ergebnisse erfolgte die weitere Umsetzung. Im Folgenden werden ausgewählte Zwischenschritte vorgestellt.

2.1.1 Mergesort

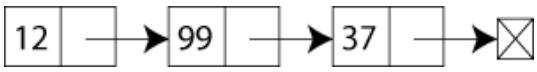


Abbildung 3: Eine Visualisierung einer `List Nat`.[7]

Der Mergesort-Algorithmus wurde in mehreren, fortlaufend optimierten Versionen implementiert. Die erste Implementierung basierte auf Listen. Unter Listen versteht man Verkettungen von im Speicher verteilten Werten, die zusätzlich einen Zeiger auf den nächsten Wert speichern (s. Abbildung 3). Ein Vorteil von Listen ist, dass das Einfügen von Werten vergleichsweise einfach möglich ist. Allerdings muss zum Lesen eines Wertes auf alle davor liegenden Werte zugegriffen werden, was bedeutet, dass die Laufzeitkomplexität dieser Operation $O(n)$ ist.

Diese Implementierung ist dennoch in der Hinsicht sinnvoll, dass Listen in Lean einfach zu verwenden sind und kompakten Code ermöglichen. Außerdem lassen sich Eigenschaften eines Algorithmus, der Listen verwendet, deutlich einfacher formal verifizieren.

mergeSortList Der Algorithmus ist kompakt und lautet wie folgt:

```
def mergeSortList (xs : List α) (isSmallerThan : α -> α -> Bool) : List α :=
  match xs with
  | [] => []
  | y :: ys =>
    if h : ys.length = 0 then [y] else
    let half := (y :: ys).length / 2
    let left := (y :: ys).take half
    let right := (y :: ys).drop half
    List.merge (mergeSortList left isSmallerThan) (mergeSortList right isSmallerThan)
      isSmallerThan
termination_by xs.length
```

Der `match`-Ausdruck lässt den Algorithmus sofort abbrechen, falls die zu sortierende Liste leer ist. Sollte diese aber durch ein Element mit einer daran angehängten Liste dargestellt werden können, läuft der Algorithmus weiter. Die angehängte Liste `ys` kann ihrerseits auch leer sein. Dann wird das eine in `xs` enthaltene Element unverarbeitet zurückgegeben. Ansonsten teilt der Algorithmus die zusammengefügte Liste in zwei Hälften auf und „mergt“ die beiden rekursiv sortiert wieder zusammen (hier mit `List.merge` aus der Standardbibliothek, Beispiel für ähnliche Funktion auf Seite 38). Dazu wird die mitübergebene Ordnungsrelation `isSmallerThan` verwendet. Schließlich benötigt Lean noch eine Hilfestellung, um die Termination der Funktion zu beweisen. Das Schlüsselwort `termination_by` weist Lean an, zu prüfen, ob die angegebene Variable in jedem rekursiven Aufruf der Abbruchbedingung näher kommt. Mit dieser Hilfe kann Lean den Terminationsbeweis selbständig führen.

Beispiel für die formale Verifikation Da die Funktion `mergeSortList` vergleichsweise übersichtlich ist, gestaltet sich die formale Verifikation ihrer Eigenschaften weniger komplex. Dies soll exemplarisch für eine Eigenschaft demonstriert werden: Alle Sortieralgorithmen haben unter anderem gemein, dass sie eine Permutation der Ursprungsliste erzeugen. Das bedeutet, dass eine Liste dieselben Elemente wie eine zweite enthält, nur in einer anderen Reihenfolge. Diese Eigenschaft nennt sich in Lean `List.Perm`. Ihr Beweis beginnt wie folgt:

```
theorem mergeSortListPerm (xs : List α) :
  xs.Perm (mergeSortList xs isSmallerThan) := by
  sorry
```

Am Schlüsselwort `by` erkennt man, dass der Taktikmodus verwendet wird, um den Beweis aufzubauen. Da über eine rekursive Funktion argumentiert wird, wird die Methode der vollständigen Induktion zur Zerlegung des Problems angewendet. Die Aussage muss bei einem beliebigen Wert der Induktionsgröße gelten, wenn sie für alle[3] kleineren Werte dieser auch galt. Dann gilt sie für alle Werte der Induktionsgröße

[3] Diese Variante der Induktion nennt man „stark“. „Normale“ Induktion ist hier nicht ausreichend, da sie ihre Induktionshypothese nur für den Vorgänger der Induktionsgröße bereitstellt. Die Induktionsgröße `xs.length` halbiert sich aber in jedem Durchgang.

und somit allgemein. Im gegebenen Fall ist `xs.length` die Induktionsgröße, da dies die Zahl ist, die auch für den Terminationsbeweis verwendet wurde. Der Beweis durch Induktion wird wie folgt eingeleitet:

```
induction h : xs.length using Nat.strongRecOn generalizing xs with
  | ind n ih => sorry
```

Die Induktionsgröße hat hier den Namen `n`. Zusätzlich erhalten wir von Lean die Induktionshypothese `ih : ∀ (m : Nat), m < n → ∀ (xs : List α), xs.length = m → xs.Perm (mergeSortList xs isSmallerThan)`. Ausformuliert lautet diese: „Wenn `m` eine natürliche Zahl kleiner als `n` ist und die Liste `xs` die Länge `m` hat, dann erzeugt ein Aufruf von `mergeSortList` auf diese Liste eine Permutation von ihr“. Mithilfe der Induktionshypothese wird später der allgemeine Fall bewiesen.

Zuerst werden aber die Fälle betrachtet, die den Algorithmus abbrechen lassen. In dem Fall, dass die Länge der Liste Null ist, wird die Liste direkt wieder zurückgegeben. Dass eine leere Liste eine Permutation von sich selbst ist, ist für Lean trivial. Daher muss man in diesem Fall Lean nur anweisen, das Ziel durch Anwendung von `mergeSortList` zu vereinfachen. Im anderen Fall reicht dies noch nicht zum Schließen des Ziels aus, wie man im Fall `y :: ys` erkennt:

```
match xs with
 | [] => simp [mergeSortList]
 | y :: ys =>
   simp [mergeSortList]
   sorry
```

An diesem Punkt zeigt Lean folgendes Ziel an:

```
(y :: ys).Perm
  (if ys.length = 0 then [y]
  else
    (mergeSortList (List.take ((ys.length + 1) / 2) (y :: ys)) isSmallerThan).merge
      (mergeSortList (List.drop ((ys.length + 1) / 2) (y :: ys)) isSmallerThan)
 isSmallerThan)
```

Es ist folglich noch eine Fallunterscheidung nötig, da auch eine Liste mit nur einem Element immer eine Permutation von sich ist:

```
by_cases hlen : ys.length = 0
   · simp [hlen, ← List.length_eq_zero_iff]
   · simp [hlen]
     sorry
```

Die Anwendung von `← List.length_eq_zero_iff` ist dabei nötig, um zwischen der Schreibweise `ys.length = 0` und `ys = []` zu übersetzen.

Für die Betrachtung der rekursiven Aufrufe werden zunächst einige Namen eingeführt.

```
let left := List.take ((ys.length + 1) / 2) (y :: ys)
let leftsorted := mergeSortList left isSmallerThan
let right := List.drop ((ys.length + 1) / 2) (y :: ys)
let rightsorted := mergeSortList right isSmallerThan
let merged := leftsorted.merge rightsorted isSmallerThan
have hsplit_perm : (y :: ys).Perm (left ++ right) := by
   rw [List.take_append_drop]
sorry
```

Dass die Konkatenation (Verkettung) der beiden Hälften eine Permutation der Ursprungsliste ist, ist später für die Beweisführung nötig. Daher wurde diese Aussage schon jetzt durch einen `have`-Ausdruck bewiesen. Im nächsten Schritt wird die Induktionshypothese angewendet[4], um zu zeigen, dass das rekursive Sortieren von `left` und `right` Permutationen dieser Listen erzeugt:

```
have hleft_perm : left.Perm leftsorted := ih left.length (by grind) left (by grind)
have hright_perm : right.Perm rightsorted := ih right.length (by grind) right (by grind)
sorry
```

Darauf basierend kann gezeigt werden, dass die Konkatenation der beiden Teillisten eine Permutation der Konkatenation ihrer sortierten Varianten ist. Weiterhin gilt, dass auch ein Aufruf von `merge` bezüglich der Permutation wie eine Verkettung wirkt. Die dafür nötigen Theoreme sind bereits in Lean enthalten.

```
have hsorted_append : (left ++ right).Perm (leftsorted ++ rightsorted) :=
   hleft_perm.append hright_perm
have hmerged : (left ++ right).Perm merged :=
   hsorted_append.trans (List.merge_perm_append isSmallerThan).symm
sorry
```

Da die Konkatenation von `left` und `right` laut `hsplit_perm` auch eine Permutation der Ausgangsliste ist, kann der Beweis an dieser Stelle geschlossen werden:

```
exact hsplit_perm.trans hmerged
```

Der beschriebene Beweis ist als vollständige, unkommentierte Version im Appendix (s. S. 37) hinterlegt.

Mergesort mit Arrays Der erste Schritt zur Verbesserung der Implementierung von Mergesort ist das Ersetzen von Listen durch Arrays. Diese haben in Lean eine veränderbare Größe und die Elemente liegen direkt nebeneinander im Speicher. Daher beträgt die Zugriffszeit $O(1)$. Allerdings gibt es in der Standardbibliothek von Lean kein `Array.merge`, deshalb muss diese Funktion selbst implementiert werden. Der Ansatz dafür ist, die zu „mergenden“ Arrays beide beim Index 0 beginnend durchzugehen und dabei jeweils das kleinere Element in die Rückgabeliste zu übertragen. Die vollständige

[4] In diesen und den folgenden `have`-Ausdrücken wird nicht mit `by` der Taktikmodus geöffnet, daher handelt es sich um normale Termbeweise. So lassen sich beide Beweiswege kombinieren.

Implementierung der Funktion `myMerge` befindet sich im Appendix (s. S. 38). Beim Betrachten der Signatur fallen zwei Besonderheiten auf:

```
def myMerge {α} (le : α → α → Bool) (xs ys : Array α) (ix iy : Nat) (work : Array α)
    (hwork : work.size = xs.size + ys.size) (hix : ix ≤ xs.size) (hiy : iy ≤ ys.size) :
    { merged : Array α // merged.size = work.size } :=
    sorry
```

Zum einen verlangt die Signatur Beweise über einige Größeneigenschaften der übergebenen Arrays und deren Verhältnis zu den verwendeten Zählern. Dies ist nötig, um beweisen zu können, dass die Array-Zugriffe in der Funktion in bounds sind. So können entsprechende Prüfungen zur Laufzeit eingespart werden und der Code läuft schneller.

Aus ähnlichem Grund ist auch der Rückgabetyp als sogenannter *Subtyp* gewählt worden. Dieser legt beweisbare Eigenschaften fest, die der Rückgabewert zusätzlich erfüllen muss. Hier muss die Größe des zurückgegebenen Arrays genauso groß wie die des `work`-Arrays und somit die Summe der Größen von `xs` und `ys` sein. Diese Eigenschaft benötigt die aufrufende `myMergeSort`-Funktion, um wiederum ihre Indexbeweise zu führen. Der Rückgabebefehl in `myMerge` sieht an einer Stelle beispielsweise so aus: ⟨`work, rfl`⟩. Dabei ist `work` das zurückzugebende Array und `rfl` ein Term, der die angegebene Eigenschaft beweist.

Die Implementierung des Hauptalgorithmus `myMergeSort` (s. S. 38) unterscheidet sich nur durch die nötigen Indexbeweise von der schon vorgestellten `mergeSortList`-Implementierung.

Bottom-up-Mergesort mit „Array-Ping-Pong“ Die gezeigte Implementierung von Mergesort ist zwar besser, aber an einigen Stellen ineffizient. So ist sie nicht *tail-recursive* (Endrekursiv). Eine Funktion ist das nur dann, wenn ihr rekursiver Aufruf immer der letzte Befehl ihrer Ausführung ist. In diesem Fall muss der Compiler keine Daten mehr aus dem aktuellen Stackframe behalten, da das Gesamtergebnis ausschließlich vom Ergebnis des kommenden Aufrufs abhängt. Wenn rekursive Funktionen so implementiert werden, erreichen sie ähnliche Speichereffizienz wie Zählschleifen in anderen Sprachen. Dies ist beim bisherigen Ansatz an Mergesort nicht möglich, weil der Aufruf von `myMerge` stets zuletzt stehen muss, während die zwei rekursiven Aufrufe bereits davor erfolgt sein müssen.

Aus diesem Grund wurde ein anderer Ansatz gewählt: Das zu sortierende Array wird in Zweierpaare aufgeteilt, die dann „gemergt“ werden. So entstehen viele jeweils zwei Elemente lange und sortierte Slices. Diese werden dann erneut mit ihrem Nachbarn „gemergt“, bis daraus ein komplett sortiertes Array entstanden ist. Der Baum, den man sich bei Mergesort gedanklich vorstellt, wird hier von unten aufgebaut.

Des Weiteren wird die Nutzung von Speicher für die Arrays optimiert. Bei der vorherigen Implementierung werden beispielsweise beim Halbieren der Arrays immer wieder neue Arrays allokiert. Dies benötigt unnötig viel Laufzeit. Es ist ausreichend, zwei Arrays zu allokieren, die so groß sind wie das zu sortierende Array. So kann in jedem Schritt das Ergebnis in das andere Array geschrieben werden. Danach wird die Richtung umgedreht und der nächste Schritt des „Mergens“ geschieht. Dafür

ist es nötig, die `myMerge`-Funktion so umzuschreiben, dass sie zwei Abschnitte eines größeren Arrays verrechnen kann, anstatt nur zwei komplette Arrays zu kombinieren.

Bei diesem Ansatz handelt es sich um die höchste Optimierungsstufe, die wir für Mergesort erreicht haben. Die genaue Implementierung findet sich in einer kommentierten Fassung ohne Indexbeweise im Appendix (s. S. 39).

2.1.2 Heapsort

Die Implementierung von Heapsort weist wenige Besonderheiten auf. Allerdings wurde hier ein weiteres Mittel zur Optimierung von Lean-Code verwendet: Attribute von Funktionen. Diese geben dem Compiler Anweisungen, wie er mit Funktionen beim Kompilieren umgehen soll.

Ein Beispiel ist bei Heapsort die Funktion[5], die den Index des linken Kindes eines Knotens berechnet:

```
def lCh := (· * 2 + 1)
```

Da diese so kurz ist, wäre es sinnvoller, den Aufruf dieser Funktion nicht als Funktionsaufruf zu kompilieren, sondern die Anweisungen an den entsprechenden Stellen direkt einzusetzen. Das nennt man *inlining*. Um dies dem Compiler mitzuteilen, schreibt man:

```
@[inline]
def lCh := (· * 2 + 1)
```

Des Weiteren haben viele andere Funktionen bei Heapsort als einen Parameter die Ordnungsrelation `lt : α → α → Bool`. Die Funktionen sind sehr groß, daher wäre *inlining* kontraproduktiv. Aus der Praxis weiß man, dass häufig die Ordnungsrelation vom Typ `Nat → Nat → Bool` verwendet wird. Es wäre also sinnvoll, eine speziell vorkompilierte Version von beispielsweise `siftDown` (s. S. 41) für diesen Fall vorzuhalten. Das vergrößerte Kompilat wird in diesem Fall in Kauf genommen, um bessere Performance zu erzielen. Dies wird dem Compiler durch das Attribut `@[specialize lt]` mitgeteilt.

2.1.3 Quicksort

Um Quicksort *tail-recursive* implementieren zu können, kann man eine Liste von noch zu erledigenden Aufgaben nutzen. Dafür wird ein eigener Typ `Todo` definiert:

```
inductive Todo (α : Type)
| SortArr : Array α → Todo α
| PushVal : Nat → α → Todo α
```

Ein Element dieses Typs kann auf zwei Wege erzeugt werden: Entweder durch den Aufruf von `SortArr` mit einem entsprechenden Array als Argument oder durch den Aufruf von `PushVal` mit einer natürlichen Zahl und einem Wert vom Typ α.

[5]Es handelt sich hier um eine Kurzschreibweise für die Lambda-Notation. Der Punkt steht dabei für den Parameter.

So kann über eine `match`-Expression herausgefunden werden, was zu tun ist:

```
match (t : Todo) with
| Todo.PushVal n x => sorry --den Wert x n-Mal an das Rückgabearray anhängen
| Todo.SortArr arr => sorry --das Array arr sortieren (sodass danach gepusht werden kann)
```

Eine weitere Charakteristik dieser Implementierung ist die Verwendung des *median of three* zur Auswahl des Pivot-Elements. Wird das erste, mittlere und letzte Element betrachtet und nachfolgend das mittelgroße der drei Elemente ausgesucht, sorgt das dafür, dass deutlich seltener ein ungünstiger Pivot gewählt wird.

Zudem erkennt der Algorithmus replizierte Elemente im Array und fasst diese zu einem `Todo.Push` zusammen. Folglich setzt dieser Ansatz konzeptuelle Verbesserungen gegenüber der aktuellen Implementierung von `Array.qsort` in der Standardbibliothek von Lean um. Dies wird vereinfacht durch die Verwendung der `[Ord` α`]`-Typklasse. Sie bietet eine Funktion `compare` an, die zwei Elemente vergleicht und zurückgibt, ob das zweite größer als (`.gt`), kleiner als (`.lt`) oder gleich wie (`.eq`) das erste Element ist. Die verwendete Typklasse ermöglicht hier somit Ähnliches wie Interfaces in objektorientierten Sprachen.

In diesem Algorithmus wurde der hybride Gedanke aufgegriffen. Ist ein zu sortierendes Subarray weniger als 28 Elemente groß, wird der Insertionsort-Algorithmus der Standardbibliothek zum Sortieren verwendet, was die Rekursionstiefe begrenzt.

Die Implementierung hat aber auch grundlegende Schwächen. So ist Quicksort einer der Sortieralgorithmen, der theoretisch in-place implementiert werden kann, was hier nicht umgesetzt wurde. Stattdessen werden an vielen Stellen neue Arrays allokiert, was viel Zeit benötigt (s. S. 43).

2.2 Pull-Request: Quicksort mit *Dutch National Flag*-Algorithmus

Die Schwächen der in Abschnitt 2.1 vorgestellten Version werden in der als Ersatz für die aktuelle Version von `Array.qsort` vorgeschlagenen Implementierung behoben. Deren Komponenten werden hier nacheinander vorgestellt (Implementierung s. S. 45ff.).

2.2.1 Insertionsort

Um die Rekursionstiefe zu begrenzen, werden sehr kleine übrige Slices mit dem Insertionsort-Algorithmus sortiert. Dieses Vorgehen hat sich bei ähnlichen Projekten, zum Beispiel in der Sprache Rust, bewährt.[8] Die gewählte Implementierung ist kompakt. Die Variable `i` speichert den ersten Index des noch unsortierten Abschnitts der zu sortierenden Slice `[sllo...slhi)`. Falls `i` die (exklusive) Obergrenze der Slice erreicht, ist diese fertig sortiert und das Array wird zurückgegeben. Wenn nicht, dann wird `i` an die lokal definierte, rekursive Funktion `movedown` übergeben. Diese prüft, ob der Wert am Index `i - 1` kleiner als der am Index `i` ist. Falls ja, ist der Wert richtig einsortiert. Andernfalls werden die Werte vertauscht. Das wird so oft wiederholt, bis der Wert richtig einsortiert ist oder das untere Ende der Slice erreicht wurde.

Wenn die Funktion `movedown` einmal durchgelaufen ist, ist der sortierte Abschnitt um 1 größer geworden. Somit wird nun `insertionSort` mit dem Argument `(i+1)` rekursiv aufgerufen, um den nächsten Durchlauf zu starten.

2.2.2 Auswählen des Pivot-Elements

Die Funktion `pivotselect` wählt genauso wie bei der schon geschilderten Implementierung von Quicksort den *median of three* als Pivot-Element aus. Allerdings wird hier dessen Index zurückgegeben. Außerdem werden Beweise übergeben, dass das Pivot-Element in der angegebenen Slice liegt, was später für den Terminationsbeweis des gesamten Algorithmus zwingend notwendig ist.

2.2.3 Partitionierung

Die meisten Implementierungen von Quicksort teilen das zu sortierende Array in zwei Partitionen auf. Schon im Abschnitt 2.1.3 wurde gezeigt, dass es effizienter sein kann, Elemente, die gleich dem Pivot-Element sind, gesondert zu betrachten.

Das *Dutch National Flag Problem* In seinem Buch[9] „A discipline of programming“ stellt Edsger Dijkstra 1976 folgendes Problem vor: Ein Computer hat vor sich eine Reihe von Eimern. Diese enthalten Kieselsteine in den Farben weiß, rot und blau, die zufällig angeordnet sind. Die einzigen Möglichkeiten des Computers zur Interaktion mit diesen sind, die Farbe eines Steines zu betrachten und die Steine an zwei Positionen zu vertauschen. Mithilfe dieser Interaktionsmöglichkeiten soll er auf möglichst effiziente Weise die Steine so umordnen, dass sie von links nach rechts die niederländische Nationalflagge ergeben: Erst rot, dann weiß und blau.

Um dieses Ziel zu erreichen, definiert Dijkstra drei Variablen, die die Eimer in vier Bereiche einteilen. Man könnte beispielsweise die Namen `white`, `unprocessed` und `fin_unprocessed` wählen, die folgende Bereiche abgrenzen:

Rote Steine	*Weiße Steine*	*Unverarbeitet*	*Blaue Steine*

Abbildung 4: Partitionierung des Arrays

- `[0...white)` enthält die roten Steine,
- `[white...unprocessed)` enthält die weißen Steine,
- `[unprocessed...fin_unprocessed]` enthält die noch nicht betrachteten Steine und
- `(fin_unprocessed...`*Anzahl der Eimer*`)` die blauen Steine.

Dabei dürfen alle Bereiche auch leer sein.

Der folgende Algorithmus löst das Problem:

1. Betrachte den Stein an der Stelle `unprocessed`. Ist er...
   - ...rot, dann vertausche ihn mit dem an der Stelle `white` und erhöhe dann `white` und `unprocessed` um 1.
   - ...weiß, dann belasse ihn an seiner Stelle und erhöhe `unprocessed` um 1.
   - ...blau, dann vertausche ihn mit dem an Stelle `fin_unprocessed` und reduziere `fin_unprocessed` um 1.
2. Wiederhole dies, solange noch Steine im unbearbeiteten Abschnitt sind, also `unprocessed` $\leq$ `fin_unprocessed` gilt.

Dieser Algorithmus kann auch zum Partitionieren eines Arrays für Quicksort verwendet werden. Man legt dabei fest, dass Elemente, die kleiner als das Pivot-Element sind, roten Steinen entsprechen, gleiche Elemente weißen Steinen und größere Elemente den blauen Steinen entsprechen.

Vorteile des Partitionierens mittels des *Dutch National Flag*-Algorithmus Gegenüber der in Abschnitt 2.1.3 vorgestellten Variante ergeben sich bei Verwendung dieses Algorithmus zwei zentrale Vorteile.

Erstens ermöglicht er es, Objekte korrekt zu sortieren, die bezüglich ihrer Ordnungsrelation gleich sind, aber trotzdem Unterschiede in anderen Attributen besitzen. So können beispielsweise zwei Textdateien exakt gleich groß sein, aber unterschiedliche Texte enthalten. Der bisherige Ansatz würde, falls eine der Dateien als Pivot-Element gewählt würde, diese kopieren und zweimal in das nach Größe sortierte Array einfügen, da er sie zu einem `Todo.PushVal` zusammenfasst. Durch den neuen Ansatz bleiben beide Dateien erhalten.

Zweitens ist auf diese Art und Weise eine In-place-Implementierung von Quicksort möglich. Anstatt neue Arrays für die „kleiner-als“- und „größer-als“-Partitionen zu allokieren, wird das zu sortierende Array nur umgeordnet. Da sich der *Dutch National Flag*-Algorithmus mit destruktiven Updates und *tail-recursive* implementieren lässt, ist es möglich, ohne viel Speicherverbrauch in Stack und Heap zu partitionieren.

Termination von Quicksort Der Terminationsbeweis des gesamten Quicksort-Algorithmus hängt von der Implementierung des Partitionierungsalgorithmus ab. Die allgemeine Begründung für die Termination von der vorliegenden Version von Quicksort ist, dass die zu sortierende Slice in mindestens einem der beiden rekursiven Aufrufe kleiner wird, da das Pivot-Element weder größer noch kleiner als es selbst ist und folglich in keine dieser beiden Slices eingeordnet wird. Um dieses Argument in Lean zu formalisieren, ist es nötig, dass die verwendete Ordnungsrelation reflexiv ist, also ein Wert bezüglich dieser gleich sich selbst ist. Diese Anforderung stellt die aktuelle Implementierung von Quicksort in der Standardbibliothek nicht, daher sollte sie aus Kompatibilitätsgründen nicht eingeführt werden.

Die vorgeschlagene Lösung besteht aus zwei getrennten Funktionen, wobei beiden der von Dijkstra vorgestellte Algorithmus zugrunde liegt. Sie unterscheiden sich vor allem in einer Invarianz, deren Beweis als Parameter angegeben ist: Bei der als `dnfstage1` bezeichneten Funktion dürfen wie bei Dijkstra alle Partitionen leer sein. `dnfstage2` fordert aber, dass die Partition mit den Werten, die gleich dem Pivot-Element sind, nicht leer ist.

Die Position des Pivot-Elements wird im Verlauf beider Funktionen mitverfolgt. Sobald es von der Funktion `dnfstage1` in die „gleich-wie“-Partition eingeordnet wird, wird `dnfstage2` aufgerufen. So kann gezeigt werden, dass durch die Partitionierung die Gesamtzahl noch zu sortierender Elemente sinkt, ohne zusätzliche Eigenschaften von der verwendeten Ordnungsrelation zu fordern.

So wird auch die Reflexivität vermieden: Es muss nicht gezeigt werden, dass ein Element gleich sich selber ist, da man auf die Annahme baut, dass wenn der Index gleich ist, auch das Element gleich ist.

Der beschriebene zweistufige Ablauf wird begonnen durch den Aufruf der mit `@[inline]` markierten Wrapper-Funktion `dnfstaged`.

2.2.4 Gesamtablauf

Die Funktion `quicksorthelper` kombiniert die bisher vorgestellten Elemente zum vollständigen Algorithmus:

```
def quicksorthelper [Ord α]
  (xs : Vector α  size) (sllo slhi : Nat)
  (hslloslhi : sllo ≤ slhi) (hslhi : slhi ≤ size)
  : Vector α size :=

  if hfin : slhi - sllo ≤ 1 then xs else

  if slhi - sllo ≤ 16 then insertionSort xs sllo slhi (sllo + 1) (by omega) (by omega) (by
    omega) (by omega) else

  let pvt := pivotselect xs sllo slhi (by omega) (by omega)
  let ⟨(ys, mid, hi), ⟨h1, h2, h3⟩⟩ := (dnfstaged xs pvt sllo slhi (by omega) (by omega) (by
    omega))

  have hterm : slhi - hi < slhi - sllo := by simp only [] at h1 h2 h3; omega
  let ys' := quicksorthelper ys hi slhi (by omega) (by omega)

  have hterm2 : mid - sllo < slhi - sllo := by simp only [] at h1 h2 h3; omega
  quicksorthelper ys' sllo mid (by omega) (by omega)

termination_by slhi - sllo
```

Zuerst wird die Abbruchbedingung geprüft. Falls die zu sortierende Slice ein Element hat oder leer ist, so wird `xs` zurückgegeben. Anschließend wird, falls es sich nur um eine kurze Slice handelt, Insertionsort zum Verringern der Rekursionstiefe aufgerufen.

Im anderen Fall wird zuerst ein Pivot-Element ausgewählt und dann das Array entsprechend partitioniert. Zurückgegeben werden dann neben dem neuen Array[6] die Partitionsgrenzen `mid` und `hi`. Außerdem erhält das Programm Beweise für relevante Invarianzen, allen voran die Aussage, dass die „gleich-wie"-Partition nicht leer ist. Mithilfe dieser Beweise `h1, h2, h3` können die Terminationsbeweise `hterm, hterm2` geführt werden. Anschließend werden die beiden noch zu sortierenden Partitionen rekursiv sortiert. Da hier anders als beispielsweise bei Mergesort keine Operation wie eine Konkatenation oder ein „Mergen" mehr nötig ist, kann wenigstens einer der rekursiven Aufrufe als speichereffizienter *tail-call* umgesetzt werden.

2.2.5 Wrapper

Weil die neue Implementierung voll kompatibel mit der bisherigen sein soll, muss auch eine Variante angeboten werden, die nicht `[Ord α]`, sondern eine Funktion `lt : α → α → Bool` als Parameter hat. Eine `[Ord α]`-Typklasse kann sehr einfach durch das Definieren der `compare`-Funktion erreicht werden. Für die Umwandlung wird die folgende Funktion verwendet:

```
@[inline, reducible]
def ordOfLt (lt : α → α → Bool) : Ord α where
  compare a b :=
    if lt a b then .lt
    else if lt b a then .gt
    else .eq
```

Um die Indexsicherheitsanforderungen von `quicksorthelper` zu erfüllen und gleichzeitig dieselbe Signatur wie bisher zu bewahren, werden `sllo` und `slhi` durch die `min`-Aufrufe in bounds gehalten. Außerdem werden Varianten des Algorithmus für die intern bereits verwendeten Vektoren und für Arrays angeboten.

Alle beschriebenen Eigenschaften werden im Listing der für den Pull-Request veränderten Datei dargestellt (s. S. 45 ff.).

[6] Dieses liegt dank destruktiver Updates an derselben Speicherstelle, hat aber den neuen Namen `ys`.

3 Ergebnisse

3.1 Benchmarken der Algorithmen

Die „Performance“ eines Algorithmus ist schwer messbar. Es können hierfür viele Größen wie die Laufzeit in Millisekunden für verschiedene Arrays, die Anzahl der Maschinencode-Instruktionen des Kompilats oder die Laufzeitkomplexität betrachtet werden.

Die Aussagekraft der Laufzeitkomplexität ist begrenzt. So hat Quicksort mit einer Worst-Case-Komplexität von $O(n^2)$ in vielen Fällen eine geringere Laufzeit als Mergesort mit $O(n \log n)$.

Die Laufzeit ist zudem nicht nur von der Länge eines Arrays abhängig, sondern auch von der Verteilung seiner Werte: Die aktuell in Lean enthaltene Version von `Array.qsort` sortiert zufällig verteilte Arrays schnell, aber benötigt lange, um ein Array von replizierten Elementen zu sortieren. Das liegt daran, dass sie zwangsläufig alle Elemente in die „größer-gleich“-Partition einsortiert und deswegen die maximale Rekursionstiefe erreicht.

Die Anzahl der Maschinencode-Instruktionen des Kompilats ist ebenfalls kein gutes Indiz, da beispielsweise das `@[specialize]`-Attribut für ein sehr großes Kompilat sorgt und trotzdem die Laufzeit des Algorithmus verbessern kann.

Der in dieser Arbeit gewählte Ansatz zum Benchmarken der Implementierungen fokussiert sich auf die Laufzeit für unterschiedlich große Arrays mit zufällig gemischten aufsteigenden Werten vom Typ `Nat`, einem replizierten Wert sowie auf- und absteigend sortierten Werten. Dies birgt den Vorteil, dass es sich bei der Laufzeit um eine sehr praxisnahe Messgröße handelt. Das „Radar“-Benchmark, das den Einfluss von Modifikationen an Lean misst, erfasst unter anderem die Veränderung der Anzahl an Instruktionen im Vergleich zur aktuellen Version von Lean und wurde ebenfalls betrachtet.

Implementierung der Benchmarks Wenn man die Laufzeit messen möchte, muss man sicherstellen, dass dem Compiler nicht auffällt, dass man das Ergebnis des Sortierens nicht weiterverwendet. Ansonsten kann es sein, dass der Compiler den „nutzlosen“ Aufruf des Algorithmus entfernt, was unbrauchbare Ergebnisse liefert.

Deswegen werden die Benchmarks in der `do`-Umgebung ausgeführt. Da diese Seiteneffekte zulässt, kann hier der Compiler keine unerwünschten Optimierungen vornehmen. Zusätzlich wird das `@[noinline]`-Attribut verwendet, das *inlining* als eine weitere Form der hier unerwünschten Optimierung aktiv unterbindet. Dafür eignet sich zum Beispiel dieser Wrapper:

```
@[noinline]
def sortIO [Ord α] [Inhabited α] (xs : Array α) (sort : Array α → Array α) :
   IO (Array α) := do
   return sort xs
```

Darauf aufbauend lässt sich ein Benchmark implementieren, das beispielsweise die Laufzeit für ein umgekehrt sortiertes Array misst:

```
def benchmarkDSC (tests n : Nat) (sort : Array Nat → Array Nat) (title : String) :
  IO Unit := do
  let arr := (Array.range n).reverse
  IO.println s!"{title}"
  for i in 0...tests do
    let before ← Std.Time.Timestamp.now
    let sortArr ←  sortIO arr sort
    let duration ← before.since
    let dms := duration.toMilliseconds
    let ind := match (sortArr == arr.reverse) with
    | true => "good"
    | false => "not sorted"
    IO.println s!"{i} | {dms}ms | {ind}"
  return ()
```

3.2 Vergleich der Sortieralgorithmen

Im Folgenden werden die Messergebnisse der verschiedenen Algorithmen für unterschiedlich große Arrays und unterschiedliche Verteilungen der Elemente als Tabellen und Diagramme dargestellt.

Um die Laufzeitkomplexitäten der Algorithmen besser erkennen zu können, sind die Diagramme in logarithmischer Skalierung. Dafür wird die Laufzeit (in ms) gegen die Anzahl der Elemente aufgetragen (s. Abb. 4 - 7). Alle Messwerte wurden für die letzten (bzw. besten) Versionen der Algorithmen entnommen.

Alle Messwerte wurden für die letzten (bzw. besten) Versionen der Algorithmen entnommen.

Die Laufzeiten der Algorithmen für zufällig gemischte Elemente sind in Tabelle 1 und Abb. 5 dargestellt.

Tabelle 1: Laufzeiten der verschiedenen Sortieralgorithmen für zufällig gemischte Elemente

	(a) 100.000 Elemente			(b) 1.000.000 Elemente			(c) 10.000.000 Elemente		
	Quick	Heap	Merge	Quick	Heap	Merge	Quick	Heap	Merge
1	10 ms	11 ms	11 ms	142 ms	156 ms	142 ms	1594 ms	2342 ms	1686 ms
2	11 ms	11 ms	11 ms	138 ms	150 ms	143 ms	1554 ms	2357 ms	1671 ms
3	10 ms	11 ms	11 ms	137 ms	150 ms	149 ms	1543 ms	2317 ms	1663 ms
4	10 ms	11 ms	11 ms	131 ms	154 ms	144 ms	1539 ms	2367 ms	1667 ms
5	10 ms	11 ms	11 ms	137 ms	151 ms	144 ms	1583 ms	2402 ms	1659 ms

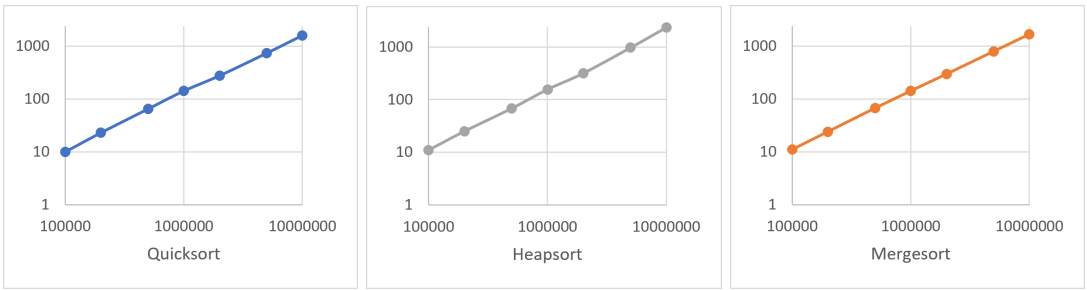


Abbildung 5: Diagramme der Messergebnisse aus Tab. 1, Laufzeiten in ms gegen Anzahl der Elemente (logarithmische Auftragung)

Für zufällig gemischte Elemente sind alle Algorithmen in etwa gleich schnell (Tab. 1).

Tabelle 2 und Abb. 6 betrachten die Laufzeiten der Algorithmen für sortierte Elemente.

Tabelle 2: Laufzeiten der verschiedenen Sortieralgorithmen für gleiche Elemente

	(a) 100.000 Elemente			(b) 1.000.000 Elemente			(c) 10.000.000 Elemente		
	Quick	Heap	Merge	Quick	Heap	Merge	Quick	Heap	Merge
1	0 ms	0 ms	6 ms	1 ms	5 ms	68 ms	11 ms	71 ms	864 ms
2	0 ms	0 ms	6 ms	1 ms	5 ms	79 ms	11 ms	71 ms	833 ms
3	0 ms	0 ms	6 ms	1 ms	5 ms	72 ms	11 ms	71 ms	830 ms
4	0 ms	0 ms	6 ms	1 ms	5 ms	71 ms	11 ms	70 ms	851 ms
5	0 ms	0 ms	6 ms	1 ms	5 ms	68 ms	11 ms	71 ms	857 ms

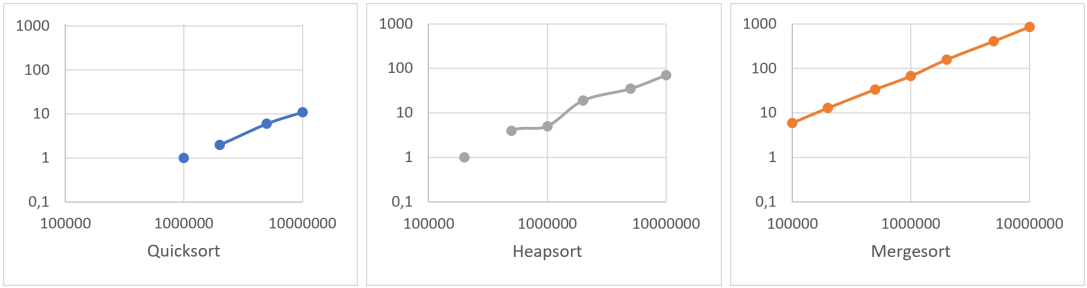


Abbildung 6: Diagramme der Messergebnisse aus Tab. 2, Laufzeiten in ms gegen Anzahl der Elemente (logarithmische Auftragung)

Für gleiche Elemente ist Quicksort deutlich schneller als die anderen beiden untersuchten Algorithmen, wobei Heapsort auch deutlich schneller ist als Mergesort.

Tabelle 3 und Abb. 7 zeigen die Laufzeiten der Algorithmen für sortierte Elemente. Die Ergebnisse für absteigend sortierte Elemente sind in Tabelle 4 und Abb. 8 dargestellt.

Tabelle 3: Laufzeiten der verschiedenen Sortieralgorithmen für sortierte Elemente

	(a) 100.000 Elemente			(b) 1.000.000 Elemente			(c) 10.000.000 Elemente		
	Quick	Heap	Merge	Quick	Heap	Merge	Quick	Heap	Merge
1	4 ms	9 ms	6 ms	37 ms	89 ms	68 ms	477 ms	985 ms	836 ms
2	3 ms	8 ms	6 ms	37 ms	86 ms	68 ms	478 ms	983 ms	830 ms
3	3 ms	8 ms	6 ms	38 ms	87 ms	72 ms	482 ms	989 ms	829 ms
4	3 ms	8 ms	6 ms	37 ms	87 ms	70 ms	479 ms	999 ms	831 ms
5	3 ms	7 ms	6 ms	38 ms	86 ms	71 ms	476 ms	984 ms	824 ms

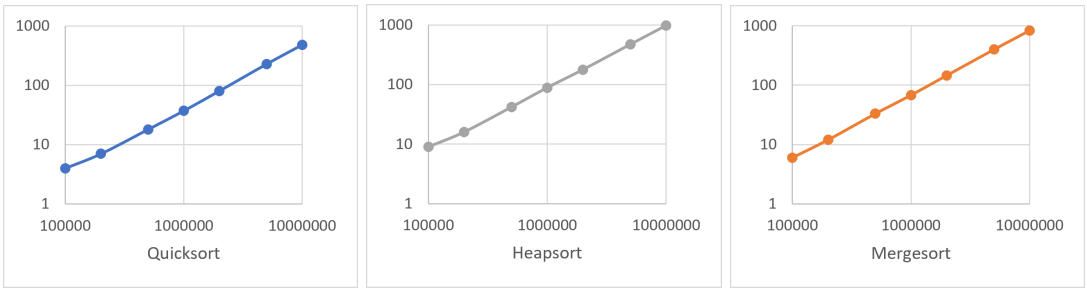


Abbildung 7: Diagramme der Messergebnisse aus Tab. 3, Laufzeiten in ms gegen Anzahl der Elemente (logarithmische Auftragung)

Tabelle 4: Laufzeiten der verschiedenen Sortieralgorithmen für absteigend sortierte Elemente

	(a) 100.000 Elemente			(b) 1.000.000 Elemente			(c) 10.000.000 Elemente		
	Quick	Heap	Merge	Quick	Heap	Merge	Quick	Heap	Merge
1	3 ms	9 ms	6 ms	37 ms	90 ms	68 ms	474 ms	1014 ms	859 ms
2	3 ms	8 ms	6 ms	37 ms	89 ms	71 ms	472 ms	1014 ms	845 ms
3	3 ms	8 ms	6 ms	37 ms	89 ms	71 ms	470 ms	1014 ms	844 ms
4	3 ms	8 ms	6 ms	38 ms	89 ms	71 ms	470 ms	1017 ms	857 ms
5	3 ms	8 ms	6 ms	37 ms	89 ms	70 ms	471 ms	1022 ms	864 ms

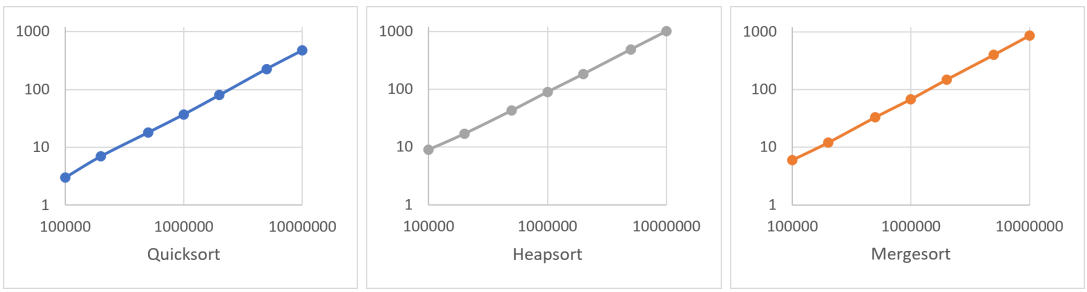


Abbildung 8: Diagramme der Messergebnisse aus Tab. 4, Laufzeiten in ms gegen Anzahl der Elemente (logarithmische Auftragung)

Sowohl bei sortierten als auch bei absteigend sortierten Elementen ist Quicksort deutlich schneller als die anderen Algorithmen, wobei diesmal Mergesort schneller ist als Heapsort. Allerdings sind die Unterschiede hier geringer als bei gleichen Elementen (Tab. 2, Abb. 6).

Im Folgenden ist die Entwicklung der Laufzeit für 1.000.000 Elemente im Verlauf der Entwicklung von Quicksort dargestellt. Die y-Achse entspricht dabei der Laufzeit in Millisekunden, die x-Achse dem Commit-Verlauf.

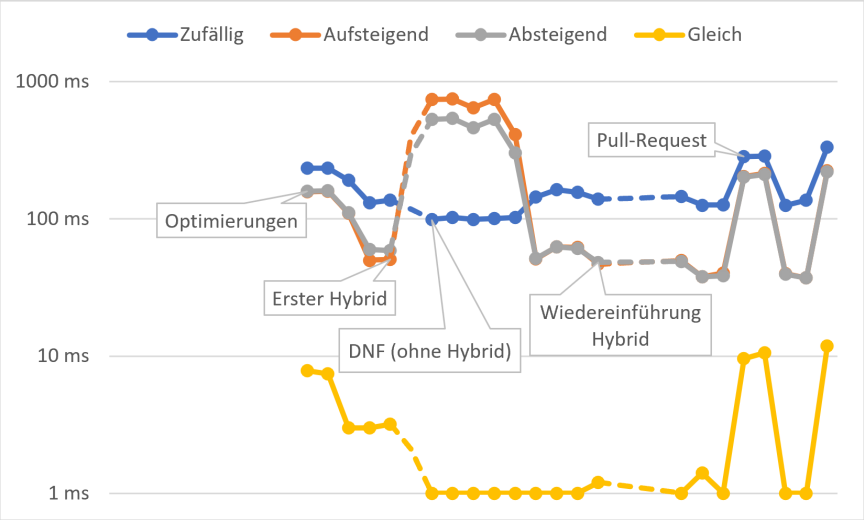


Abbildung 9: Benchmark für 1.000.000 Elemente in ms im Verlauf der Entwicklung.

3.3 Einfluss des Pull-Requests auf Leans „Radar“-Benchmark

Das in Abschnitt 3.1 beschriebene „Radar“-Benchmark wurde nach der Einreichung des Pull Requests automatisch ausgeführt, um den Einfluss der Änderungen auf die Menge an Instruktionen, Laufzeiten und Dateigrößen von Lean (im Vergleich mit der bestehenden offiziellen Version) zu messen.

Abb. 10 zeigt dessen Ergebnisse für die erste Version des Pull Requests.

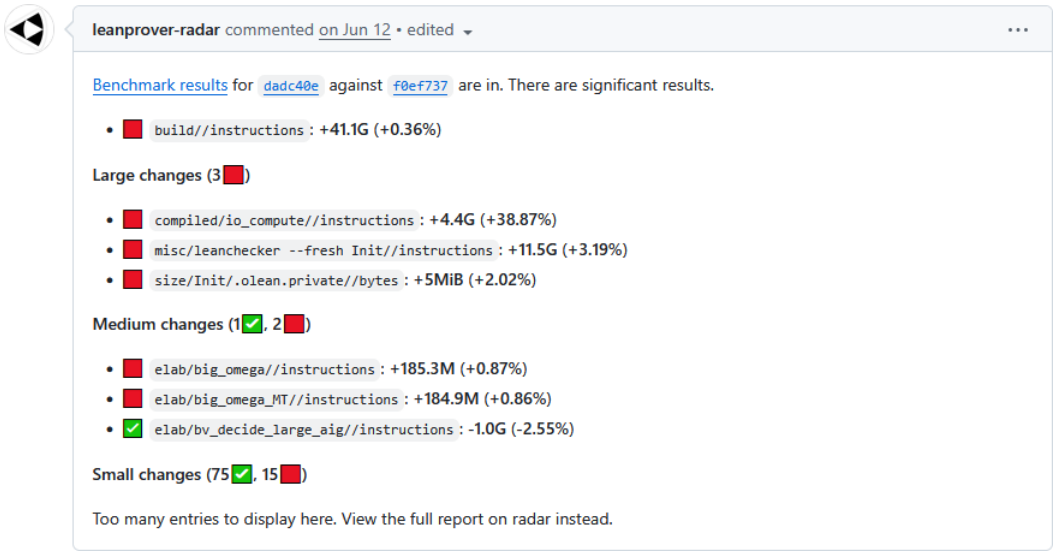


Abbildung 10: Ergebnisse des „Radar“-Benchmarks beim ersten Commit

Die Angaben des Radar-Benchmarks sind als Anzahl der Instruktionen sowohl absolut als auch relativ zu der vorherigen Version angegeben.

Es gibt insgesamt 96 bemerkte Veränderungen, von denen 76 eine Verbesserung darstellen. Von den 90 kleinen Veränderungen sind 75 Verbesserungen und 15 Verschlechterungen, von den 3 mittleren Veränderungen ist nur eine Verbesserung und von den 3 großen Veränderungen sind alle drei Verschlechterungen. Insbesondere die Anweisungen unter „compiled/io_compute//instructions“ zeigen eine relative Veränderung von +38,87 %, insgesamt 4,4 Milliarden Anweisungen mehr, was eine deutliche Verschlechterung ist.

Auf Hinweis[7] haben wir, um diese Verschlechterungen zu reduzieren, Attribute an einigen Funktionen ergänzt. Diese sollten dafür sorgen, dass die Optimierungen des Compilers besser greifen.

[7]Henrik Böving, Research Software Engineer bei der Lean FRO

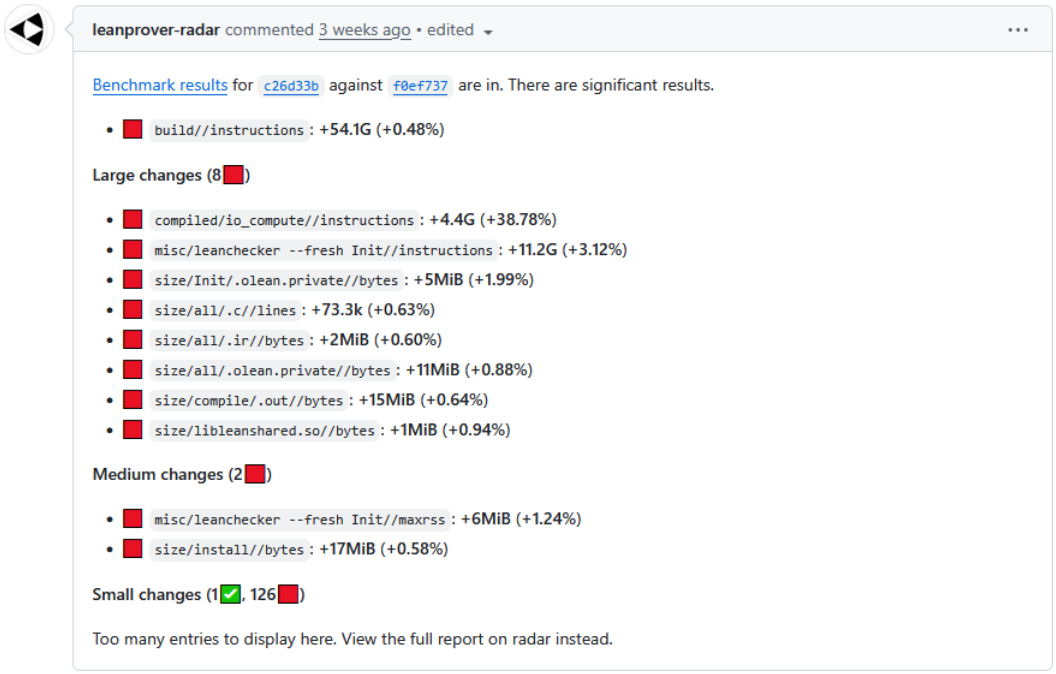


Abbildung 11: Ergebnisse des „Radar“-Benchmarks nach den ergänzten Attributen

Die Ergebnisse dieses Benchmarks zeigen, dass statt der ursprünglich 76 nur noch eine kleine Verbesserung und dafür 136 Verschlechterungen, von denen 8 groß und 2 mittel sind, vorliegen. Die schon erwähnte drastische Verschlechterung wurde nicht reduziert, die durchgeführten Veränderungen haben die angestrebten positiven Effekte nicht erbracht.

4 Diskussion

4.1 Bewertung der Ergebnisse

Mithilfe der Daten aus den Benchmarks (siehe Abschnitt 3.2) kann man die Auswirkungen der durchgeführten Anpassungen auf die Laufzeit analysieren.

Bei den Laufzeiten hat Quicksort oft einen deutlichen Vorteil gegenüber den anderen Algorithmen. Dies lässt sich unter anderem auf den höheren Optimierungsaufwand zurückführen, den wir in Quicksort investiert haben. Aufgrund seiner Struktur (siehe Abschnitt 2.1.3 und Abschnitt 2.2) bietet Quicksort vergleichsweise mehr Ansatzpunkte für Optimierungen als Mergesort oder Heapsort.

Mergesort Mergesort scheint in den meisten Fällen eine Laufzeitkomplexität von $\mathrm{O}(n \log n)$ zu zeigen, wobei die Konstante bei sortierten bzw. umgekehrt sortierten und gleichen Elementen deutlich kleiner ist als bei zufällig gemischten Elementen.

Mergesort ist ein sehr konsistenter Algorithmus, da seine Laufzeitkomplexität im Best-, Average- und Worst-Case bei $O(n \log n)$ liegt. Trotz jeglicher Optimierung bleibt Mergesort in jedem Fall bei $\mathrm{O}(n \log n)$. Durch den zusätzlichen Speicherbedarf und den damit verbundenen Kopiervorgängen ist die Laufzeit in der Praxis dennoch meist schlechter als bei anderen Algorithmen mit $\mathrm{O}(n \log n)$.

Die Verwendung des Bottom-up-Ping-Pong-Verfahrens zeigt sich als vorteilhaft, da mit der erhöhten Speichereffizienz und der *tail-recursion* die effektive Laufzeit kompetitiv zu den anderen Algorithmen ist.

Heapsort Die Ergebnisse bei Heapsort sind ähnlich wie bei Mergesort, wobei die Konstante bei zufällig gemischten und sortierten bzw. umgekehrt sortierten Elementen größer ist als bei Mergesort, was die Laufzeit verschlechtert. Das liegt vermutlich daran, dass der Aufbau des Heaps selbst Zeit, theoretisch O(n), aber praktisch vermutlich mehr, braucht. Dafür ist Heapsort bei gleichen Elementen deutlich schneller als Mergesort. Heapsort liegt im Bereich von $\mathrm{O}(n)$, während Mergesort bei $\mathrm{O}(n \log n)$ liegt.

Da keine größeren Optimierungen bei Heapsort vorgenommen wurden, ist die Performance vermutlich kaum auf die Implementierung selbst zurückzuführen. Der Algorithmus stimmt in den Ergebnissen aller Messungen mit der Theorie überein. Die geringe Laufzeit bei gleichen Elementen liegt daran, dass bei der Arbeit mit Heaps gleiche Zahlen bei der Rekursion nur einmal verglichen werden, wodurch die Laufzeitkomplexität auf $\mathrm{O}(n)$ sinkt. Die Konstante ist dabei höher als bei Quicksort.

Quicksort Quicksort ist in allen Messungen der schnellste Algorithmus, wobei die Unterschiede bei zufällig gemischten Elementen nicht so groß sind wie bei den anderen drei Fällen.

Während bei Mergesort und Heapsort die Laufzeitkomplexität im Worst-Case bei $\mathrm{O}(n \log n)$ liegt, kann Quicksort im Worst-Case auf $\mathrm{O}(n^2)$ sinken. Dafür hat Quicksort die besten Best-Case-Laufzeiten, bis zu einer Laufzeitkomplexität von $\mathrm{O}(n)$ bei gleichen Elementen.

Optimierungen bei Quicksort zielen meist darauf ab, den Worst-Case unwahrscheinlicher zu machen, wodurch der Durchschnittsfall besser wird. So ist Quicksort zwar im Durchschnitt schneller als

vergleichbare Algorithmen, kann aber im Einzelfall deutlich schlechter sein als die anderen Algorithmen.

Wie bereits erwähnt, ist die Laufzeitkomplexität von $O(n)$ für gleiche Elemente auf die Implementierung des *Dutch National Flag*-Algorithmus zurückzuführen, wodurch Quicksort in diesem Fall deutlich schneller ist als die anderen Algorithmen. Die Konstante ist dabei niedriger als die bei Heapsort.

Durch die Verwendung des *median of three* zur Auswahl des Pivots tritt der Worst-Case von $O(n^2)$ in der Praxis fast nie auf, wodurch die Laufzeitkomplexität bei zufällig gemischten und sortierten oder umgekehrt sortierten Elementen fast immer bei $O(n \log n)$ liegt. Die Konstante ist dabei niedriger als bei Mergesort und Heapsort.

Bei aufsteigend sortierten Elementen ist die Laufzeitkomplexität von $O(n \log n)$ zu einem großen Teil auf die Wahl des Pivot-Elements zurückzuführen. Hier hat der Insertionsort-Hybrid auch Auswirkungen, welcher eine Laufzeitkomplexität von $O(n)$ für sortierte Arrays hat. Während dieser bei großen Eingaben keine entscheidende Rolle spielt, hilft er bei der Vereinfachung der Rekursion. Bei umgekehrt sortierten Arrays hat der Insertionsort-Hybrid eine Laufzeitkomplexität von $O(n^2)$, aber die Vorteile bei der Rekursion gelten hier wie bei den anderen Fällen. Da Insertionsort nur bis zu einer Größe von 16 Elementen verwendet wird, wirkt sich die $O(n^2)$-Laufzeit bei großen Eingaben wie 1.000.000 Elementen nur geringfügig auf die Gesamtlaufzeit aus.

Der Umbau in eine Slice-basierte Implementierung hat die Speichereffizienz deutlich verbessert, wodurch die Laufzeit weiter sinkt.

So ist Quicksort im Durchschnitt am schnellsten, da der Worst-Case seltener und der Best-Case häufiger auftritt. Im Durchschnitt liegt die Laufzeitkomplexität bei $O(n \log n)$, wobei die Konstante niedriger ist als bei Mergesort und Heapsort. In besonderen Fällen, wie bei gleichen Elementen, liegt sie bei $O(n)$, wobei die Konstante auch niedriger ist als bei Heapsort. Bei der Aufteilung der Partitionen in eigene Arrays wäre der Overhead für die Verwaltung der Arrays zu groß, wodurch die Laufzeit deutlich schlechter wäre. Die Slice-basierte Implementierung hat sich im Verlauf der Arbeit als die beste herausgestellt.

Beim Einreichen des Pull-Requests ist eine deutliche Verschlechterung aufgetaucht. Diese tritt bei der Überbrückung von der Umsetzung mit LT zu der mit Ord auf, wie Tests ergeben haben. Das ist aber vorerst nötig, da andere Funktionen in den Lean-Bibliotheken die Funktion `Array.qsort` mit LT aufrufen, während die neue Implementierung Ord benötigt. Es gab auch schon eine Variante, die mit Ord arbeitet, aber diese war auch nur von der LT-Variante her überbrückt und wurde selten verwendet. Die genauen Ursachen dafür sind uns nicht bekannt, liegen aber vermutlich beim Compiler. Um diese Verschlechterung zu beheben, sollte eine zweite Version von Quicksort implementiert werden, die auf LT basiert.

4.2 Bedeutung der Ergebnisse im Hinblick auf Lean

Der Quicksort-Algorithmus wird primär von allen Nutzern verwendet, die ein Array sortieren wollen. Für diese ist die Laufzeit des Algorithmus entscheidend, da sie direkt die Laufzeit ihrer eigenen Programme bzw. Algorithmen beeinflusst.

Ansonsten wird er größtenteils in der Standardbibliothek von Funktionen verwendet, die Sortierungen durchführen. Ein veränderter Quicksort-Algorithmus hat demnach auf deren Effizienz einen direkten Einfluss.

Einige Funktionen, die nicht direkt mit Sortierungen zu tun haben, sind ebenfalls betroffen. Beide eingebauten Beweisassistenten `omega` und `grind` verwenden an einigen Stellen `qsort` als Schritt in einem Beweis. Die Effizienz dieser Beweisassistenten ist also von der Effizienz von `qsort` abhängig. Viele andere Funktionen benötigen sortierte Werte, manche nur für alphabetische oder sortierte Ausgaben, andere, weil deren Funktionalität auf sortierten Werten beruht, wie das bekannte Verfahren der binären Suche, die Elemente durch wiederholtes Halbieren eines sortierten Arrays findet.

Die aus den Radar-Benchmarks sichtbaren Ergebnisse zeigen, dass sich die Anzahl der Instruktionen bei der Einführung des neuen Quicksort-Algorithmus deutlich verändert hat. Der Algorithmus hat also einen messbaren Einfluss auf die Gesamteffizienz. Bei leanchecker handelt es sich um eine Messung, die das gesamte Modul `Init` testet. Es enthält sehr viele Funktionen, die auf sortierte Werte angewiesen sind, also ist es nicht verwunderlich, dass die Anzahl der Instruktionen sich bei der Einführung des neuen Quicksort-Algorithmus merklich erhöht hat.

Die Änderung bei „io_compute“ ist bedeutsamer. Dabei handelt es sich um einen Test, der die verschiedenen Schritte des Sortiervorgangs mit `qsort` misst. Die Erklärung dafür ist bislang nicht bekannt.

Die vorherige Version von Quicksort verwendete keinen *Dutch National Flag*-Algorithmus, wodurch bei Eingaben mit vielen gleichen Elementen eine Laufzeitkomplexität von $O(n^2)$ entstehen konnte. Daher musste für Eingaben mit vielen gleichen Elementen ein anderer Algorithmus, meist Mergesort, verwendet werden. Bei einer Eingabe von 1.000.000 Elementen entsprechen $O(n \log n)$ ungefähr $2 \cdot 10^7$ und $O(n^2)$ ungefähr $1 \cdot 10^{12}$ Berechnungsschritten. Damit liegt die Anzahl der Berechnungsschritte im quadratischen Fall ungefähr 50.000-mal höher, was sich in der Praxis deutlich auf die Laufzeit auswirken kann.

4.3 Verbesserungsmöglichkeiten

Die beabsichtigte formale Verifikation des vorgeschlagenen Algorithmus war leider zeitlich nicht mehr möglich. Zu zeigen ist insbesondere, dass eine Permutation des eingegebenen Arrays erzeugt wird, bei der gilt, dass jedes Element gleich wie oder größer als das vorherige ist.

Zudem bietet der von uns vorgeschlagene Algorithmus einige weitere Möglichkeiten zur Optimierung.

Ein Ansatzpunkt ist die gewählte Implementierung von Insertionsort. Durch die Verwendung von binärer Suche zum Finden der Stelle, an der ein Element eingefügt werden soll, könnten Vergleichsope-

rationen eingespart werden, was wahrscheinlich zu einer verbesserten Performance führt.

Eine weitere Optimierung betrifft den Rückgabetyp von `dnfstaged`. Dieser ist aktuell ein Subtyp eines Dreier-Tupels, welches das Array sowie die Partitionsgrenzen enthält. In Lean gibt es aber eigentlich keine Dreier-Tupel; tatsächlich lautet der Typ `Vector` α `size` $\times$ `(Nat` $\times$ `Nat)`. Es handelt sich also um zwei verschachtelte Tupel. Die Konsequenz hieraus ist, dass im Speicher zwei Tupel allokiert werden, wobei eines eine Referenz auf das andere enthält. Das ist ineffizient. Stattdessen könnte eine eigene Datenstruktur mit drei Feldern definiert werden.

Vor allem aber zeigen unsere Benchmarks, dass der vorgeschlagene Algorithmus deutlich langsamer ist, wenn er über die Wrapper aufgerufen wird, die auf die Funktion `ordOfLt` zurückgreifen, um eine `Ord`-Instanz zu erzeugen. Dieser Fall tritt in der Praxis sehr häufig ein. Eine mögliche Erklärung wäre, dass der Code mit `ordOfLt` ineffizient kompiliert wird. Daher ist zu prüfen, ob es nicht sinnvoller ist, den Code so umzuschreiben, dass er nativ mit einer `lt`-Funktion zum Vergleichen arbeitet. Bis diese Herausforderung nicht bewältigt ist, ist es nicht vertretbar, den hier vorgestellten Algorithmus in die Standardbibliothek von Lean aufzunehmen.

Dank

Wir danken Julia Markus Himmel und der Lean Focused Research Organization für die Ermöglichung und Durchführung des Projektes. Die breite Expertise und tiefen Einblicke, auch hinter die Kulissen von Lean, haben uns sehr bereichert.

Des Weiteren danken wir Anke Richert für die Betreuung von Seiten des Hector Seminars. Wie in unserer gesamten Hector-Zeit hat sie uns auch in diesem Projekt zu Höchstleistungen motiviert.

Dem Ehepaar Hector und der Hector Stiftung danken wir für die Ermöglichung des Hector-Seminars sowie allen Kursleitenden, die uns im Laufe der Jahre begleitet und gefördert haben.

Bei unseren Schulen bedanken wir uns dafür, dass sie unsere Teilnahme am Hector Seminar stets unterstützt haben.

Abschließend danken wir allen, die uns während des Projekts und unserer gesamten Zeit im Hector Seminar begleitet haben, sei es durch Feedback oder angeregte Fachdiskussionen am Abendessenstisch.

Selbständigkeitserklärung

Wir versichern, die Arbeit selbständig mit Unterstützung durch Julia Markus Himmel verfasst, alle benutzten Quellen und Hilfsmittel vollständig und genau angegeben und alles kenntlich gemacht zu haben, was aus Arbeiten anderer unverändert oder mit Abänderungen entnommen wurde.

Karlsruhe, der 20. September 2026

Johan Henkel, Elias Jungmann

5 Appendix

5.1 Listings

Listings zu 2.1.1 Mergesort

```
theorem mergeSortListPerm (xs : List α) :
  xs.Perm (mergeSortList xs isSmallerThan) := by
  induction h : xs.length using Nat.strongRecOn generalizing xs with
  | ind n ih =>
    match xs with
    | [] => simp[mergeSortList]
    | y :: ys =>
      simp[mergeSortList]
      by_cases hlen : ys.length = 0
      · simp[hlen, ← List.length_eq_zero_iff]
      · simp[hlen]
        let left  := List.take ((ys.length + 1) / 2) (y :: ys)
        let leftsorted := mergeSortList left isSmallerThan
        let right := List.drop ((ys.length + 1) / 2) (y :: ys)
        let rightsorted := mergeSortList right isSmallerThan
        let merged := leftsorted.merge rightsorted isSmallerThan
        have hsplit_perm : (y :: ys).Perm (left ++ right) := by
          rw [List.take_append_drop]
        have hleft_perm : left.Perm leftsorted :=
          ih left.length (by grind) left (by grind)
        have hright_perm : right.Perm rightsorted :=
          ih right.length (by grind) right (by grind)
        have hsorted_append : (left ++ right).Perm (leftsorted ++ rightsorted) :=
          hleft_perm.append hright_perm
        have hmerged : (left ++ right).Perm merged :=
          hsorted_append.trans (List.merge_perm_append isSmallerThan).symm
        exact hsplit_perm.trans hmerged
```

```
def myMerge {α} (le : α → α → Bool) (xs ys : Array α) (ix iy : Nat) (work : Array α)
  (hwork : work.size = xs.size + ys.size) (hix : ix ≤ xs.size) (hiy : iy ≤ ys.size) :
  { merged : Array α // merged.size = work.size } :=
  if hix : ix < xs.size then
    if hiy : iy < ys.size then
      if le xs[ix] ys[iy] then
        ⟨myMerge le xs ys (ix + 1) iy (work.set (ix+iy) xs[ix])
          (by grind) (by grind) (by grind), (by grind)⟩
      else
        ⟨myMerge le xs ys ix (iy + 1) (work.set (ix+iy) ys[iy])
          (by grind) (by grind) (by grind), (by grind)⟩
    else
      ⟨myMerge le xs ys (ix + 1) iy (work.set (ix+iy) xs[ix])
        (by grind) (by grind) (by grind), (by grind)⟩
  else
    if hiy : iy < ys.size then
      ⟨myMerge le xs ys ix (iy + 1) (work.set (ix+iy) ys[iy])
        (by grind) (by grind) (by grind), (by grind)⟩
    else
      ⟨work, rfl⟩

def myMergeSort (xs aux : Array α) (le : α → α → Bool) (haux : xs.size = aux.size) :
  { sorted : Array α // sorted.size = xs.size } :=
  if h : xs.size ≤ 1
  then ⟨xs, rfl⟩
  else
    let mid := xs.size / 2
    let left := myMergeSort (xs.take mid) (aux.take mid) le (by grind)
    let right := myMergeSort (xs.drop mid) (aux.drop mid) le (by grind)
    have hlr : aux.size = left.val.size + right.val.size := by grind
    ⟨myMerge le left.val right.val 0 0 aux hlr (by grind) (by grind), (by grind)⟩
```

```
--variation of myMerge w/o indexing proofs and on two full arrays, replacing xs and ys by
    slicing indices
def myMerge_sliced {α} [Inhabited α] (le : α → α → Bool) (inp : Array α) (start mid last
    ix iy : Nat) (work : Array α) : Array α :=
  --xs finished? (ix < len xs)
  if ix < (mid - start) then
  --ys finished? (iy < len ys)
    if iy < (last - mid) then
      if le inp[start+ix]! inp[mid+iy]!
      --if xs[ix] is smaller, it is written to the output at the correct location
        then myMerge_sliced le inp start mid last (ix + 1) iy (work.set! (ix+iy+start)
    inp[start+ix]!)
      --else, it's ys[iy]
        else myMerge_sliced le inp start mid last ix (iy + 1) (work.set! (ix+iy+start)
    inp[mid+iy]!)
    else
    --if ys is done, but not xs, the next element from xs is copied
      myMerge_sliced le inp start mid last (ix + 1) iy (work.set! (ix+iy+start)
    inp[start+ix]!)
  else
  --ys finished?
    if iy < (last - mid) then
    --if xs is done, but not ys, the next element from ys is copied
      myMerge_sliced le inp start mid last ix (iy + 1) (work.set! (ix+iy+start)
    inp[mid+iy]!)
    else
    --if both slices are done, return
      work

def copyFinalSlice (inp out : Array α) (start : Nat) : Array α :=
  if h : start < inp.size then --arrived at end?
  --copy current value to same position in out, increase current index
    copyFinalSlice inp (out.set! start inp[start]) (start + 1)
  else
    out --if there is nothing left to copy, return
```

```
partial def mMS2_helper [Inhabited α] (le : α → α → Bool) (arr1 arr2 : Array α) (i width
    : Nat) : Array α :=
  --outer loop: runs until the whole list has been merged over
    if width < arr1.size then
    --inner loop: goes through all the couples to merge, then increases i (the start
    marker) by the range that was just processed
      if i < arr1.size then
      --only merge if there is a counterpart to merge with
        if i + width < arr1.size then
        --if the counterpart is shorter, no problem, just use it
          if i + 2 * width > arr1.size then
          --special case: counterpart is shorter
            mMS2_helper le arr1 (myMerge_sliced le arr1 i (i+width) arr1.size 0 0 arr2)
    (i+2*width) width
          else
          --normal case
            mMS2_helper le arr1 (myMerge_sliced le arr1 i (i+width) (i+2*width) 0 0 arr2)
    (i+2*width) width
        else
        --special case: no counterpart => the already sorted portion is carried over
          mMS2_helper le arr1 (copyFinalSlice arr1 arr2 i) (i+2*width) width
      else
      --if all couples have been merged, switch arrays and merge those with their new
    counterparts
        mMS2_helper le arr2 arr1 0 (2*width)
    else
    --finished
      arr1

def myMergeSort2 [Inhabited α] (xs : Array α) (le : α → α → Bool) : Array α :=
  mMS2_helper le xs xs 0 1 --copy xs one time as an auxiliary array, initialise i and width
```

Listing zu 2.1.2 Heapsort

```
@[inline]
def par := fun i => (i-1) / 2 --calculate relatives to a node by using its index
@[inline]
def lCh := (· * 2 + 1)
@[inline]
def rCh := (· * 2 + 2)

@[specialize lt]
partial def siftDown [Inhabited α] (lt : α → α → Bool) (heap : Array α) (i heapsize :
    Nat): Array α := --starts a a certain node and works its way down, resoring the
    max-heap property
  if (lCh i < heapsize) ∧ (lt (heap[i]'sorry) (heap[lCh i]'sorry)) then --left child?
    if (rCh i < heapsize) ∧ (lt (heap[i]'sorry) (heap[rCh i]'sorry)) then --right child?
      if lt (heap[lCh i]'sorry) (heap[rCh i]'sorry) then --both exist and don't fit =>
    which is larger?
        siftDown lt (heap.swap (rCh i) i sorry sorry) (rCh i) heapsize --right one is
    largest
      else
        siftDown lt (heap.swap (lCh i) i sorry sorry) (lCh i) heapsize --left one is largest
    else
      siftDown lt (heap.swap (lCh i) i sorry sorry) (lCh i) heapsize --right child does not
    exist, but left child does not fit
  else
    if (rCh i < heapsize) ∧ (lt (heap[i]'sorry) (heap[rCh i]'sorry)) then --right child?
      siftDown lt (heap.swap (rCh i) i sorry sorry) (rCh i) heapsize ----left child does
    not exist, but right child does not fit
    else
      heap --no children or both fit

@[specialize lt]
def cH_helper [Inhabited α] (lt : α → α → Bool) (xs : Array α) (i : Nat): Array α :=
    --works its way up the heap restoring max-heap property downwards
  if i = 0 then --finished?
    siftDown lt xs 0 xs.size --one last time
  else
    cH_helper lt (siftDown lt xs i xs.size) (i-1) --do it for every node

@[inline]
def createHeap [Inhabited α] (lt : α → α → Bool) (xs : Array α) : Array α := --calls
    helper to build heap
  cH_helper lt xs (xs.size/2) --can start at half, because the other half is leaves
```

```
@[specialize lt]
def mHS_helper [Inhabited α] (lt : α → α → Bool) (heap : Array α) (heapsize : Nat):
    Array α := --sorts a heap by extracting the max element and moving it to the end of the
    array
  if heapsize = 0 then heap --finished => return
  else
    mHS_helper lt (siftDown lt (heap.swap 0 (heapsize - 1) sorry sorry) 0 (heapsize-1))
    (heapsize - 1) --swap with last element of heap, make the heap one element samller and
    restore max-heap property

@[inline]
def myHeapSort [Inhabited α] (xs : Array α) (lt : α → α → Bool): Array α := --creates
    heap and calls helper to sort
  mHS_helper lt (createHeap lt xs) xs.size
```

Listing zu 2.1.3 Quicksort

```
-- Todo: differentiate tasks between unsorted numbers and finished numbers
inductive Todo (α : Type)
| SortArr : Array α → Todo α
| PushVal : Nat → α → Todo α

/- functions -/
-- pivotselect: select a pivot using the median of start, middle, and end of the array
-- pivotsplit: divide array into smaller than, equal to, or greater than a pivot
def pivotselect [Ord α] (arr : Array α) (h : arr.size > 0) : α :=
  let size := arr.size
  let half := size/2
  let p1 := arr[0]
  let p2 := arr[half]
  let p3 := arr[size - 1]
  let le := fun a b => compare a b != .gt
  if le p1 p2 then
    if le p2 p3 then p2
    else if le p1 p3 then p3
    else p1
  else
    if le p1 p3 then p1
    else if le p2 p3 then p3
    else p2

def pivotsplitHelper [Ord α] (i eq : Nat) (arr lt gt : Array α) (pvt : α) : Array α × Nat
    × Array α :=
  if h : i >= arr.size then
    (lt, eq, gt)
  else
    let x := arr[i]
    match compare x pvt with
    | .lt => pivotsplitHelper (i+1) eq arr (lt.push x) gt pvt
    | .eq => pivotsplitHelper (i+1) (eq+1) arr lt gt pvt
    | .gt => pivotsplitHelper (i+1) eq arr lt (gt.push x) pvt
termination_by arr.size - i

/- wrappers -/
def pivotsplit [Ord α] (arr : Array α) (pvt : α): Array α × Nat × Array α :=
  pivotsplitHelper 0 0 arr #[] #[] pvt
```

```
/- main algorithm + wrapper -/
-- quicksort: uses a Todo list and pivotsplit/select to progress through list and sort an
    array
partial def quicksortHelper [Ord α] (todos : List (Todo α)) (acc : Array α) : Array α :=
  match todos with
  | [] => acc
  | inst :: rest =>
    match inst with
    | Todo.PushVal n x => quicksortHelper rest (acc.append (Array.replicate n x))
    | Todo.SortArr arr =>
      if h : arr.size < 28 then
        let lt := fun a b => compare a b == .lt
        quicksortHelper rest (acc.append (arr.insertionSort lt))
      else
        let pvt := pivotselect arr (by omega)
        let (lt, eq, gt) := pivotsplit arr pvt
        let new := Todo.SortArr lt :: Todo.PushVal eq pvt :: Todo.SortArr gt :: rest
        quicksortHelper new acc

def quicksort [Ord α] (arr : Array α) :=
  let acc : Array α := (Array.mkEmpty arr.size)
  quicksortHelper [Todo.SortArr arr] acc
```

Listing zu 2.2 Pull-Request: Quicksort mit *Dutch National Flag*-Algorithmus

lean4/src/Init/Data/Array/QSort/Basic.lean

```
/-
Copyright (c) 2019 Microsoft Corporation. All rights reserved.
Released under Apache 2.0 license as described in the file LICENSE.
Authors: Elias Jungmann and Johan Henkel
-/
module

prelude
public import Init.Data.Vector.Basic
public import Init.Data.Ord.Basic
import Init.Omega
public section
set_option linter.listVariables true -- Enforce naming conventions for
    `List`/`Array`/`Vector` variables.
-- We do not enable `linter.indexVariables` because it is helpful to name index variables
    `sllo`, `slhi`, etc.


/-
  Simple implementation of insertion sort as a cutoff for small slices
-/
def insertionSort [Ord α]
  (xs : Vector α size) (sllo slhi i : Nat)
  (hslloslhi : sllo < slhi) (hslhi : slhi ≤ size)
  (hslloi : sllo ≤ i) (hslhii : i ≤ slhi)
  : Vector α size :=

  if hfin : i = slhi then xs else

  let rec movedown [Ord α] (xs : Vector α size) (j : Nat) (hjsllo : sllo ≤ j) (hjslhi : j
    < slhi) : Vector α size :=

    if hfin : j = sllo then xs else

    if compare (xs[j]) (xs[j - 1]) = .lt then
      movedown (xs.swap j (j - 1)) (j - 1) (by omega) (by omega)
    else xs

  insertionSort (movedown xs i (by omega) (by omega)) sllo slhi (i + 1) (by omega) (by
    omega) (by omega) (by omega)
```

```
/-
  Selects the median of three in the given slice [sllo...slhi)
-/
def pivotselect [Ord α]
  (xs : Vector α size) (sllo slhi : Nat)
  (hslhi : slhi ≤ size) (hslloslhi : sllo < slhi)
  : {idx : Nat // sllo ≤ idx ∧ idx < slhi} :=

  let p1 := xs[sllo]
  let p2 := xs[sllo + (slhi - sllo)/2]
  let p3 := xs[slhi - 1]

  let le := fun a b => compare a b != .gt

  if le p1 p2 then
    if le p2 p3 then ⟨sllo + (slhi - sllo)/2, (by omega)⟩
    else if le p1 p3 then ⟨slhi - 1, (by omega)⟩
    else ⟨sllo, (by omega)⟩

  else
    if le p1 p3 then ⟨sllo, (by omega)⟩
    else if le p2 p3 then ⟨slhi - 1, (by omega)⟩
    else ⟨sllo + (slhi - sllo)/2, (by omega)⟩
```

```
/-Second stage of dnf algorithm
    pivot is in the eq-partition, therefore the other two slices that will need further
    sorting must be smaller than the original one -/
def dnfstage2 [Ord α]
  (xs : Vector α size) (pvt eq unproc fin_unproc sllo slhi : Nat)
  (heq_unproc : eq < unproc) (hunproc_fin_unproc : unproc ≤ fin_unproc) (hfin_unproc :
    fin_unproc < size)
  (hsllo : sllo ≤ eq) (hfin_unproc_slhi : fin_unproc < slhi) (hlohi : slhi - sllo > 1)
    (hslhi : slhi ≤ size)
  (hpvt_sllo : sllo ≤ pvt) (hpvt_slhi : pvt < slhi) (hpvt_eq : eq ≤ pvt) (hpvt_fin_unproc
    : pvt ≤ fin_unproc)
  : {r : (Vector α size × Nat × Nat) // r.snd.snd ≤ slhi ∧ sllo ≤ r.snd.fst ∧ r.snd.fst
    ≤ size ∧ r.snd.fst < r.snd.snd} :=

  match compare xs[unproc] xs[pvt] with
  | .lt =>
    if hfin : unproc ≥ fin_unproc then ⟨((xs.swap unproc eq), eq + 1, fin_unproc + 1), by
    simp; omega⟩ else
    if hpvt : eq = pvt then dnfstage2 (xs.swap unproc eq) unproc (eq + 1) (unproc + 1)
    fin_unproc sllo slhi (by omega) (by omega) (by omega) (by omega) (by omega) (by omega)
    (by omega) (by omega) (by omega) (by omega) (by omega) else
    if compare xs[eq] xs[unproc] = .lt then dnfstage2 xs pvt (eq + 1) (unproc + 1)
    fin_unproc sllo slhi (by omega) (by omega) (by omega) (by omega) (by omega) (by omega)
    (by omega) (by omega) (by omega) (by omega) (by omega) else
    dnfstage2 (xs.swap unproc eq) pvt (eq + 1) (unproc + 1) fin_unproc sllo slhi (by omega)
    (by omega) (by omega) (by omega) (by omega) (by omega) (by omega) (by omega) (by omega)
    (by omega) (by omega)

  | .gt =>
    if hfin : unproc ≥ fin_unproc then ⟨(xs, eq, fin_unproc), by simp; omega⟩ else
    if hpvt : fin_unproc = pvt then dnfstage2 (xs.swap unproc fin_unproc) unproc eq unproc
    (fin_unproc - 1) sllo slhi (by omega) (by omega) (by omega) (by omega) (by omega) (by
    omega) (by omega) (by omega) (by omega) (by omega) (by omega) else
    if compare xs[fin_unproc] xs[unproc] = .gt then dnfstage2 xs pvt eq unproc (fin_unproc
    - 1) sllo slhi (by omega) (by omega) (by omega) (by omega) (by omega) (by omega) (by
    omega) (by omega) (by omega) (by omega) (by omega) (by omega)
    else dnfstage2 (xs.swap unproc fin_unproc) pvt eq unproc (fin_unproc - 1) sllo slhi (by
    omega) (by omega) (by omega) (by omega) (by omega) (by omega) (by omega) (by omega) (by
    omega) (by omega) (by omega)

  | .eq =>
    if hfin : unproc ≥ fin_unproc then ⟨(xs, eq, fin_unproc + 1), by simp; omega⟩ else
    dnfstage2 xs pvt eq (unproc + 1) fin_unproc sllo slhi (by omega) (by omega) (by omega)
    (by omega) (by omega) (by omega) (by omega) (by omega) (by omega) (by omega) (by omega)
```

```
/-
  First stage of dnf algorithm: partitions until the pivot is in the eq-partition
    During both stages, the following partitioning scheme applies:
      [sllo...eq) - elements smaller than the pivot
      [eq...unproc) - elements that are equal to the pivot
      [unproc...fin_unproc] - elements that have not been processed yet
      [fin_unproc + 1...slhi) - elements that are greater than the pivot -/
def dnfstage1 [Ord α]
  (xs : Vector α size) (pvt eq unproc fin_unproc sllo slhi : Nat)
  (heq_unproc : eq ≤ unproc) (hunproc_fin_unproc : unproc ≤ fin_unproc) (hfin_unproc :
    fin_unproc < size)
  (hsllo : sllo ≤ eq) (hfin_unproc_slhi : fin_unproc < slhi) (hlohi : slhi - sllo > 1)
    (hslhi : slhi ≤ size)
  (hpvt_sllo : sllo ≤ pvt) (hpvt_slhi : pvt < slhi) (hpvt_unproc : unproc ≤ pvt)
    (hpvt_fin_unproc : pvt ≤ fin_unproc)
  : {r : (Vector α size × Nat × Nat) // r.snd.snd ≤ slhi ∧ sllo ≤ r.snd.fst ∧ r.snd.fst
    ≤ size ∧ r.snd.fst < r.snd.snd} :=

  if hpvt : unproc = pvt then
    if hfin : fin_unproc = unproc then ⟨(xs, eq, fin_unproc + 1), by simp; omega⟩
    else dnfstage2 xs pvt eq (unproc + 1) fin_unproc sllo slhi (by omega) (by omega) (by
    omega) (by omega) (by omega) (by omega) (by omega) (by omega) (by omega) (by
    omega)

  else match compare xs[unproc] xs[pvt] with
  | .lt =>
    dnfstage1 (xs.swap eq unproc) pvt (eq+1) (unproc + 1) fin_unproc sllo slhi (by omega)
    (by omega) (by omega) (by omega) (by omega) (by omega) (by omega) (by omega)
    (by omega) (by omega)

  | .gt =>
    if hfin : fin_unproc = unproc then ⟨(xs, eq, fin_unproc), by simp; omega⟩ else
    if hfin2 : fin_unproc - unproc = 1 then ⟨(xs.swap unproc fin_unproc, eq, fin_unproc),
    by simp; omega⟩ else
    if hpvt2 : pvt = fin_unproc then dnfstage2 (xs.swap unproc pvt) unproc eq (unproc + 1)
    (fin_unproc - 1) sllo slhi (by omega) (by omega) (by omega) (by omega) (by omega) (by
    omega) (by omega) (by omega) (by omega) (by omega) (by omega) else
    dnfstage1 (xs.swap unproc fin_unproc) pvt eq unproc (fin_unproc - 1) sllo slhi (by
    omega) (by omega) (by omega) (by omega) (by omega) (by omega) (by omega) (by omega) (by
    omega) (by omega) (by omega)

  | .eq =>
    dnfstage2 xs pvt eq (unproc + 1) fin_unproc sllo slhi (by omega) (by omega) (by omega)
    (by omega) (by omega) (by omega) (by omega) (by omega) (by omega) (by omega) (by omega)
```

```
/-
  Wrapper for the two-staged Dutch National Flag algorithm
    returns (xs, mid, hi) and some proofs so that xs is partitioned as follows:
      [sllo...mid) --- elements smaller than the pivot
      [mid...hi) -- elements equal to the pivot
      [hi...slhi) - elemts that are greater than the pivot
-/
@[inline]
def dnfstaged [Ord α]
  (xs : Vector α size) (pvt : Nat) (sllo slhi : Nat)
  (hlohi : slhi - sllo > 1) (hhi : slhi ≤ size)
  (hpvt : sllo ≤ pvt ∧ pvt < slhi)
  : {r : (Vector α size × Nat × Nat) // r.snd.snd ≤ slhi ∧ sllo ≤ r.snd.fst ∧ r.snd.fst
    ≤ size ∧ r.snd.fst < r.snd.snd} :=

  dnfstage1 xs pvt sllo sllo (slhi - 1) sllo slhi (by omega) (by omega) (by omega) (by
    omega) (by omega) (by omega) (by omega) (by omega) (by omega) (by omega)

/-
  Main algorithm
-/
def quicksorthelper [Ord α]
  (xs : Vector α  size) (sllo slhi : Nat)
  (hslloslhi : sllo ≤ slhi) (hslhi : slhi ≤ size)
  : Vector α size :=

  if hfin : slhi - sllo ≤ 1 then xs else

  if slhi - sllo ≤ 16 then insertionSort xs sllo slhi (sllo + 1) (by omega) (by omega) (by
    omega) (by omega) else

  let pvt := pivotselect xs sllo slhi (by omega) (by omega)
  let ⟨(ys, mid, hi), ⟨h1, h2, h3⟩⟩ := (dnfstaged xs pvt sllo slhi (by omega) (by omega) (by
    omega))


  have hterm : slhi - hi < slhi - sllo := by simp only [] at h1 h2 h3; omega
  let ys' := quicksorthelper ys hi slhi (by omega) (by omega)

  have hterm2 : mid - sllo < slhi - sllo := by simp only [] at h1 h2 h3; omega
  quicksorthelper ys' sllo mid (by omega) (by omega)

termination_by slhi - sllo
```

```
/-
  Uses a lt function to create a Ord instance for the actual algorithm
    Necessary for compatibility reasons
-/
@[inline, reducible]
def ordOfLt (lt : α → α → Bool) : Ord α where
  compare a b :=
    if lt a b then .lt
    else if lt b a then .gt
    else .eq

/-
  Wrappers for qsort to ensure compatibility to previous code
-/
def Array.qsort (xs : Array α) (lt : α → α → Bool := by exact (· < ·)) (sllo := 0) (slhi
    := xs.size) : Array α :=
  letI : Ord α := ordOfLt lt
  let slhi := min slhi xs.size
  let sllo := min sllo slhi
  (quicksorthelper xs.toVector sllo slhi (by omega) (by omega)).toArray

def Vector.qsort {size} (xs : Vector α size) (lt : α → α → Bool := by exact (· < ·))
    (sllo := 0) (slhi := size) : Vector α size :=
  letI : Ord α := ordOfLt lt
  let slhi := min slhi size
  let sllo := min sllo slhi
  quicksorthelper xs sllo slhi (by omega) (by omega)

def Array.qsortOrd [Ord α] (xs : Array α) (sllo := 0) (slhi := xs.size) : Array α :=
  let slhi := min slhi xs.size
  let sllo := min sllo slhi
  (quicksorthelper xs.toVector sllo slhi (by omega) (by omega)).toArray

def Vector.qsortOrd [Ord α] {size} (xs : Vector α size) (sllo := 0) (slhi := size) :
    Vector α size :=
  let slhi := min slhi size
  let sllo := min sllo slhi
  quicksorthelper xs sllo slhi (by omega) (by omega)
```

Literatur

[1] en:User:RolandH. Sorting quicksort anim. https://commons.wikimedia.org/wiki/File:Sorting_quicksort_anim.gif, 2011. CC BY-SA 3.0 (http://creativecommons.org/licenses/by-sa/3.0/), via Wikimedia Commons, Zuletzt abgerufen am 08.07.2026.

[2] Lean FRO. Lean in Action. https://lean-lang.org/, 2026. Zuletzt abgerufen am 27.06.2026.

[3] D. T. Christiansen. Functional Programming in Lean. https://lean-lang.org/functional_programming_in_lean/Introduction/#Functional-Programming-in-Lean--Introduction, 2025. Zuletzt abgerufen am 27.06.2026.

[4] Haskell.org, Inc. Haskell Language. https://www.haskell.org/, 2026. Zuletzt abgerufen am 14.07.2026.

[5] D. T. Christiansen. Functional Programming in Lean. https://lean-lang.org/functional_programming_in_lean/Programming-with-Dependent-Types/#Functional-Programming-in-Lean--Programming-with-Dependent-Types, 2025. Zuletzt abgerufen am 04.07.2026.

[6] J. Avigad et al. Theorem Proving in Lean 4. https://lean-lang.org/theorem_proving_in_lean4/Propositions-and-Proofs/#propositions-and-proofs, 2025. Zuletzt abgerufen am 05.07.2026.

[7] Derrick Coetzee. Singly linked list. https://commons.wikimedia.org/wiki/File:Singly_linked_list.png, 2005. Public Domain, via Wikimedia Commons, Zuletzt abgerufen am 14.07.2026.

[8] Lukas Bergdoll @Voultapher and Orson Peters @orlp. driftsort: an efficient, generic and robust stable sort implementation. https://github.com/Voultapher/sort-research-rs/blob/main/writeup/driftsort_introduction/text.md, 2024. Zuletzt abgerufen am 15.07.2026.

[9] Edsger Wybe Dijkstra. *A Discipline of Programming.* Prentice-Hall, 1976. S. 111-114.