Universität Stuttgart

Institut für Formale Methoden der Informatik

Universitätsstraße 38
70569 Stuttgart

Bachelorarbeit

Analyse von Pivotwahlstrategien bei Quicksort

Maximilian Frech

Studiengang:	Softwaretechnik
1. Prüfer:	Dr. Armin Weiß
2. Prüfer:	
Betreuer:	Dr. Armin Weiß
begonnen am:	05.07.2021
beendet am:	05.01.2022

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einleitung

Sortieren ist ein grundlegendes Problem der Informatik und es existieren viele bekannte Sortieralgorithmen. Der wohl meistgenutzte ist Quicksort, ein rekursiver in-place Algorithmus, welcher nach dem „Divide and Conquer“ Prinzip arbeitet.

Quicksort wurde 1960 von Tony Hoare entwickelt und wird seitdem stetig verbessert. Die Eingabe des Algorithmus wird durch die Wahl eines Pivotelements aufgeteilt. Hierbei werden alle Elemente, die kleiner als das Pivotelement sind, links davon platziert und alle Elemente die größer sind rechts. Nun wird der Algorithmus erneut aufgerufen, einmal mit dem linken Teil der Eingabe und einmal mit dem rechten Teil. Dies geschieht rekursiv, bis die Eingabe sortiert ist.

Ziel dieser Arbeit ist es, gängige Methoden zur Wahl des Pivotelements herauszuarbeiten und zu vergleichen. Dabei soll auch überprüft werden, ob es Sinn macht, das Verfahren je nach Eingabegröße zu wechseln.

Zusätzlich werden die Auswirkungen von Randomisierung in der Pivotwahl untersucht. Außerdem sollen verschiedene Inputarten wie Vorsortierte Listen oder speziell entwickelte Worst Case Szenarien getestet werden. Untersucht werden, die Auswirkungen der verschiedenen Methoden im Bezug auf: Laufzeit, Anzahl der Vergleiche und Anzahl der Zuweisungen.

Die Ergebnisse werden statisch ausgewertet und dargestellt, um die Methoden hinsichtlich der drei Kriterien zu bewerten.

2 Funktionsweise Quicksort

Quicksort ist ein nicht stabiler Sortieralgorithmus. Dies bedeutet, die Reihenfolge von Elementen, mit dem gleichen Wert in der Eingabe, ist nicht sichergestellt in der sortierten Liste. Die Eingabe wird durch die Wahl eines Pivotelements aufgeteilt. Nun wird der Algorithmus rekursiv aufgerufen, einmal mit dem linken Teil der Eingabe und einmal mit dem rechten Teil. Dies geschieht rekursiv, bis die Eingabe sortiert ist.

Algorithm 1: Quicksort

```
1 function quicksort(array, first, last)
2     if first ≤ last then
3         return
4     end
5     pivotindex ← partition()
6
7     quicksort(array, first, pivotindex − 1)
8     quicksort(array, pivotindex + 1, last)
9 end
```

Damit Quicksort korrekt arbeitet, müssen alle Elemente die kleiner als das Pivotelement sind links und alle die größer sind rechts davon eingeordnet werden. Dieser Vorgang wird Partitionierung genannt.

2.1 Partitionierung

Für die Partitionierung existieren zwei gängige Versionen. Das Hoare- und Lomuto Partitionsschema.

2.1.1 Das Hoare Partitionsschema

Das Hoare Partitionsschema ist das urpsrünglich entworfene Partitionsschema, welches Tony Hoare mit der Erfindung von Quicksort etablierte.

Hierbei werden zwei Zeiger benutzt, welche an beiden Enden der Eingabe initalisiert werden. Der linke Zeiger wird solange inkrementiert, bis ein Element gefunden wird, welches größer oder gleich wie das Pivotelement ist. Wenn ein solches Element gefunden wurde, wird der rechte Zeiger dekrementiert, bis ein Element gefunden wird, das kleiner als das Pivotelement ist. Diese beiden Elemente werden dann miteinander getauscht.

Dies wiederholt sich solange, bis sich die zwei Zeiger treffen. Die Eingabe ist somit richtig partitioniert, das Pivotelement wird am Kollisionspunkt eingeordnet und dessen Index

wird zurückgegeben.

Algorithm 2: Hoare Partitionsschema

```
1  function partitioning(array, first, last) : int
2      pivot ← array[first]
3      i ← first − 1
4      j ← last + 1
5      while true do
6          do
7              i ← i + 1
8          while array[i] < pivot
9          do
10             j ← j − 1
11         while array[i] > pivot
12         if i ≥ j then
13             swap(array[j], array[first])
14             return j
15         end
16         swap(array[i − 1], array[first])
17     end
18 end
```

2.1.2 Vergleichsoptimiertes Hoare Partitionsschema

Das Hoare Partitionsschema benötigt in der grundlegenden Variante, bei Eingabegröße n, mehr als $n - 1$ Vergleiche um alle Elemente einzuornden. Deshalb wurden einige Modifikationen vorgenommen. Die Folgende Implementierung des Partitionsschemas benötigt nur noch $n - 1$ Vergleiche für die Partitionierung.

Algorithm 3: Hoare Partitionsschema Optimiert

```
1  function partitioning(array, first, last) : int
2      pivot ← array[first]
3      i ← first
4      j ← last + 1
5      while true do
6          do
7              i ← i + 1
8          while i ≠ j − 1 && array[i] < pivot
9          if i == j − 1 then
10             if array[j - 1] < pivot then
11                 swap(array[j − 1], array[first])
12                 return j − 1
13             else
14                 swap(array[j − 2], array[first])
15                 return j − 2
16             end
17         end
18         do
19             j ← j − 1
20         while i ≠ j − 1 && array[i] ≥ pivot
21         if j == i + 1 then
22             if array[i + 1] < pivot then
23                 swap(array[i + 1], array[i])
24                 swap(array[i], array[first])
25                 return i
26             else
27                 swap(array[i − 1], array[first])
28                 return i − 1
29             end
30         end
31         swap(array[i], array[j])
32     end
33 end
```

2.1.3 Das Lomuto Partitionsschema

Das Lomuto Partitionsschema wurde von Nico Lomuto erfunden, erlangte aber erst durch die Veröffentlichung im Buch „Programming Pearls“, von John Bentley Bekanntheit. Es wird in der Praxis oft benutzt, um Quicksort zu erklären, da der Code deutlich kürzer und besser verständlich ist.

Standardmäßig wird als Pivotelement das letztes Element des Arrays gewählt. Es wird ein Zeiger i vor das erste Element initalisiert und dann mit j durch die Eingabe iteriert. Das Element an der Stelle j wird mit dem Pivotelement verglichen. Wenn es größer als das Pivotelement ist, passiert nichts und j läuft weiter. Falls es jedoch kleiner als das Pivotelement ist, wird Zeiger i inkrementiert und die Elemente an Stelle i und j getauscht.

Wenn j durch die ganze Eingabe iteriert ist, steht der Zeiger i also auf dem letzten Element, welches kleiner als das Pivotelement ist. Nun wird das Pivotelement an die Stelle $i + 1$ getauscht und die Eingabe ist damit richtig partitioniert.

Algorithm 4: Lomuto Partitionierungsschema

```
1  function partitioning(array, first, last) : int
2      pivot ← array[last]
3      i ← first − 1
4      for j = first to last − 1 do
5          if array[j] < pivot then
6              i ← i + 1
7              swap(array[i], array[j])
8          end
9      end
10     swap(array[i + 1], array[last])
11     return i+1
12 end
```

2.2 Wahl des Pivotelements

Für die Performanz von Quicksort ist die Wahl des Pivotelement sehr wichtig.

Ursprünglich wurde als Pivotelement das Element an der Ersten oder letzten Stelle gewählt. Es wird schnell klar, dass dies nicht optimal ist, da z.B. vorsortierte Listen als Eingabe ein Worst Case Fall darstellen. Als simple Alternative dafür bietet sich das mittlere Element als Pivotelement an, da eine Worst Case Eingabe dafür deutlich unwahrscheinlicher ist.

Eine weitere, weit verbreitete Abwandlung der Pivotwahl, wurde 1976 von Robert Sedge-

wick, in dessen Paper „The Analysis of Quicksort Programs“ [1] vorgestellt, das median-of-three Prinzip. Hierbei wird als Pivotelement der Median aus drei, nämlich dem ersten, mittleren und letzten Element gebildet.

Algorithm 5: Median of three

```
1  function median of three(array, first, last)
2      mid ← first + (last − first)/2
3      firstElement ← array[first]
4      middleElement ← array[mid]
5      lastElement ← array[last]
6      if middleElement < firstElement then
7          swap(array[mid], array[first])
8      end
9      if lastElement < middleElement then
10         swap(array[last], array[mid])
11     end
12     if middleElement < firstElement then
13         swap(array[mid], array[first])
14     end
15     return middleElement
16 end
```

Die Bestimmung des Medians aus drei Elementen bietet viele Vorteile gegenüber der festen Wahl eines Pivotelements. Durch die Auswahl von drei Elementen ist es sehr unwahrscheinlich, dass immer die drei größten/kleinsten Elemente auswählt, was der Worst Case wäre. Somit sind auch vorsortierte Listen o.Ä., hinsichtlich der Performanz, kein Problem mehr.

Dazu kommt, dass versucht wird, das Pivotelement so zu wählen, dass die Eingabe nach der Partitionierung, für die rekursiven Aufrufe, in der Mitte geteilt wird für die rekursiven Aufrufe. Dabei ist es auch von Vorteil, wenn der Median aus drei Elementen bestimmt wird, da somit die Wahrscheinlichkeit für ein „gutes“ Pivotelement steigt.

3 Theoretische Performanz von Quicksort

Wie bereits erwähnt, hängt die Performanz (bzw. Anzahl der Vergleiche) von der Wahl des Pivotelements ab. Hier lässt sich zwischen drei Fällen unterscheiden: Best Case, Average Case und Worst Case.

Für die folgenden Berechnung wird angenommen, dass immer genau ein Element als Pivotelement dient, es werden also keine zusätzlichen Vergleiche zur Bestimmung des Pivotelements benötigt (vgl. [2] und [3]).

Die Anzahl an Vergleiche, die Quicksort für Eingabegröße $n-1$ benötigt, wird als $C(n)$ definiert. Für die Partitionierung werden in jedem Schritt n Vergleiche durchgeführt. Damit ergibt sich

$$C(n) = n - 1 + C(k-1) + C(n-k) \text{ mit } C(0) = C(1) = 0$$

wobei $C(k-1)$ die Rekursion des linken Teils und $C(n-k)$ die Rekursion des rechten Teils darstellt.

3.1 Best Case

Im Besten Fall wird die Eingabe halbiert und die Rekursion mit Eingabegröße $\frac{n-1}{2}$ aufgerufen. Mit Rekursionstiefe k und $n = 2^k$ ergibt sich annäherungsweise

$$\begin{aligned} C(n) &= n - 1 + 2C(\frac{n}{2}) \\ &= 2[2C(\frac{n}{4}) + \frac{n}{2} - 1] + n - 1 \\ &= 2^k C(\frac{n}{2^k}) + kn - (2^k - 1) \end{aligned}$$

Durch Einsetzen von $2^k = n \iff k = \log_2(n)$ ergibt sich

$$C(n) = n\log_2(n) - n + 1 \in \mathcal{O}(n \log_2(n))$$

für die Anzahl an Vergleichen im Best Case.

3.2 Worst Case

Im Schlechtesten Fall wird als Pivotelement dass jeweils größte oder kleinste Element der Eingabe gewählt. Dadurch ist pro Durchgang in einem Rekursionsaufruf nur ein Element

und $n-1$ Elemente im anderen. Für die Anzahl der Vergleiche ergibt sich dann:

$$\begin{aligned} C(n) &= C(n-1) + n - 1 \\ &= C(n-2) + n - 1 + n - 2 \\ &= C(1) + \sum_{i=0}^{n-1} (n-i) \\ &= \frac{n(n-1)}{2} \in \mathcal{O}(n^2) \end{aligned}$$

im schlechtesten Fall.

3.3 Average Case

Da sowohl der Best Case, als auch der Worst Case normalerweise nicht eintreten, wird als dritter Fall der Average Case betrachtet.

Das Pivotelement wird aus einer zufälligen Eingabe gewählt, somit ist die Wahrscheinlichkeit, dass dies das k-größte Element ist $\frac{1}{k}$. Nach der Partitionierung befindet es sich dann an Position k der Eingabe. Über alle k ergibt sich dann:

$$\begin{aligned} C(n) &= \frac{1}{n} \sum_{k=1}^{n} (n + C(k-1) + C(n-k)) \\ &= n + \frac{1}{n} \sum_{k=1}^{n} C(k-1) + \frac{1}{n} \sum_{k=1}^{n} C(n-k) \end{aligned}$$

Da beide Summen das $C(0)$ bis $C(n-1)$ addieren, können sie zusammengefasst werden.

$$C(n) = n + \frac{2}{n} \sum_{i=0}^{n-1} C(i), \text{ bzw.} \tag{1a}$$

$$nC(n) = n^2 + 2 \sum_{i=0}^{n-1} C(i) \tag{1b}$$

Wenn in (1b) n durch $n-1$ ersetzt wird ergibt sich:

$$(n-1)C(n-1) = (n-1)^2 + 2 \sum_{i=0}^{n-2} C(i) \tag{2a}$$

Nun wird (2a) von (1b) subtrahiert

$$(n)C(n) = (n+1)C(n-1) + 2n - 1$$

Geteilt durch $n(n+1)$

$$\frac{C(n)}{n+1} = \frac{C(n-1)}{n} + \frac{2n-1}{n(n+1)}$$

Dies wird approximiert mit:

$$\frac{C(n)}{n+1} = \frac{C(n-1)}{n} + \frac{2}{n}$$

Jetzt ergibt sich mit $D(n) = \frac{C(n)}{n+1}$ und $D(1) = 0$:

$$\begin{aligned}
D(n) &= D(n-1) + \frac{2}{n} \\
&= D(n-2) + \frac{2}{n-1} + \frac{2}{n} \\
&= D(n-3) + \frac{2}{n-2} + \frac{2}{n-1} + \frac{2}{n} \\
&= D(1) + \frac{2}{2} + \frac{2}{3} + \ \ldots\ + \frac{2}{n-2} + \frac{2}{n-1} + \frac{2}{n} \\
&= 2\ln(2) - 2 \\
&\approx 2\ln(2) \\
&= 2\ln(2)\ \log_2(2) \\
&= 1.39\ \log_2(n)
\end{aligned}$$

Also ist $C(n) = (n+1)D(n) \approx 1.39\ (n+1)\ \log_2(n) \approx 1.39\ n\ \log_2(x)$

Es fällt auf, dass der Average Case lediglich 39 % mehr Vergleiche als der Best Case benötigt. Damit liegt der Average Case viel näher am Best Case, als am Worst Case.

4 Quicksort in der Praxis

Da Quicksort ein oft benutzter Sortieralgorithmus ist, wurde untersucht, wie verschiedene Programmiersprachen bzw. Compiler Quicksort implementiert haben und wie sie ihre Pivotwahl duchführen, falls sie Quicksort benutzen. Dafür wurde die Implementierung der Standard Sort Funktion der jeweiligen Sprache untersucht.

4.1 C

4.1.1 Glib

Die Gnu C-Bibliothek Glib benutzt einen sogenannten Introsort Algorithmus, dieser Sortieralgorithmus startet mit Quicksort. Sobald die Eingabegröße einen Wert unterschreitet (hier vierfache Typgröße z.B. 16 für Integer), wird auf Insertion Sort gewechselt, da Insertion Sort für kleine Eingaben schneller ist als Quicksort.

Zur Bestimmung des Pivotelemente wird auf Median aus 3 zurückgegriffen, dieser wird aus dem Ersten, Mittleren und Letzten Element gebildet. [4]

4.2 C++

Für C++ unterscheidet sich die Implementierung je nach Compiler, bzw. nach deren C++ Bibliothek.

4.2.1 Libstdc++

Die Libstdc++ Bibliothek wird z.B. von Gnu benutzt. In der Vergangenheit (bis Version 4.6) wurde das Pivotelement durch Median aus 3 bestimmt. Hierbei wurden das Erste, Mittlere und Letzte Element benutzt. [5]

Mittlerweile (ab Version 4.7) wurde dies etwas modifiziert, der Median aus 3 wird nun mit dem Zweiten, Mittleren und Letztem Element gebildet. [6] Ab einer Eingabelänge unter 30, wird auf Insertion Sort gewechselt.

4.2.2 Libc++

Die Libc++ Bibliothek wird unter anderem von Clang genutzt, dort wird auch Introsort benutzt. Für die Pivotwahl wird Median aus 5 benutzt, dabei werden das Erste, Mittlere und Vorletzte Element, sowie die zwei Elemente, die dazwischen liegen (Position $\frac{1}{4}$ und $\frac{3}{4}$ der Eingabelänge) benutzt.

Sobald die Eingabelänge kleiner als 1000 ist, wird auf Median aus 3 gewechselt und das Erste, Mittlere und Vorletzte Element verglichen. Bei einer Eingabelänge kleiner 30 wird auf Insertion Sort ausgewichen. [7]

4.2.3 Visual Studio

Die „Microsoft Standard Library“ implementiert auch Introsort. Bis zu einer Eingabegröße von 32 wird Quicksort verwende, darunter Insertion Sort.

Zur Bestimmung des Pivotelements wird „Tukey's Ninther“, auch genannt „Pseudomedian aus 9“, benutzt. Hierbei wird dreimal der Median aus 3 gebildet, mit jeweils verschiedenen Eingabe. Danach wird ein letzter Median gebildet, aus den Medians der vorherigen Berechnungen, welcher dann als Pivotelement benutzt wird. Konkret werden hier die Mediane der Elemente an den Stellen: $[0, \frac{1}{8}, \frac{2}{8}]$; $[\frac{3}{8}, \frac{4}{8}, \frac{5}{8}]$ und $[\frac{6}{8}, \frac{7}{8}, n]$ der Eingabe berechnet.

Bei einer Eingabegröße unter 40 wird jedoch auf Median aus 3 gewechselt, dann werden das Erste, Mittlere und Letzte Element verglichen. [8]

4.3 Php

Php benutzt für die meisten sort Funktionen Quicksort, dabei wird das Mittlere Element als Pivotelement gewählt. [9]

4.4 Java

Java selbst benutzt Dual Pivot Quicksort, jedoch benutzt GNU Classpath (Open Source Implementierung der Standardklassenbibliotheken) Quicksort. Dabei wird Tukey's Ninther bzw. Pseudomedian aus 9 benutzt. Es wird dreimal der Median aus 3 bestimmt und zuletzt der finale Median aus den drei vorherigen. Sobald die Eingabelänge kleiner als 40 ist wird auf Median aus 3 zurückgegriffen. Für Kleine Eingaben (≤ 7) wird Insertion Sort verwendet. [10]

4.5 Swift

Bis Swift 5 wurde Introsort als Standard Sortieralgorithmus benutzt. Hierbei wurde ab einer Eingabegröße von 20 Elementen auf Insertion Sort gewechselt. Die Pivotwahl wurde als Median aus 3 implementiert, welcher aus dem Ersten, Mittleren und Letzten Element gebildet wird. [11]

4.6 Rust

Rust benutzt auch Tukey's Ninther/Pseudomedian aus 9, hierbei werden die 3 Mediane der Elemente an Stelle $\frac{1}{4}/\frac{2}{4}/\frac{3}{4}$ (der Eingabelänge) und deren direkten Nachbarelementen berechnet. Das Pivotelement ist dann der Median von diesen drei Positionen.

Ab Eingabelänge unter 50 wird zu Median aus 3 gewechselt, der wieder aus den Elementen an Stelle $\frac{1}{4}/\frac{2}{4}/\frac{3}{4}$ berechnet wird, für Eingabegrößen kleiner 10 wird schließlich auf Insertion

Sort ausgewichen. [12]

4.7 Fortran

Fortran besitzt keine eingebauten Sortieralgorithmen.

4.8 Python

Python benutzt Timsort als eingebauten Sortieralgorithmus. [13]

4.9 Zusammenfassung

Nachfolgend sind die Ergebnisse der Recherche aufgelistet, für alle Sprachen bzw. Bibliotheken, die Quicksort benutzen.

Sprache	Pivotwahl	Partitionierung	Grenze Mo3	Grenze I Sort
C Glib	Median aus 3	Hoare	-	16
C++ Libstdc++	Median aus 3	Hoare	-	16
C++ Libc++	Median aus 5	Hoare	1000	30
C++ Visual Studio	Tukey's Ninther	Hoare	40	32
Php	Mittleres Element	Hoare	-	-
Java (GNU)	Tukey's Ninther	Hoare	40	7
Swift	Median aus 3	Hoare	-	20
Rust	Tukey's Ninther	Hoare	50	10

5 Testframework

5.1 Technologie und Allgemeines

Das Framework wurde in C++ geschrieben, alle Diagramme, Boxplots etc. wurden mit Python erstellt. Als Eingabe für die Sortierfunktionen wird std::vector benutzt. Um die Anzahl der Vergleiche zu zählen, wird eine eigene Compare Funktion benutzt und für die Zählung der Zuweisungen eine eigene Klasse.

Alle Tests wurden mit Daten vom Typ Integer durchgeführt bzw. die eigene Klasse, welche als einziges Datenfeld einen Integer Wert besitzt. Die Quicksort Funktionen sind als Introsort implementiert, ab einer bestimmten Eingabegröße wird auf Insertion Sort gewechselt, bzw. für größere Pivot Methoden erst auf Median aus 3 und dann auf Insertion Sort.

Die Schranken für den Wechsel auf Median aus 3 und Insertion Sort werden im Folgenden noch näher untersucht.

5.2 Insertion Sort

Für kleine Eingaben wird Insertion Sort mit folgender Implementierung benutzt.

Algorithm 6: Insertion Sort

```
1  function insertionSort(array, first, last)
2      if first ≤ last then
3          return
4      end
5
6      for i ← first + 1 to last do
7          current ← array[i]
8          prev ← i
9          while (prev > first) && current < array[prev − 1] do
10             array[prev] ← array[prev − 1]
11             prev − −
12         end
13         array[prev] ← current
14     end
15 end
```

6 Implementierung aller Pivotwahlmethoden

Es wurde folgende Hilfsfunktion implemetiert, welche zwei Elemente vergleicht und dann das kleinere von beiden Elementen an die linke Stelle tauscht.

Algorithm 7: Sortiere zwei Elemente in Array

```
1 function sortElements(array, first, second)
2     if array[second] < array[first] then
3         swap(array[first], array[second])
4     end
5 end
```

Die gefundenen Arten der Pivotwahlmethoden aus der Praxis wurden in das Framework übernommen und auf folgende Weise implementiert:

6.1 Median aus 3

Median aus 3 wird wie in den meisten Fällen implementiert, das Pivotelement ist der Median des Ersten, Mittleren und Letztem Element.

Algorithm 8: Median aus 3

```
1 function medianOf3(array, first, last)
2     middle ← first + (last−first)/2
3
4     sortElements(array, second, first)
5     sortElements(array, third, second)
6     sortElements(array, second, first)
7
8     return array[middle]
9 end
```

6.2 Median aus 5

Median aus 5 wird wie in der libc++ Bibliothek implementiert, es wird der Median des Ersten, Mittleren, Letztem, sowie den Elementen an Stelle $\frac{1}{4}$ und $\frac{3}{4}$ der Eingabe bestimmt.

Algorithm 9: Median aus 5

```
1  function medianOf5(array, first, last)
2      length ← last − first
3      quarter ← first + length/4
4      middle ← first + length/2
5      threeQuarter ← first + 3 * length/4
6
7      sortElements(array, first, quarter)
8      sortElements(array, threeQuarter, last)
9      sortElements(array, first, threeQuarter)
10     sortElements(array, quarter, last)
11     sortElements(array, middle, threeQuarter)
12     sortElements(array, quarter, middle)
13     sortElements(array, middle, threeQuarter)
14
15     return array[middle]
16 end
```

6.3 Median aus 7

Als eine weitere Methode, welche nicht aus der Bibliothek einer Programmiersprache entnommen wurde, wird Median aus 7 hinzugefügt. Diese Methode zur Pivotwahl wurde aus einem Paper [14] entnommen.

Algorithm 10: Median aus 7

```
1  function medianOf7(array, first, last)
2      length ← last − first
3      sevenArray[0] ← first
4      sevenArray[1] ← first + length/6
5      sevenArray[2] ← first + length/3
6      sevenArray[3] ← first + length/2
7      sevenArray[4] ← first + 2 ∗ length/3
8      sevenArray[5] ← first + 5 ∗ length/6
9      sevenArray[6] ← last
10
11     insertionSort(sevenArray, 0, 6)
12
13     return array[sevenArray[3]]
14 end
```

6.4 Tukey's Ninther - Pseudomedian aus 9

Tukey's Ninther wird nach dem Vorbild von Visual Studio übernommen. Der Median wird zwischen dem Ersten, Mittleren und Letzten, sowie jeweils drei Elementen dazwischen bestimmt.

Algorithm 11: Tukey's Ninther - Pseudomedian aus 9

```
1  function tukeyysNinther(array, first, last)
2      length ← last − first
3      step ← length/8
4      middle ← first + length/2
5      twoStep ← step ∗ 2
6
7      firstMedian ← medianOf3(array, first, first + stepfirst + twoStep)
8      secondMedian ← medianOf3(array, middle − step, middle, middle + step)
9      thirdMedian ← medianOf3(array, last − twoStep, last − step, last)
10
11     finalMedian ← medianOf3(array, first + step, middle, last − step)
12     return finalMedian
13 end
```

7 Optimierung der Vergleichsanzahl

Wie bereits erwähnt, benötigt die Partitionierung $n - 1$ Vergleiche, dazu kommen die Vergleiche, welche durch die Bestimmung des Pivotelement durchgeführt werden. Dies benötigt zusätzlich zwischen 3 (Median aus 3) und 21 (Worst Case bei Median aus 7) Vergleiche.

Um die Anzahl der Vergleiche zu verringern, wird sich zunutze gemacht, dass bei der Bestimmung eines Medians die Elemente schon sortiert werden. Dadurch können die Elemente nach der Median Bestimmung an die Außenseite der Eingabe getauscht und dann bei der Partitionierung ignoriert werden.

Beispielsweise bei Median aus 5 werden die ersten 2 Elemente (kleiner als das Pivotelement) an den Anfang getauscht und die letzten 2 Elemente (größer als das Pivotelement) ans Ende der Eingabe. Danach werden in der Partitionierung die beiden Zeiger i und j zwei Positionen weiter innen initialisiert, wodurch 4 Vergleiche gespart werden. Damit benötigt die Partitionierung nur noch $n - 5$ Vergleiche.

Für alle Varianten ergeben sich dann folgende Vergleichszahlen für die Pivotwahl und Partitionierung:

Pivot Methode	Vergleiche zur Pivotwahl	Vergleiche mit Partitionierung
Erstes Element	0	$n - 1$
Mittleres Element	0	$n - 1$
Letztes Element	0	$n - 1$
Median aus 3	3	n
Median aus 5	7	$n + 2$
Median aus 7	$\leq 21; \sim 15$ *	$n + \sim 8$
Tukey's Ninther	12	$n + 9$

* Zur Bestimmung des Medians aus den 7 Elementen wird Insertion Sort benutzt. Dies benötigt im Schnitt 15 Vergleiche, um die 7 Elemente zu sortieren.

8 Experiment: Grenze für Median aus 3

Ziel der Methoden, welche viele Elemente zur Bestimmung des Pivotelements vergleichen, ist es, ein möglichst gutes Pivotelement zu wählen. Dabei stellt sich die Frage, bis zu welcher Eingabegröße es einen größeren Nutzen hat, viele Elemente zu vergleichen anstatt auf eine andere Form der Pivotwahl, welche weniger Elemente vergleicht, zurückzugreifen.

In der Praxis wird ab einer bestimmten Größe auf Median aus 3 gewechselt, diese Grenze variiert allerdings stark.

Es soll nun untersucht werden, was die Auswirkungen eines Wechsels auf Median aus 3 sind und wie sich diese für verschiedene Grenzwerte entwickeln. Dafür werden einige Grenzwerte von 30 bis 2000 getestet.

8.1 Ergebnisse

Die Durchläufe wurden mit einer Eingabegröße von 1.000.000 durchgeführt, jede Median aus 3 Grenze wurde 500 mal getestet, um das Gewicht von Ausreißern zu minimieren. Zum Vergleich ist an Stelle 0 die jeweilige Pivotwahl, ohne ein Wechsel auf Median aus 3, dargestellt.

8.1.1 Vergleiche

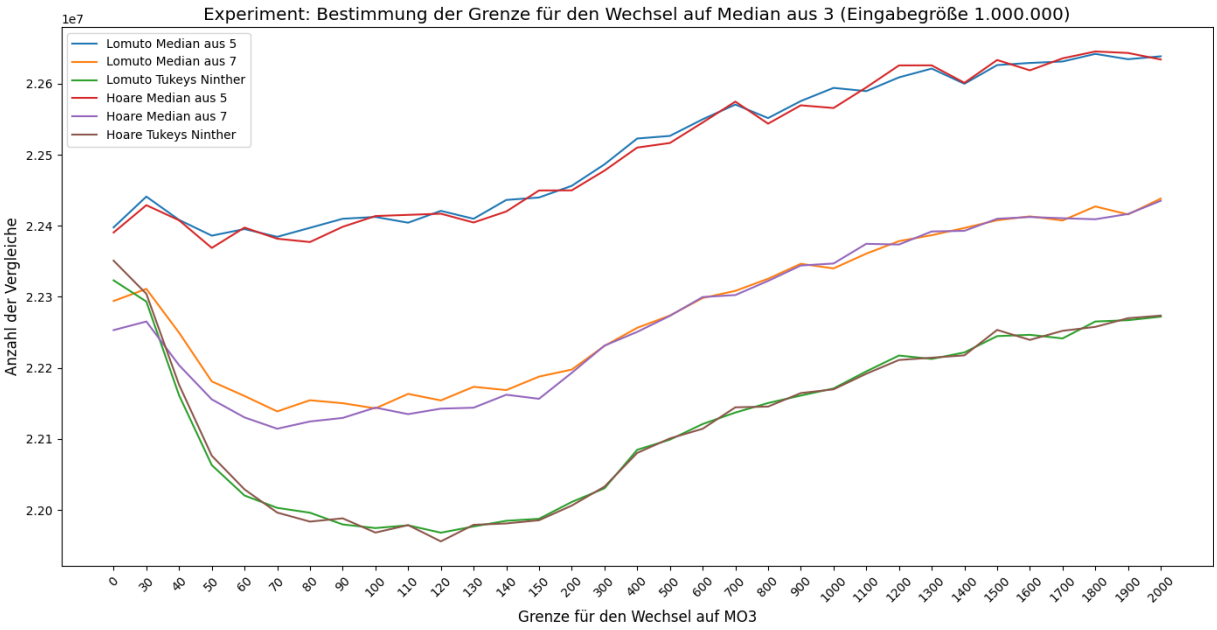

Abbildung 1: Vergleichszahlen - Experiment: Grenze für Median aus 3

Es fällt auf, je mehr Elemente für die Median Bestimmung benutzt werden, desto größer

ist der Nutzen, hinsichtlich der Vergleichsanzahl, auf Median aus 3 zu wechseln. Dies wird insbesondere bei Median aus 7 und Tukey's Ninther deutlich. Das Optimum liegt für Median aus 7 bei ca. 70 Elementen und 120 bei Tukey's Ninther.

Bei Median aus 5 ist der Nutzen des Wechsels auf Median aus 3 nur minimal, da Median aus 5 auch nur zwei Vergleiche mehr pro Partitionierung braucht.

8.1.2 Laufzeit

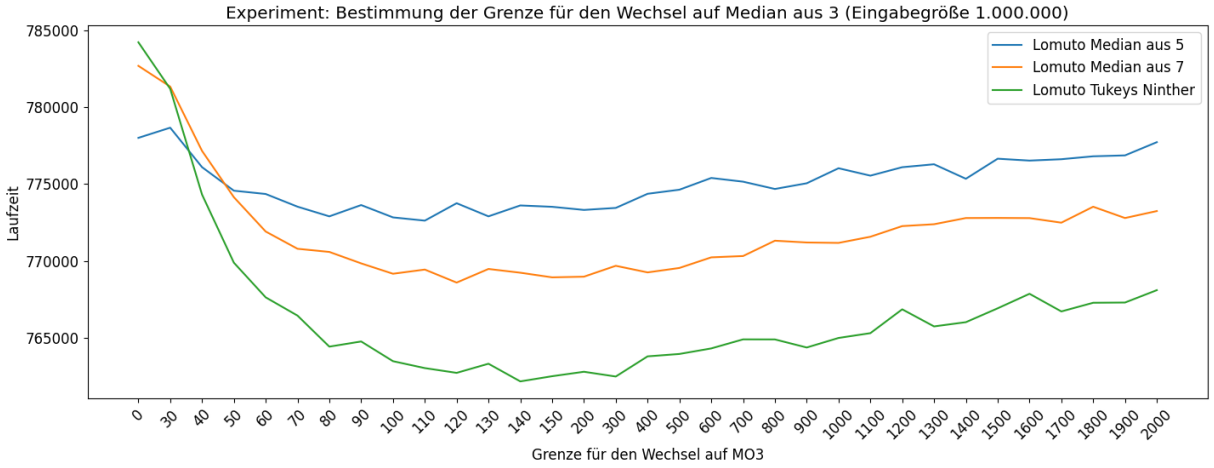

Abbildung 2: Laufzeit Lomuto Funktionen - Experiment: Grenze für Median aus 3

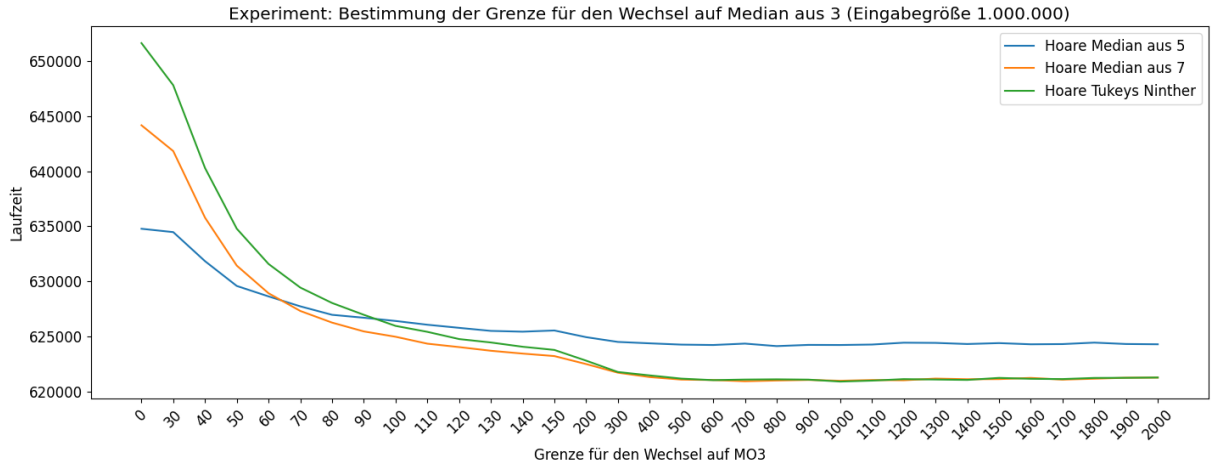

Abbildung 3: Laufzeit Hoare Funktionen - Experiment: Grenze für Median aus 3

Hinsichtlich der Laufzeit ist der Nutzen für alle Methoden deutlich, was nicht überrascht, da Median aus 3 eben nur 3 Elemente vergleicht bzw. ordnet, also deutlich weniger als z.B. bei Tukey's Ninther.

8.1.3 Zuweisungen

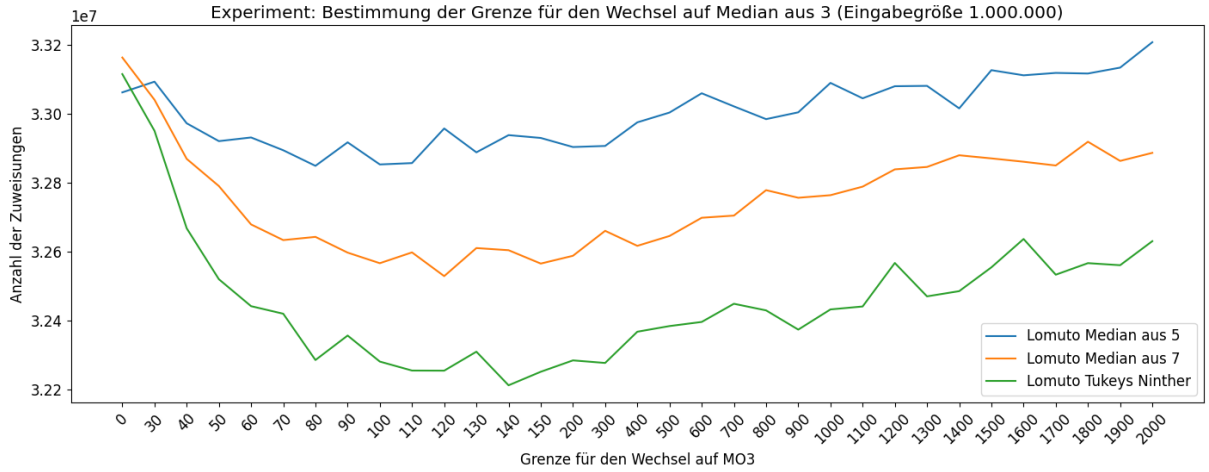

Abbildung 4: Zuweisungen Lomuto Funktionen - Experiment: Grenze für Median aus 3

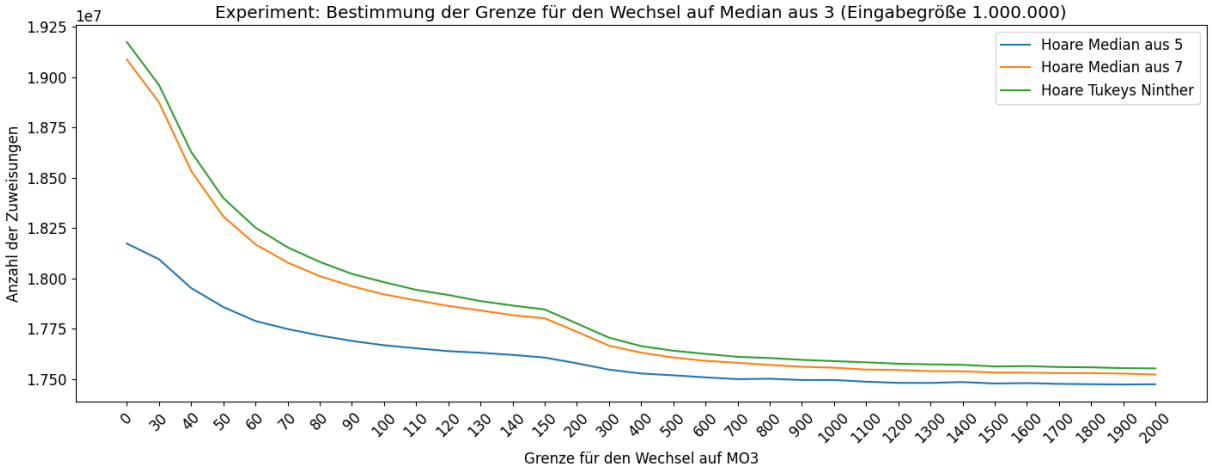

Abbildung 5: Zuweisungen Hoare Funktionen - Experiment: Grenze für Median aus 3

Die Auswirkungen auf die Zuweisung ähnelt sehr den Auswirkungen der Laufzeit.

8.2 Zusammenfassung

Hinsichtlich der Vergleichszahlen sind folgende Optimas festzustellen:

Pivotwahl	Bester Grenzwert	Vergleiche	
		mit Mo3	ohne Mo3
Lomuto Median aus 5	70	22384520	22397810
Lomuto Median aus 7	70	22138835	22294213
Lomuto Tukey's Ninther	120	21968156	22323225
Hoare Median aus 5	50	22368887	22390599
Hoare Median aus 7	70	22114449	22253240
Hoare Tukey's Ninther	120	21955918	22350945

Da der Wechsel auf Median aus 3 für alle Blickpunkte Vorteile bringt, werden die „besten Grenzwerte“, welche in der Tabelle aufgelistet sind, im Testframework übernommen.

9 Experiment: Grenze für Insertion Sort

In den meisten Quicksort Implementierungen wird für sehr kleine Eingabegrößen auf Insertion Sort gewechselt. Die Auswirkungen davon werden nachfolgend, für die Pivotwahlmethoden: Erstes/Mittleres/Letztes Element, sowie Median aus 3 untersucht.

Für die Restlichen Methoden wird Tukey's Ninther als Referenz getestet, da alle diese Methoden schließlich auf Median aus 3 wechseln.

9.1 Ergebnisse

Die Durchläufe wurden mit einer Eingabegröße von 100.000 durchgeführt, jede Insertion Sort Grenze wurde 500 mal getestet.

Zum Vergleich ist an Stelle 0 die jeweilige Pivotwahl, ohne ein Wechsel auf Insertion Sort dargestellt. Bei Median aus 3 und Tukey's Ninther wird hier, ab einer Eingabegröße von 3, auf Mittleres Element als Pivot gewechselt.

9.1.1 Vergleiche

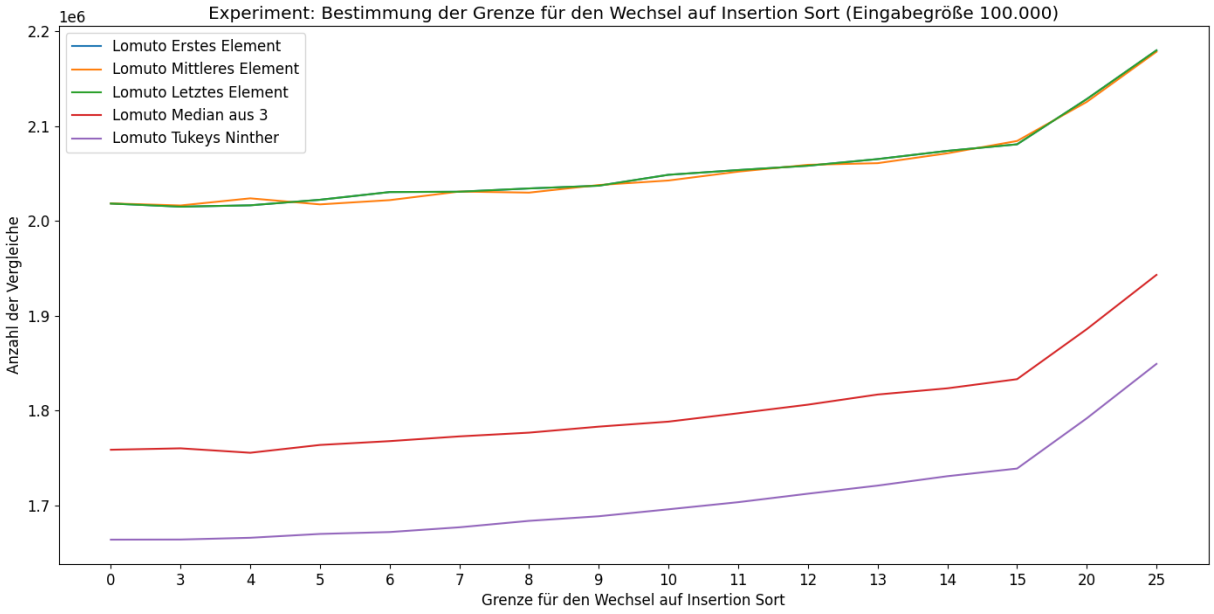

Abbildung 6: Vergleiche Lomuto Funktionen - Experiment: Grenze für Insertion Sort

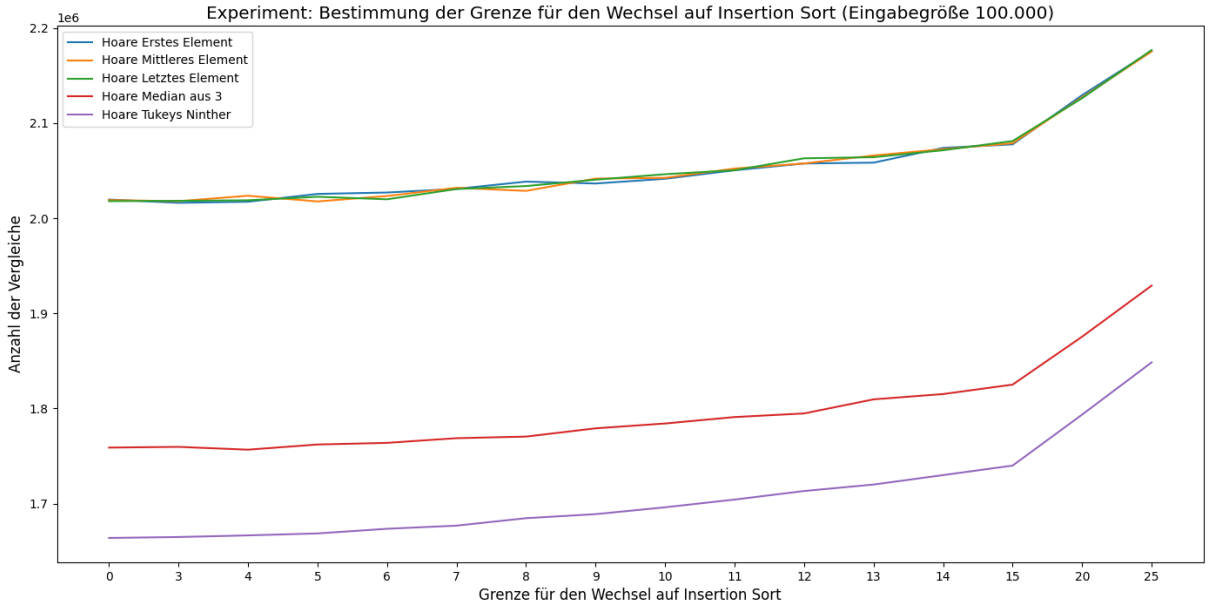

Abbildung 7: Vergleiche Hoare Funktionen - Experiment: Grenze für Insertion Sort

Für die Anzahl an Vergleichen, macht es fast keinen Unterschied, ob auf Insertion Sort gewechselt wird oder nicht. (Bezüglich dem Optimalen Wechselzeitpunkt für Insertion Sort)

9.1.2 Laufzeit

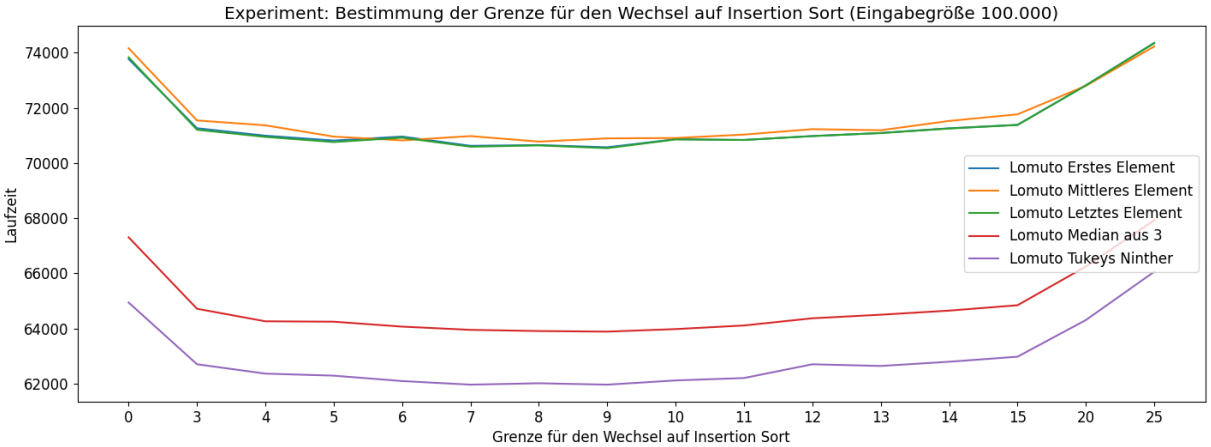

Abbildung 8: Laufzeit Lomuto Funktionen - Experiment: Grenze für Insertion Sort

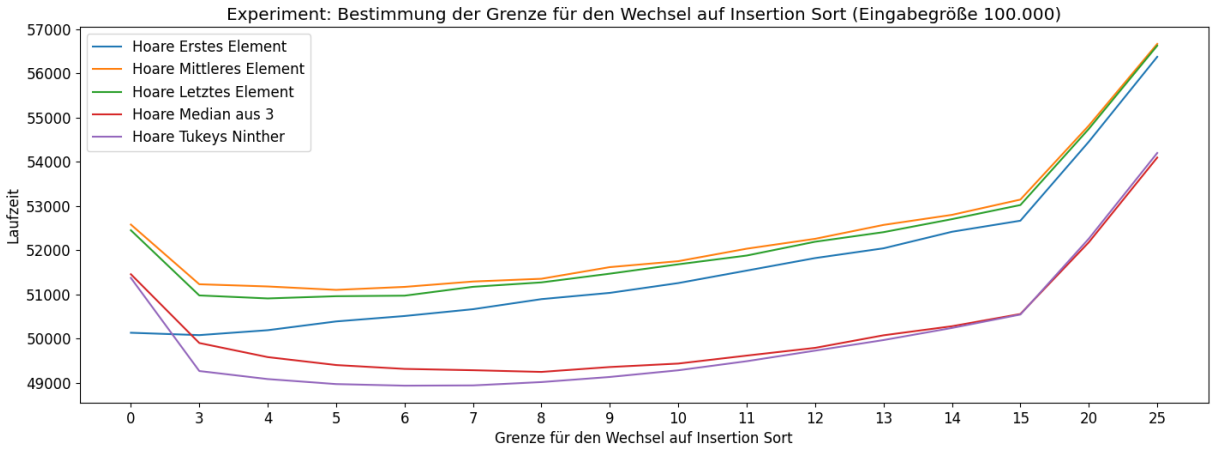

Abbildung 9: Laufzeit Hoare Funktionen - Experiment: Grenze für Insertion Sort

Für beide Partitionierungsmethoden ist es ein deutlicher Gewinn, wenn auf Insertion Sort gewechselt wird. Ein früher Wechsel (bei Eingabegröße > 20) auf Insertion Sort hingegen, resultiert in einer erhöhten Laufzeit.

9.1.3 Zuweisungen

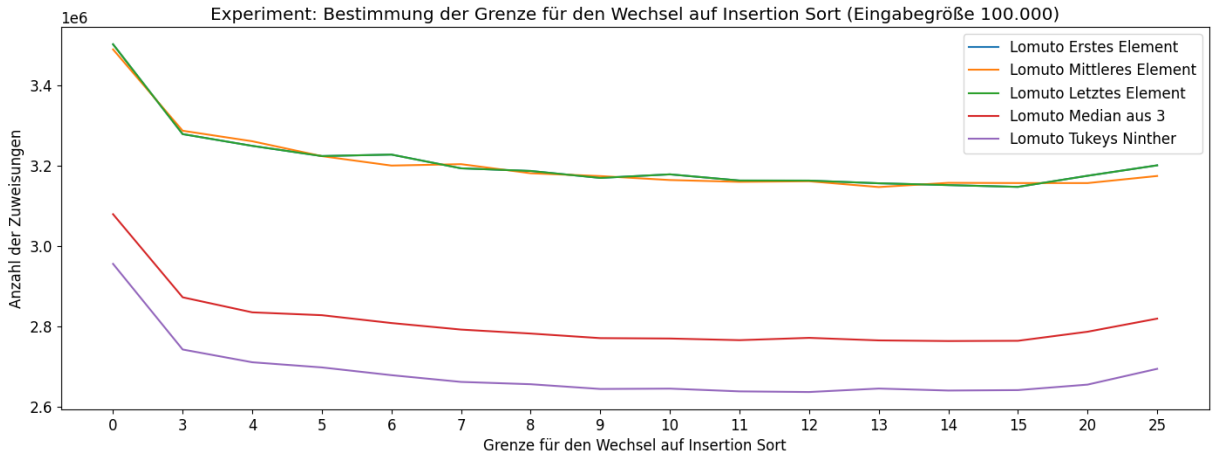

Abbildung 10: Zuweisungen Lomuto Funktionen - Experiment: Grenze für Insertion Sort

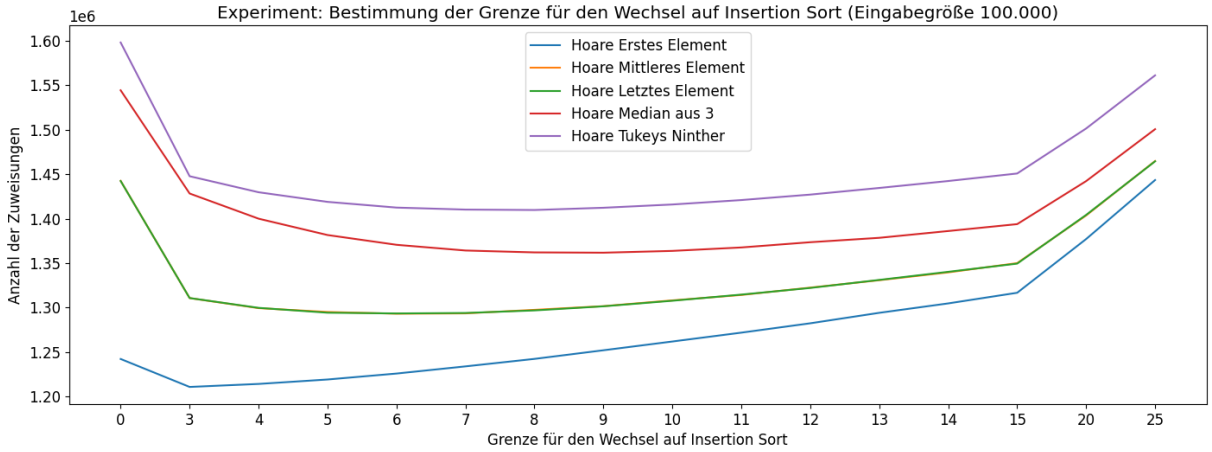

Abbildung 11: Zuweisungen Hoare Funktionen - Experiment: Grenze für Insertion Sort

Im Bezug auf die Zuweisungen ist auch eine Verbesserung feststellbar, wenn für kleine Eingabegröße auf Insertion Sort gewechselt wird. Insbesondere bei den Lomuto Funktionen macht sich dies bemerkbar.

9.2 Zusammenfassung

Zusammenfassend kann es definitiv Sinn machen, auf Insertion Sort zu wechseln. Es sind deutliche Verbesserungen der Laufzeit und eine Verringerung der Zuweisungen festzustellen, wenn man bei sehr kleinen Eingabegrößen auf Insertion Sort wechselt (Eingabegröße 3-10). Die Anzahl an Vergleichen ist zwar ohne Insertion Sort am niedrigsten, jedoch ist diese Erhöhung der Vergleiche, im Gegensatz zu den anderen Verbesserungen, eher unwesentlich.

Wenn jedoch sehr früh (Eingabegröße $>$ 30) auf Insertion Sort gewechselt wird, verschlechtert sich die Laufzeit und die Anzahl an Zuweisungen und Vergleichen erhöht sich stark.

10 Benchmarks aller Methoden

Die vorhandenen Methoden zur Pivotwahl werden hinsichtlich ihrer Laufzeit, Anzahl an Vergleichen und Zuweisungen miteinander verglichen. Dabei werden alle Methoden, mit den zuvor bestimmten Grenzen für Median aus 3 und Insertion Sort implementiert und getestet.

10.1 Ergebnisse

Alle Methoden wurden auf verschiedenen Input Größen getestet: 1.000, 5.000, 10.000, 50.000, 100.000, 250.000, 500.000, 1.000.000 und 16.000.000. Die nachfolgend abgebildeten Boxplots sind auf Basis der Eingabegröße 16.000.000 (Die Ergebnisse für andere Eingabegrößen sind am Ende der Arbeit zu finden).

Die Boxplots sind standardmäßig aufgebaut, die Box reicht vom unteren Quartil zum oberen Quartil. Der Mittelwert ist als Grünes Dreieck eingezeichnet und der Median als Grüne Linie. Die Länger der Whisker beträgt 1,5-Facher Interquartilsabstand von dem jeweiligen Quartil und die Ausreißer sind als Kreise eingezeichnet.

10.1.1 Vergleiche

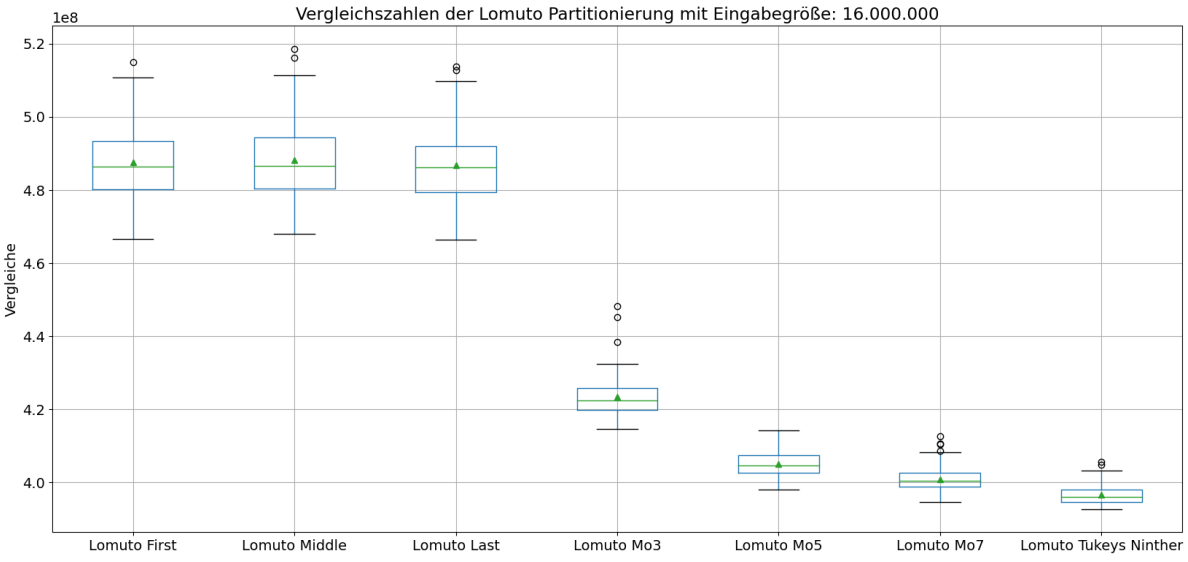

Abbildung 12: Vergleiche Lomuto Funktionen - Benchmarks aller Methoden

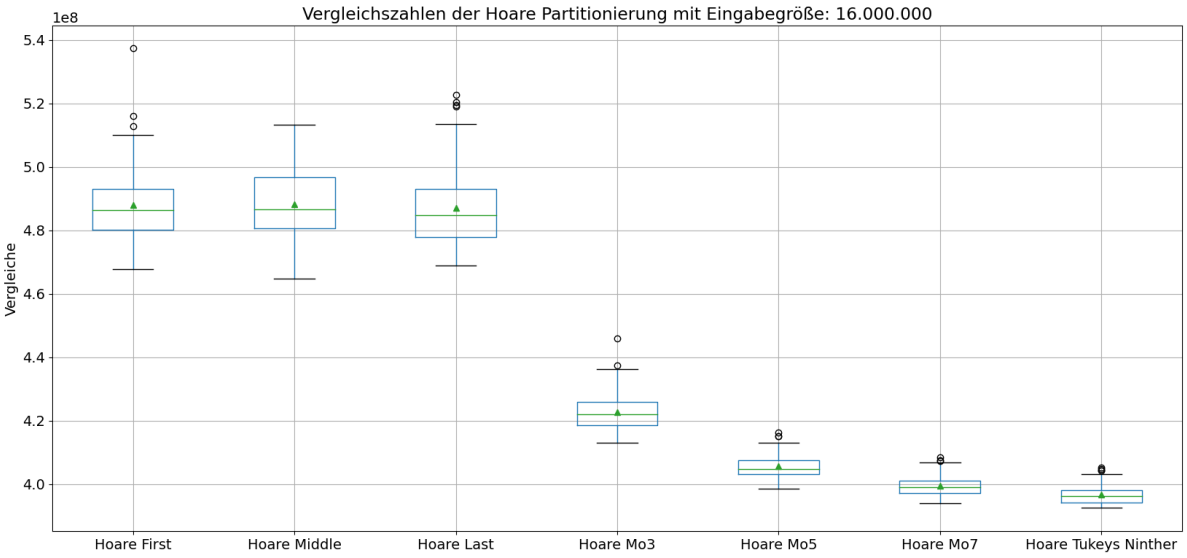

Abbildung 13: Vergleiche Hoare Funktionen - Benchmarks aller Methoden

	Lomuto	Hoare
Pivotwahl	Mittelwert der Vergleiche	Mittelwert der Vergleiche
Erstes Element	$487622662,6$	$488044786,8$
Mittleres Element	$488200952,5$	$488138998,1$
Letztes Element	$486769349,0$	$487037435,2$
Median aus 3	$423397648,5$	$422662539,6$
Median aus 5	$405152286,2$	$405653642,5$
Median aus 7	$400929195,2$	$399478240,2$
Tukey's Ninther	$396646435,6$	$396789537,8$

Die Anzahl der durchgeführten Vergleiche ist für beide Partitionierungen fast identisch, da beide Partitionierungen $n-1$ Vergleiche zur Partitionierung benötigen.

Außerdem fällt auf, das die Anazhl an Vergleichen sinkt, je mehr Elemente zur Berechnung des Pivotelements benutzt werden. Die zusätzlichen Vergleiche, die die Pivotwahl benötigt, zahlen sich also aus, da ein „besseres“ Pivotelement gefunden wird. Tukey's Ninther benötigt insgesamt am wenigsten Vergleiche.

10.1.2 Laufzeit

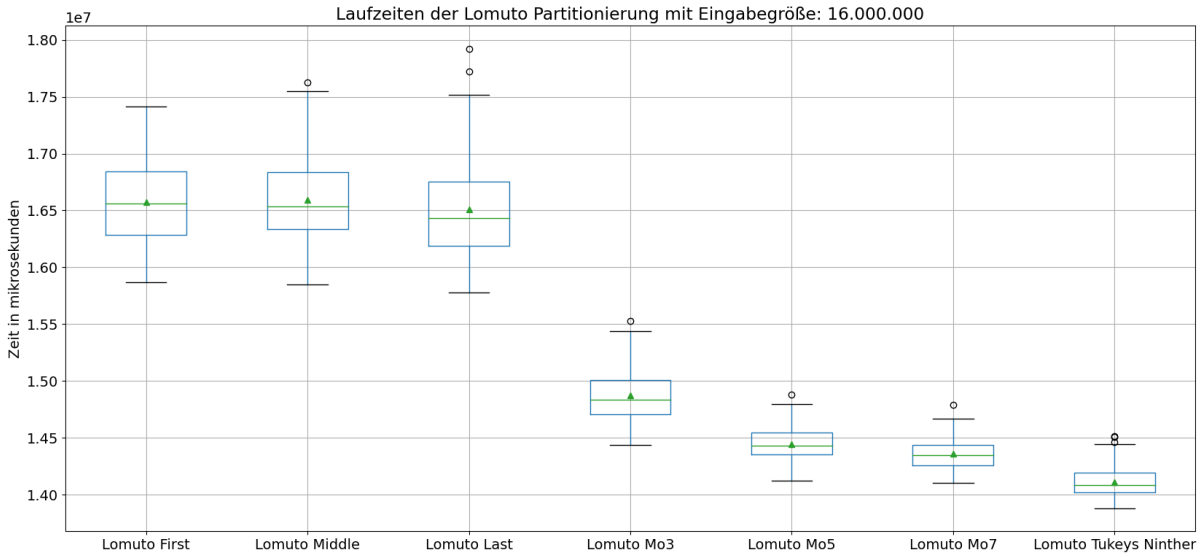

Abbildung 14: Laufzeit Lomuto Funktionen - Benchmarks aller Methoden

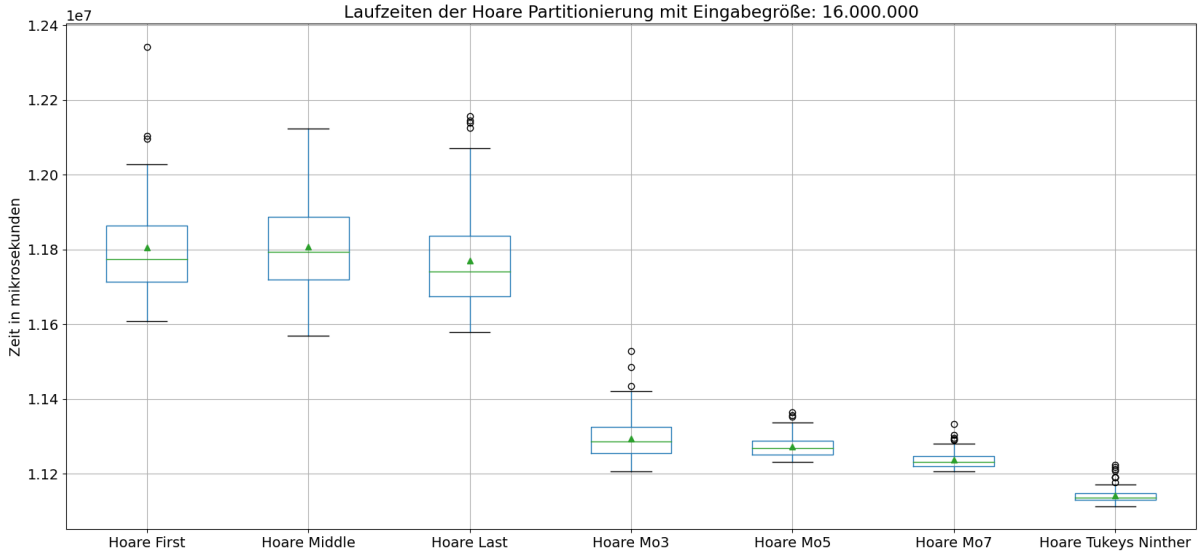

Abbildung 15: Laufzeit Hoare Funktionen - Benchmarks aller Methoden

	Lomuto	Hoare
Pivotwahl	Mittelwerte der Laufzeiten	Mittelwerte der Laufzeiten
Erstes Element	16573455, 21	11804328, 90
Mittleres Element	16591851, 71	11807581, 83
Letztes Element	16509927, 27	11770076, 37
Median aus 3	14873105, 77	11294158, 76
Median aus 5	14443880, 06	11273085, 91
Median aus 7	14359575, 26	11236555, 74
Tukey's Ninther	14112428, 85	11142259, 43

Hinsichtlich der Laufzeit fällt auf, dass die Hoare Partitionierung generell schneller als die Lomuto Partitionierung ist.

Unter den einzelnen Methoden ist die Tendenz ähnlich, wie bei der Anzahl der Vergleiche. Je mehr Elemente zur Bestimmung des Medians benutzt werden, desto schneller sortieren diese die Eingabe.

10.1.3 Zuweisungen

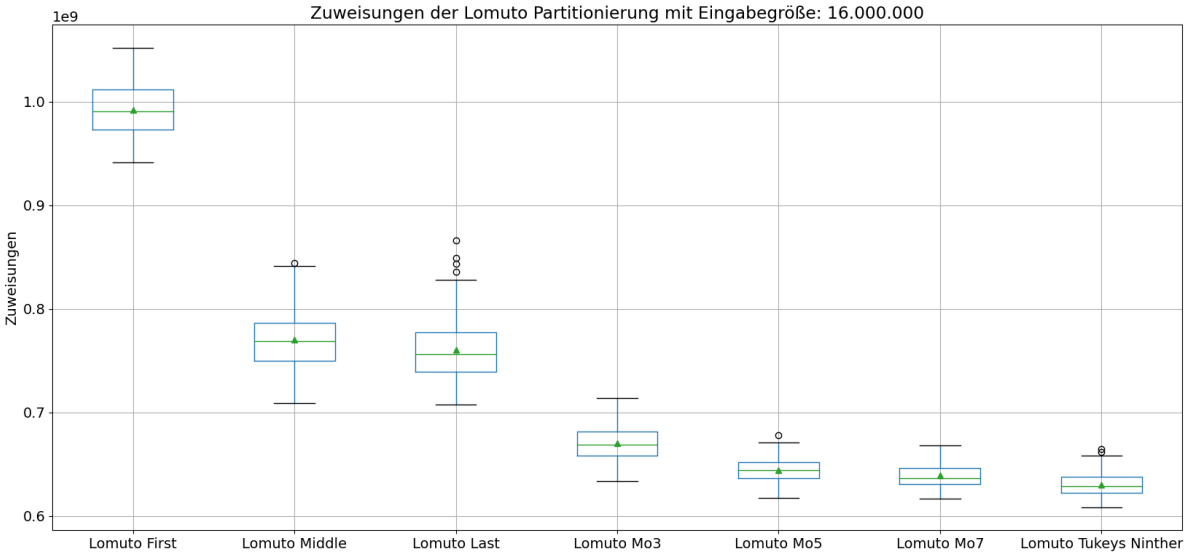

Abbildung 16: Zuweisungen Lomuto Funktionen - Benchmarks aller Methoden

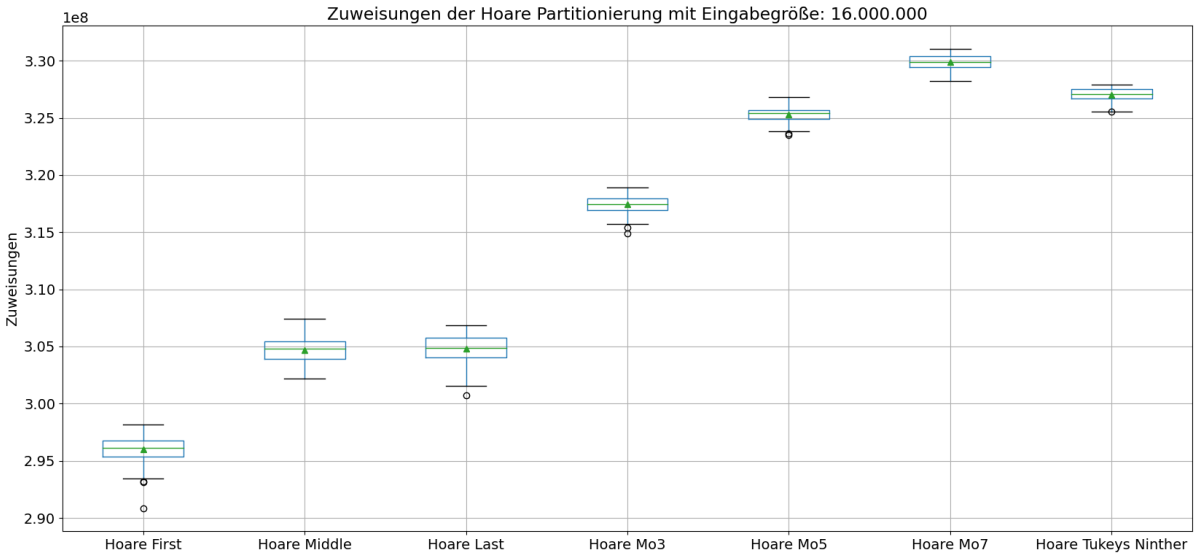

Abbildung 17: Zuweisungen Hoare Funktionen - Benchmarks aller Methoden

	Lomuto	Hoare
Pivotwahl	Mittelwerte der Zuweisungen	Mittelwerte der Zuweisungen
Erstes Element	992009373, 4	295975850, 9
Mittleres Element	769837209, 7	304700929, 7
Letztes Element	760441516, 3	304781703, 4
Median aus 3	670104953, 3	317419566, 1
Median aus 5	644402945, 6	325280078, 4
Median aus 7	638788338, 6	329857184, 4
Tukey's Ninther	630294748, 0	327032712, 4

Bei der Anzahl der Zuweisungen gibt es einen großen Unterschied zwischen beiden Partitionierungsalgorithmen. Die Lomuto Partitionierung benötigt generell deutlich mehr Zuweisungen, da die Elemente durch den Eingabearray geschoben werden, anstatt, wie bei Hoare, direkt getauscht werden.

Bei der Lomuto Partitionierung verringern sich die Zuweisungen, je mehr Elemente zur Pivotwahl benutzt werden, bei Hoare hingegen, steigen die Zuweisungen für diese Methoden.

Da beim Hoare Algorithmus das Pivotelement an den Anfang der Eingabe getauscht wird, bei der Pivotwahl des Ersten Elements dies damit schon richtig platziert ist, benötigt diese Methode hier deutlich weniger Zuweisungen als bei der Wahl des mittleren/letzten Element.

11 Experiment: Inklusion Pivotelement in Rekursion

In manchen Beispielen von Quicksort fällt auf, dass der Algorithmus so aufgebaut ist, dass das Pivotelement im Rekursionsaufruf mitgegeben wird.

Es wird dann meistens so implementiert, dass das Pivotelement in der Rekursion des rechten Teils mitgegeben wird.

Algorithm 12: Inklusion des Pivotelements

```
1 function quicksort(array, first, last)
2 |   pivotindex ← partition()
3 |
4 |   quicksort(array, first, pivotindex − 1)
5 |   quicksort(array, pivotindex, last)
6 end
```

Dies scheint auf den ersten Blick, hinsichtlich der Performanz von Quicksort, eher suboptimal, da durch die Inklusion des Pivotelements in jeder neuen Rekursionstiefe ein zusätzlicher Vergleich durchgeführt werden muss.

Die Auswirkungen der Inklusion des Pivotelement wird Experimentell untersucht, hierfür werden alle Methoden, die als Pivotelement einen Median bilden, einmal mit und einmal ohne Inklusion des Pivotelements miteinander verglichen.

11.1 Vergleich mit und ohne Inklusion

Alle Methoden wurden auf verschiedenen Input Größen getestet: 1.000, 5.000, 10.000, 50.000, 100.000, 250.000, 500.000, 1.000.000 und 16.000.000. Die nachfolgend abgebildeten Boxplots sind auf Basis der Eingabegröße 16.000.000.

11.1.1 Vergleiche

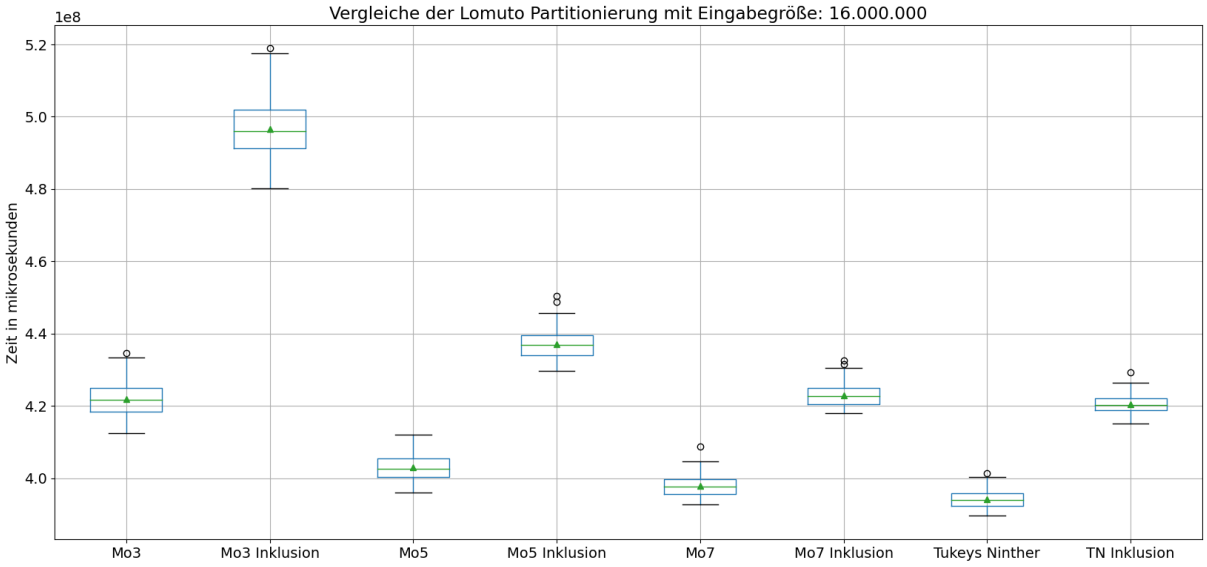

Abbildung 18: Vergleiche Lomuto Funktionen - Inklusion Pivotelement in Rekursion

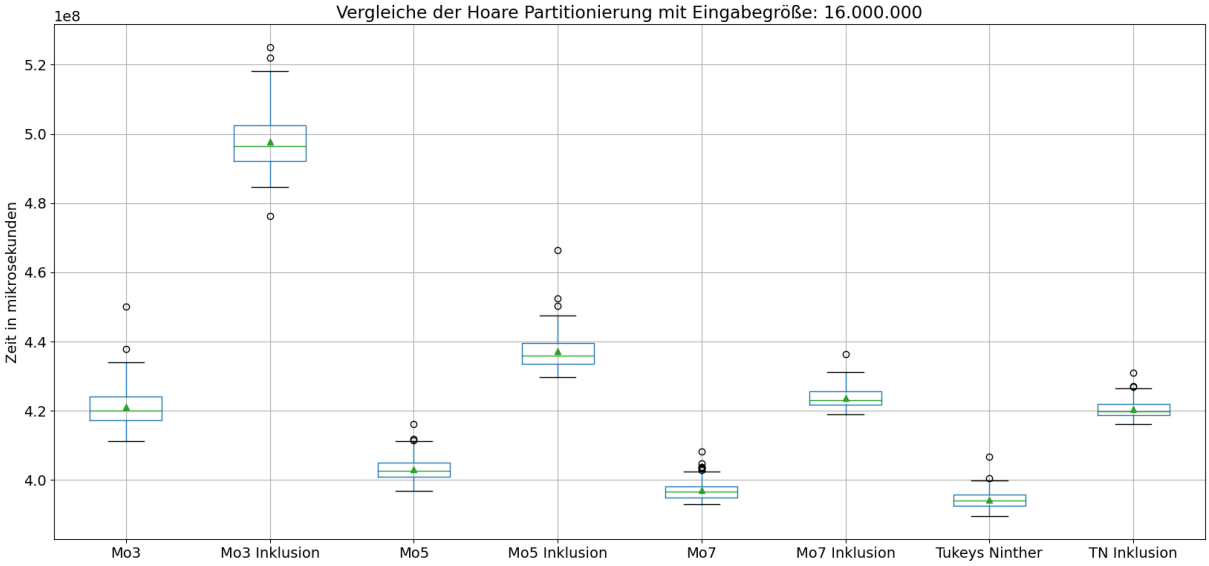

Abbildung 19: Vergleiche Hoare Funktionen - Inklusion Pivotelement in Rekursion

Wenn das Pivotelement in der Rekursion inkludiert wird, kommt es offensichtlich zu mehr vergleichen, als wenn es exkludiert werden würde. Dies ist auch deutlich in den Boxplots zu erkennen.

11.1.2 Laufzeit

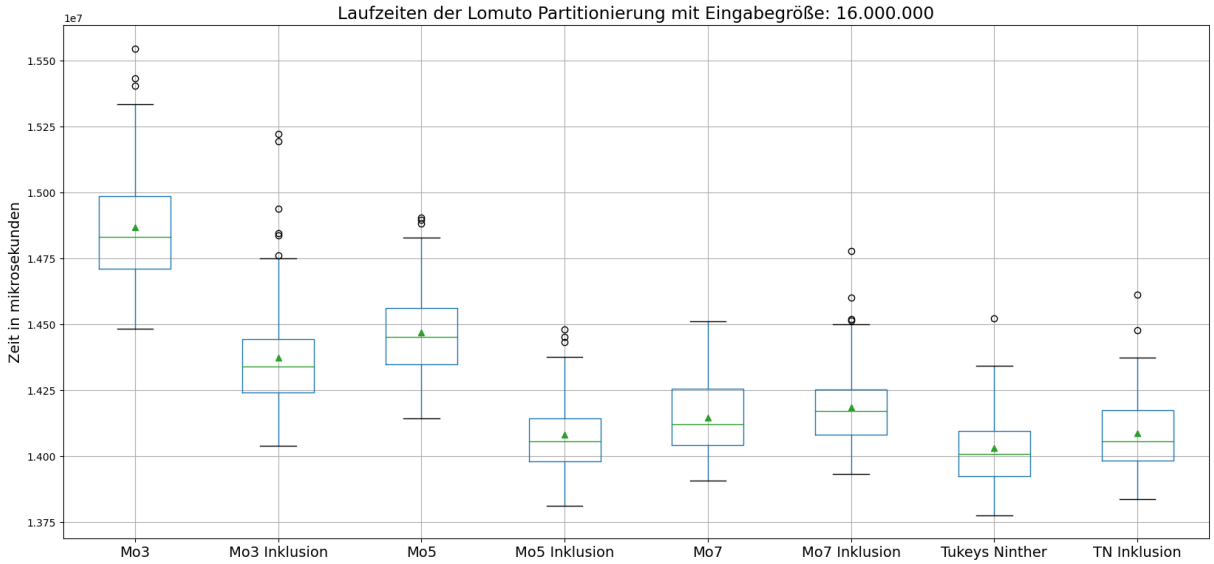

Abbildung 20: Laufzeit Lomuto Funktionen - Inklusion Pivotelement in Rekursion

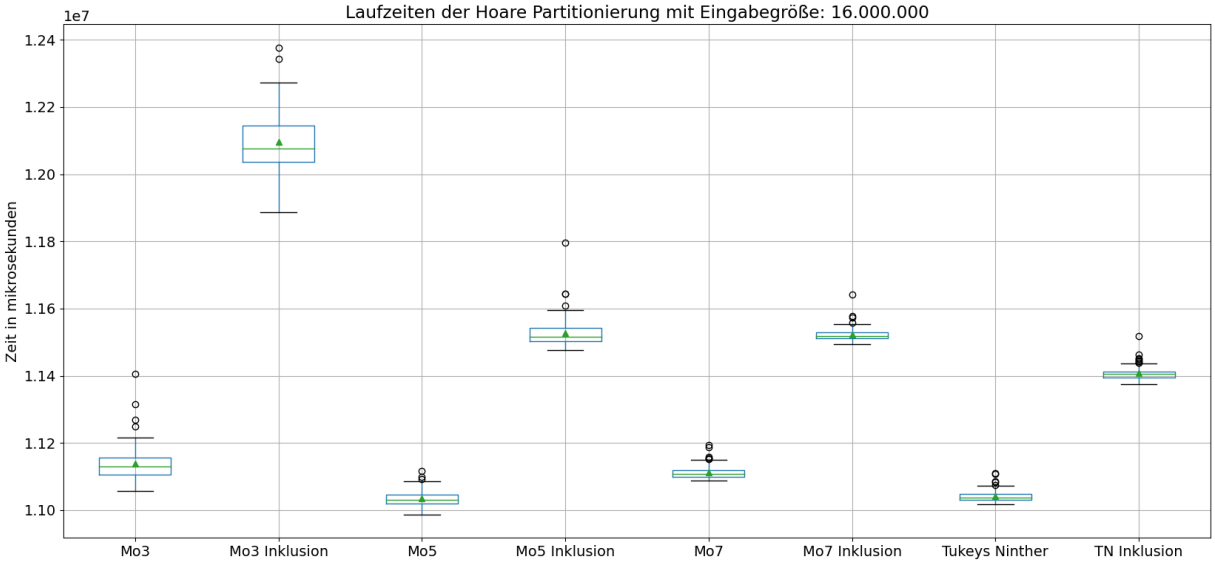

Abbildung 21: Laufzeit Hoare Funktionen - Inklusion Pivotelement in Rekursion

Hinsichtlich der Laufzeit sind die Ergebnisse sehr unterschiedlich, generell fällt auf, dass sich die Pivotinklusion bei der Hoare Partitionierung definitv nicht lohnt. Bei der Lomuto Partitionierung hingegen lohnt es sich mehr, je weniger Elemente für die Berechnung des Medians benutzt werden.

Für Median aus 7 und Tukey's Ninther ergibt sich eine längere Laufzeit.

11.1.3 Zuweisungen

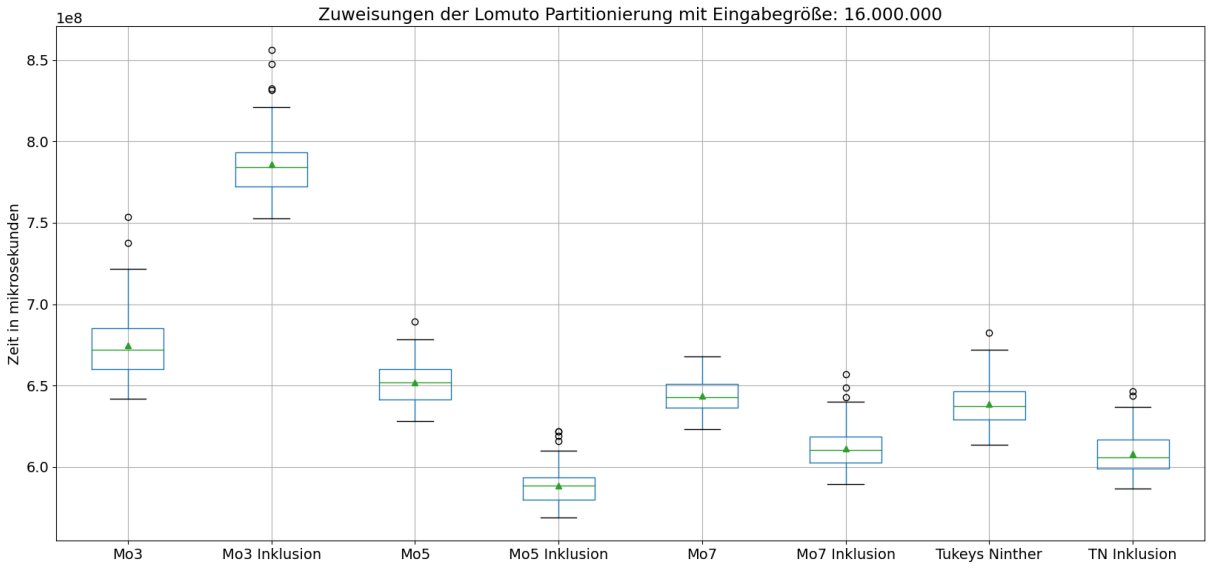

Abbildung 22: Zuweisungen Lomuto Funktionen - Inklusion Pivotelement in Rekursion

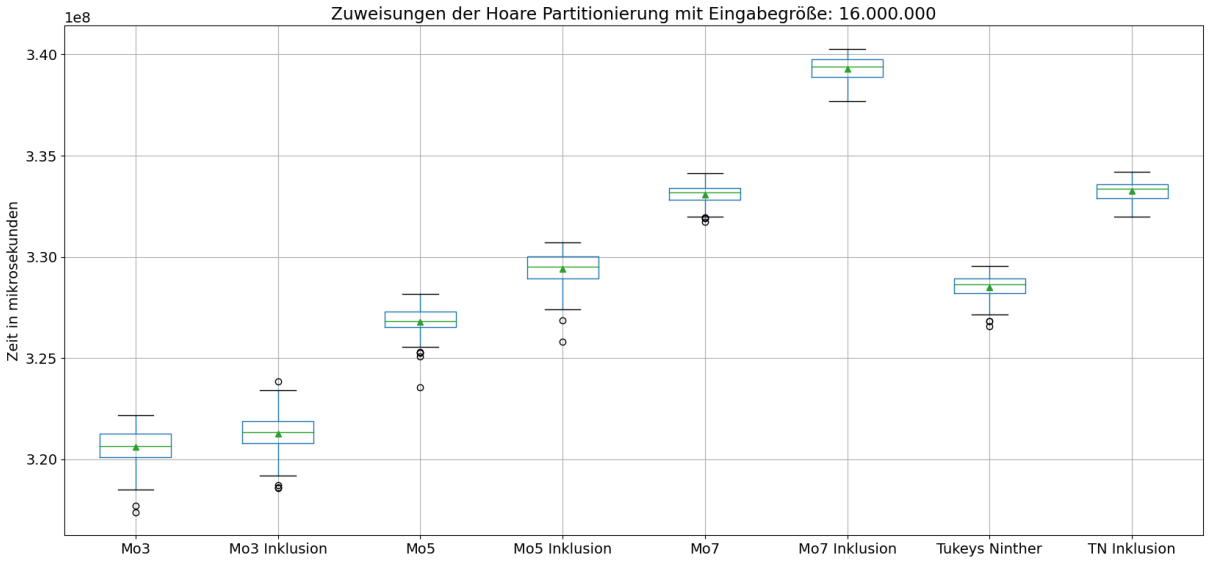

Abbildung 23: Zuweisungen Hoare Funktionen - Inklusion Pivotelement in Rekursion

Für die Anzahl der Zuweisungen ergibt sich ein ähnliches Bild wie bei der Laufzeit, wenn die Hoare Partitionierung benutzt wird, braucht jede Methode mehr Zuweisungen, als bei

der jeweiligen Variante ohne Inklusion des Pivotelements.

Bei der Lomuto Partitionierung ergeben sich wieder unterschiedliche Ergebnisse, mit ausnahme von Median aus 3 führen alle Methoden weniger Zuweisungen aus.

11.1.4 Zusammenfassung

Generell lohnt es sich nicht, das Pivotelement zu inkludieren, da hinsichtlich allen Punkten die jeweils „beste“ Variante eine Hoare Partitionierung ohne Pivot Inklusion ist.

Allerdings kann es je nach persönlicher Implementierung und Pivotwahl auf jeden Fall Sinn machen, die Inklusion des Pivotelements zu testen, da es für manche Szenarien performanter ist.

12 Worst Case Eingaben

Für Pivotwahlmethoden wie Erstes/Letztes Element ist es ziemlich offensichtlich, wie eine Worst-/Bad-Case Eingabe aussieht, es muss lediglich eine vorsortierte Liste als Eingabe gewählt werden, um eine quadratische Anzahl an Vergleichen zu provozieren.

Andere Pivotwahlmethoden performen auf Vorsortierten Listen jedoch ganz normal. Deshalb sollen spezielle Eingaben für weitere Pivotwahlen konstruiert und getestet werden.

12.1 Mittleres Element und Median aus 3

Die Eingaben für Mittleres Element als Pivot und Median aus 3 werden zusammengefasst, da sie sehr ähnlich sind.

Die Erste Hälfte der Eingabe besteht aus allen geraden Zahlen, welche kleiner als die Eingabegröße sind. Die zweite Hälfte sind alle Ungeraden Zahlen, welche abwechselnd angeordnet sind (siehe Abbildung 24 und Pseudcode). Je nach Partitionierungsverfahren wird der Algorithmus etwas angepasst.

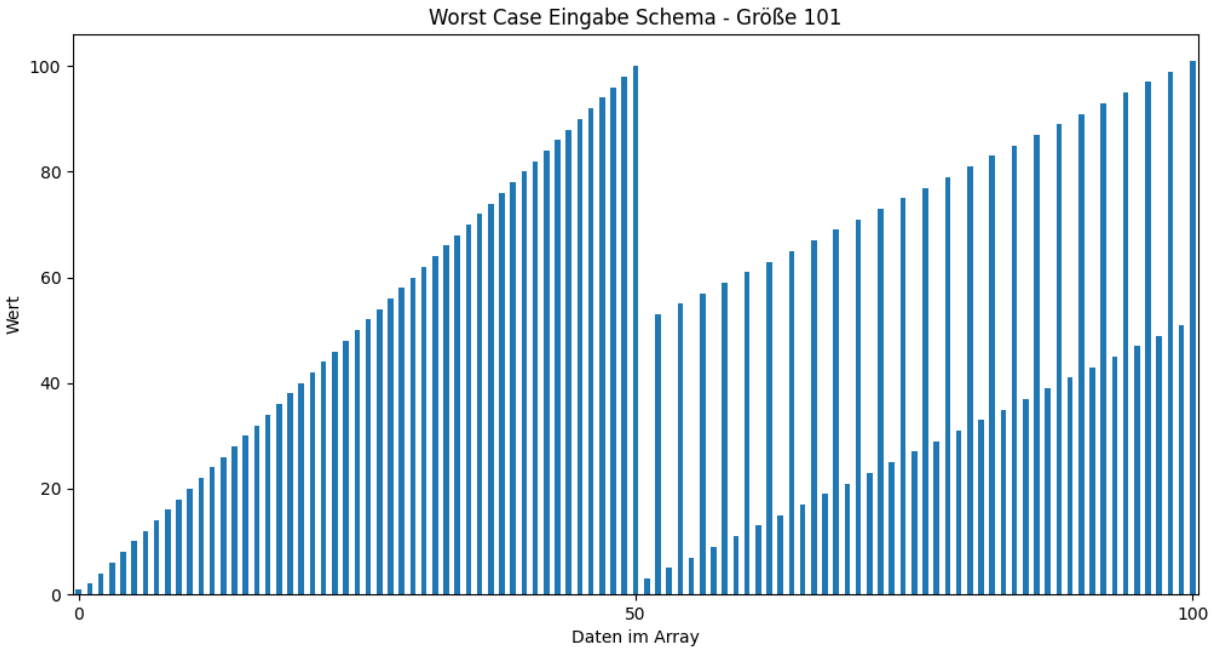

Abbildung 24: Worst Case Schema für Mittleres Pivot und Median aus 3

Algorithm 13: Worst Case Generator für Median aus 3

```
1  function worstCaseGenerator(array, size)          // Size should be odd
2
3      for i = 2 to i = lastEvenElement with i ← i + 2 do
4          array.append(i)
5      end
6      array.append(size)
7      rest ← (size−1)/2
8      if rest % 2 == 0 then
9          for i = 0 to rest do
10             if i % 2 == 0 then
11                 array.append(i + 3)
12             else
13                 array.append(rest + 2 + i)
14             end
15         end
16     else
17         for i = 0 to rest do
18             if i % 2 == 0 then
19                 array.append(rest + 2 + i)
20             else
21                 array.append(i + 2)
22             end
23         end
24     end
25 end
```

12.2 Auswirkungen

Die Y-Achsen sind logarithmisch dargestellt. Alle Methoden wurden mit einer Eingabegröße von 10.000 getestet.

12.2.1 Aufsteigende Liste

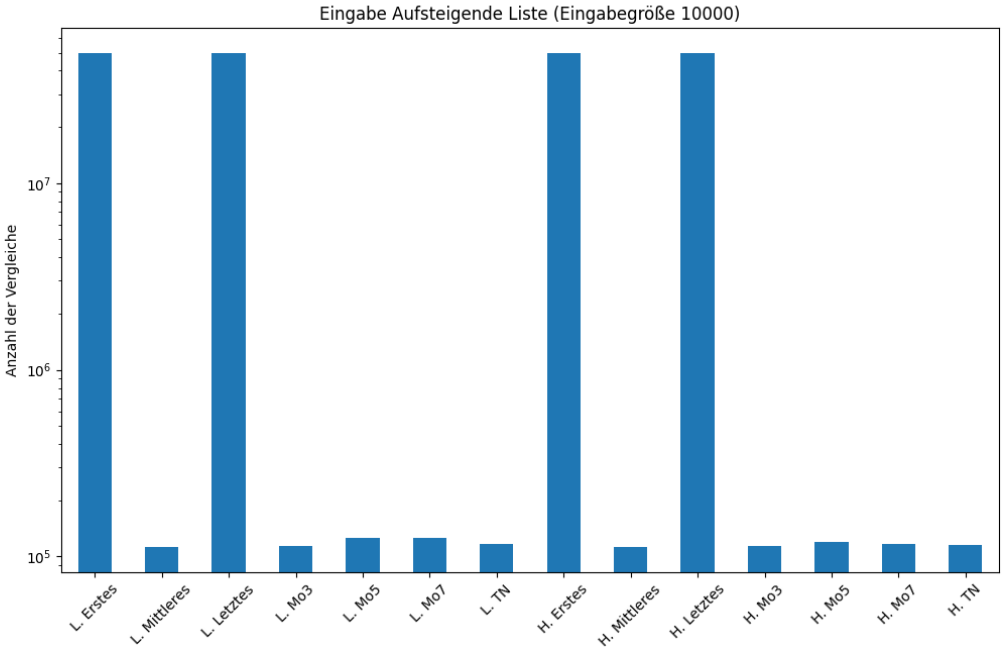

Abbildung 25: Vergleiche - Aufsteigende Liste

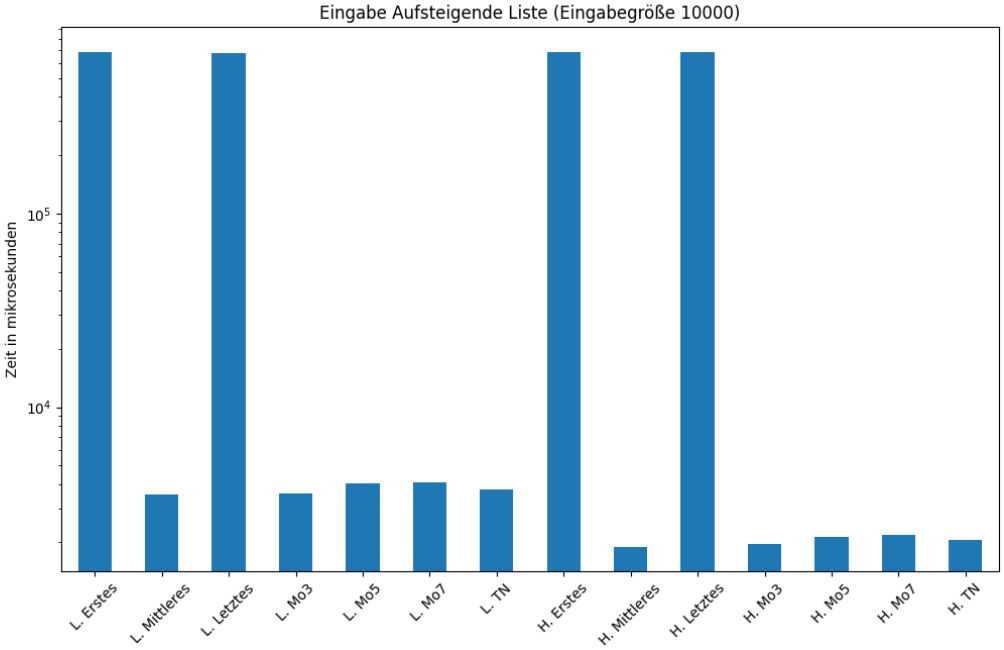

Abbildung 26: Laufzeit - Aufsteigende Liste

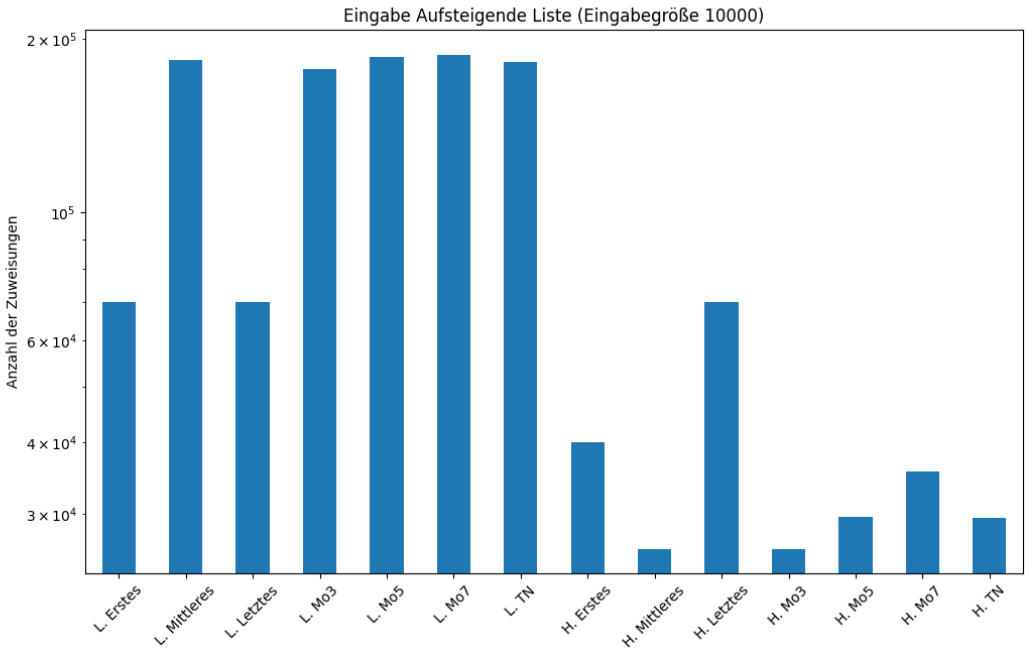

Abbildung 27: Zuweisungen - Aufsteigende Liste

12.2.2 Zusammenfassung

Wie erwartet performen die Methoden, welche das Erste oder Letzte Element als Pivot wählen, schlecht bei Eingabe einer vorsortierten Liste. Die beiden Methoden benötigen ungefähr $0.5n^2$ Vergleiche, um die Eingabe durchzugehen.

Dementsprechend ist auch die Laufzeit für beide Methoden enorm höher, als für alle anderen Methoden. Für die restlichen Methoden sind keine Auswirkungen zu sehen.

12.2.3 Absteigende Liste

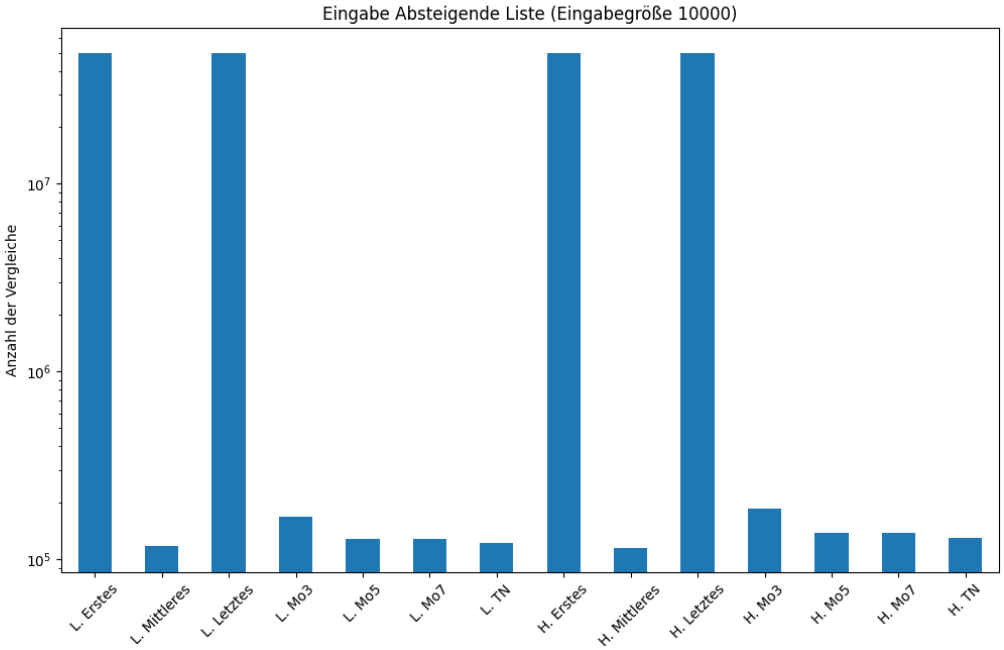

Abbildung 28: Vergleiche - Absteigende Liste

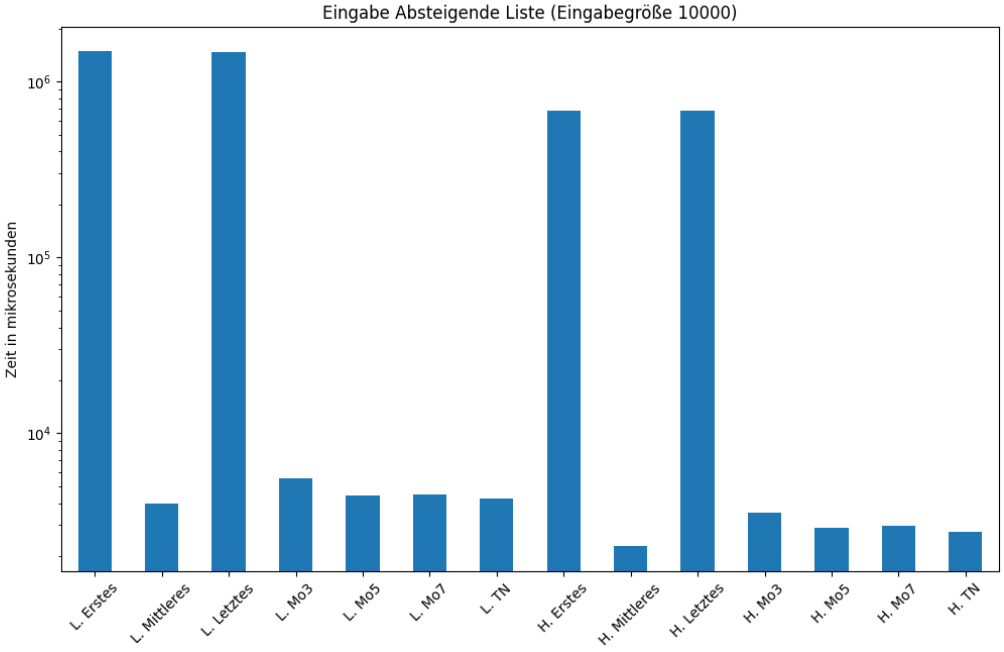

Abbildung 29: Laufzeit - Absteigende Liste

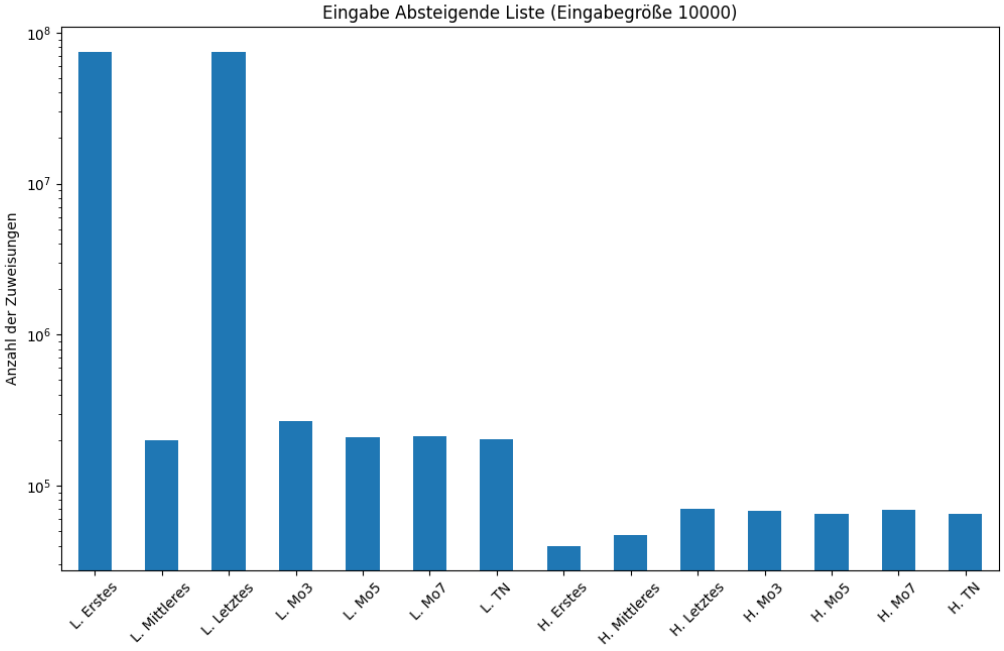

Abbildung 30: Zuweisungen - Absteigende Liste

12.2.4 Zusammenfassung

Bei der Eingabe einer absteigend sortierten Liste zeichnet sich quasi das gleiche Bild, wie bei der aufsteigend sortierten Liste. Die Methoden, welche das Erste oder Letzte Element als Pivotelement wählen, performen hier wieder sehr schlecht, da immer das größte/kleinste Element als Pivotelement gewählt wird.

Die Anzahl an Vergleiche, beträgt auch hier ca. $0.5n^2$ Vergleiche, was wieder ungefähr den Worst Case dargestellt.

12.2.5 Schlechte Eingabe für Lomuto Mittleres Element

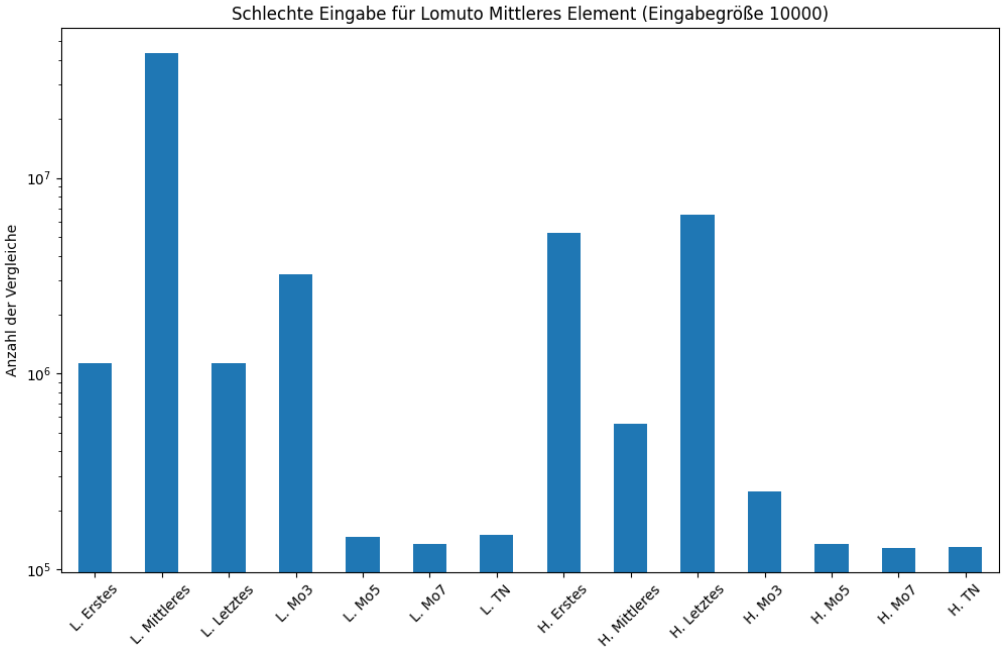

Abbildung 31: Vergleiche - Schlechte Eingabe für Lomuto Mittleres Element

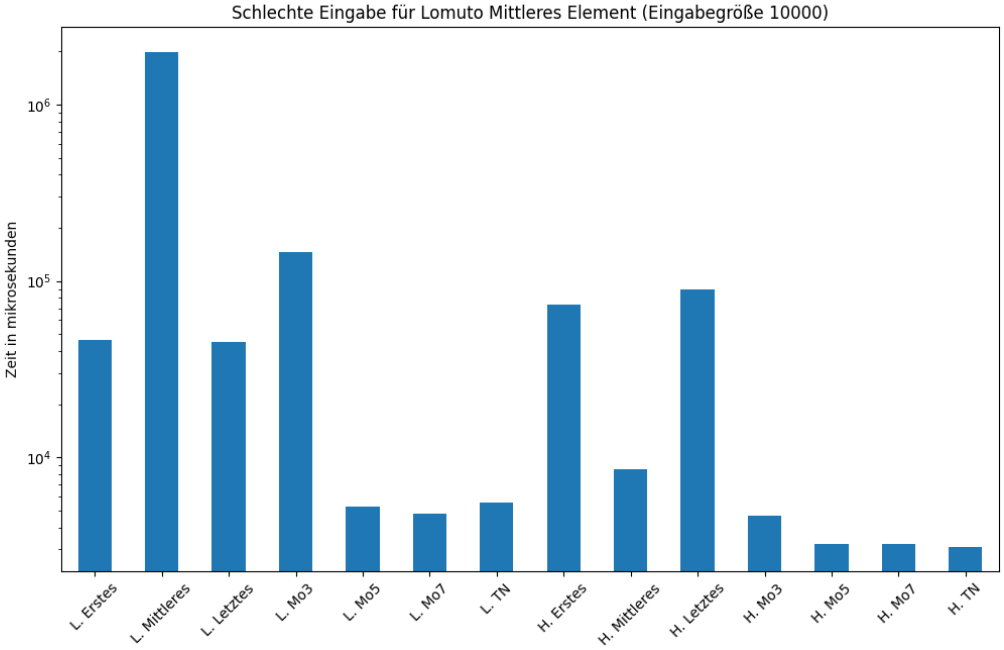

Abbildung 32: Laufzeit - Schlechte Eingabe für Lomuto Mittleres Element

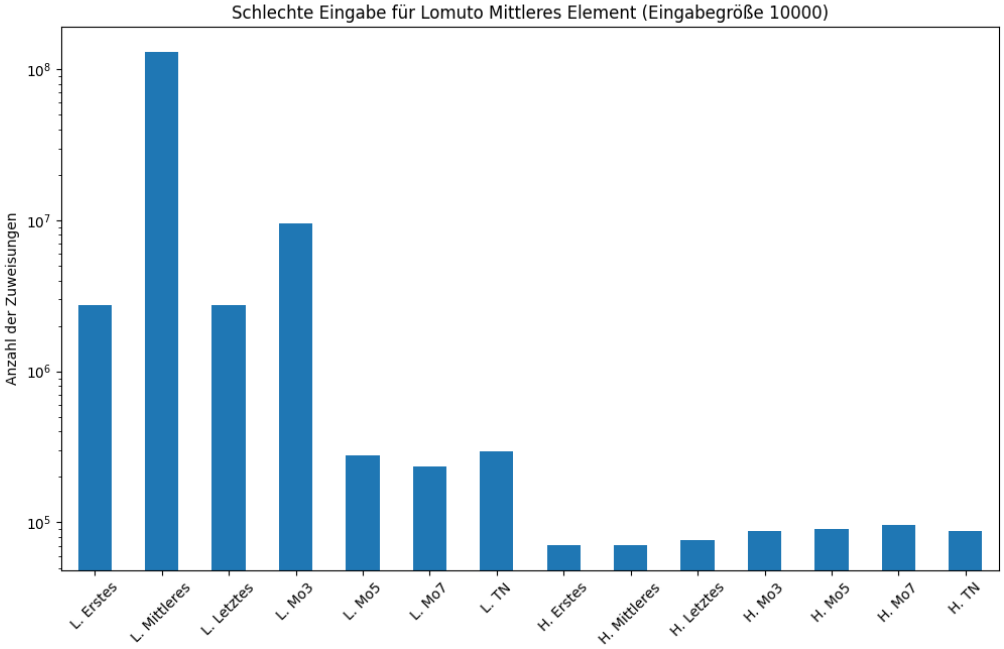

Abbildung 33: Zuweisungen - Schlechte Eingabe für Lomuto Mittleres Element

12.2.6 Zusammenfassung

Da die Eingabe speziell für die Pivotwahl des Mittleren Elements mit Lomuto Partitionierung erstellt wurde, überrascht es nicht, dass diese hier auch am schlechtesten abschneidet. Hierbei werden ca. $0.45n^2$ Vergleiche durchgeführt.

Außerdem schneiden auch die Methoden mit erstem oder letzten Element, sowie Lomuto Median aus 3 deutlich schlechter ab als im Average Case.

12.2.7 Schlechte Eingabe für Hoare Mittleres Element

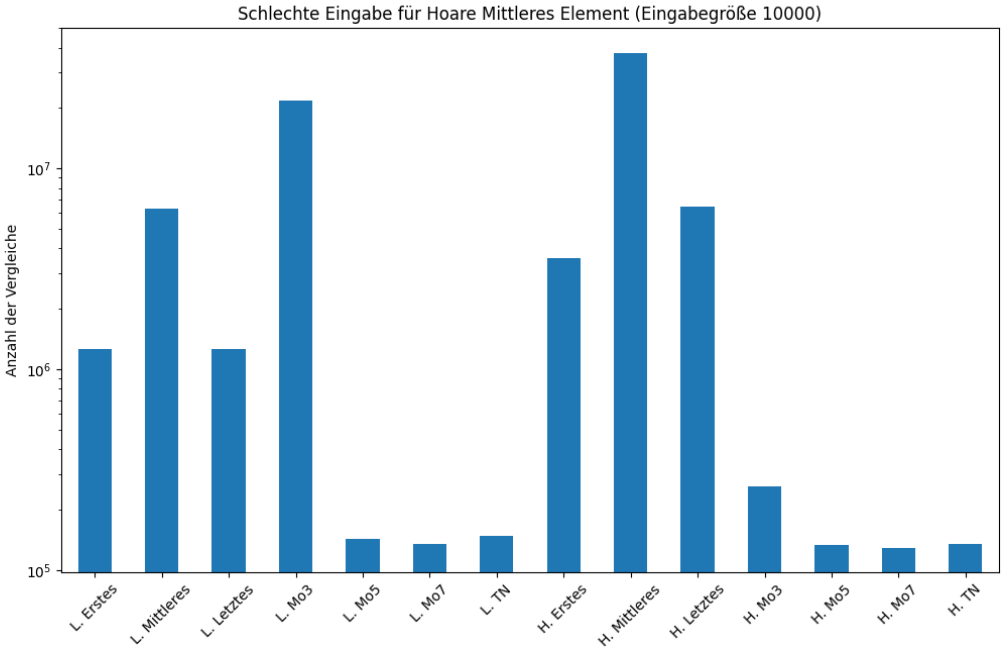

Abbildung 34: Vergleiche - Schlechte Eingabe für Hoare Mittleres Element

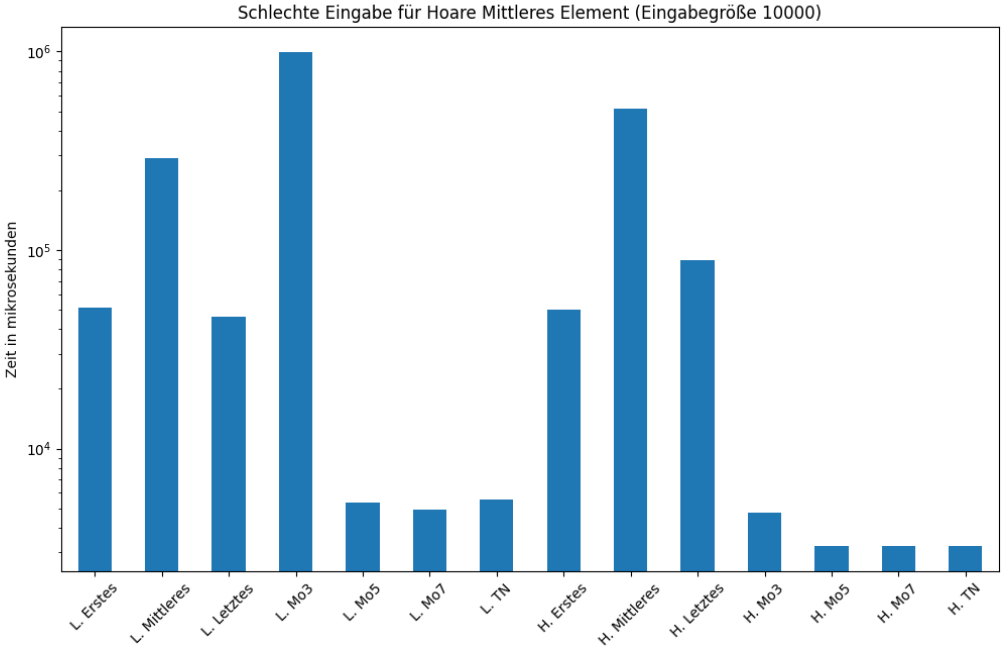

Abbildung 35: Laufzeit - Schlechte Eingabe für Hoare Mittleres Element

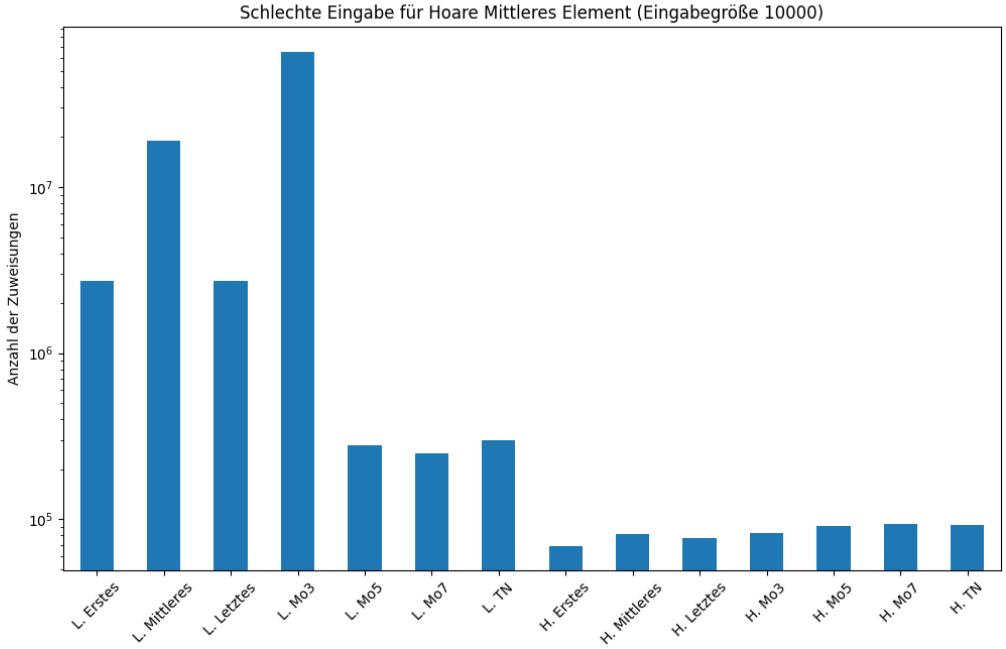

Abbildung 36: Zuweisungen - Schlechte Eingabe für Hoare Mittleres Element

12.2.8 Zusammenfassung

Auch hier sind die Ergebnisse wieder sehr ähnlich zu den Vorherigen. Offensichtlich performt die Hoare Partitionierung mit mittlerem Element als Pivot am schlechtesten, wieder mit etwa $0.45n^2$ Vergleichen.

Die Vergleiche und Laufzeiten der Methoden mit erstem oder letztem Element als Pivot und Lomuto Median aus 3 sind auch hier wieder deutlich erhöht.

12.2.9 Schlechte Eingabe für Median aus 3

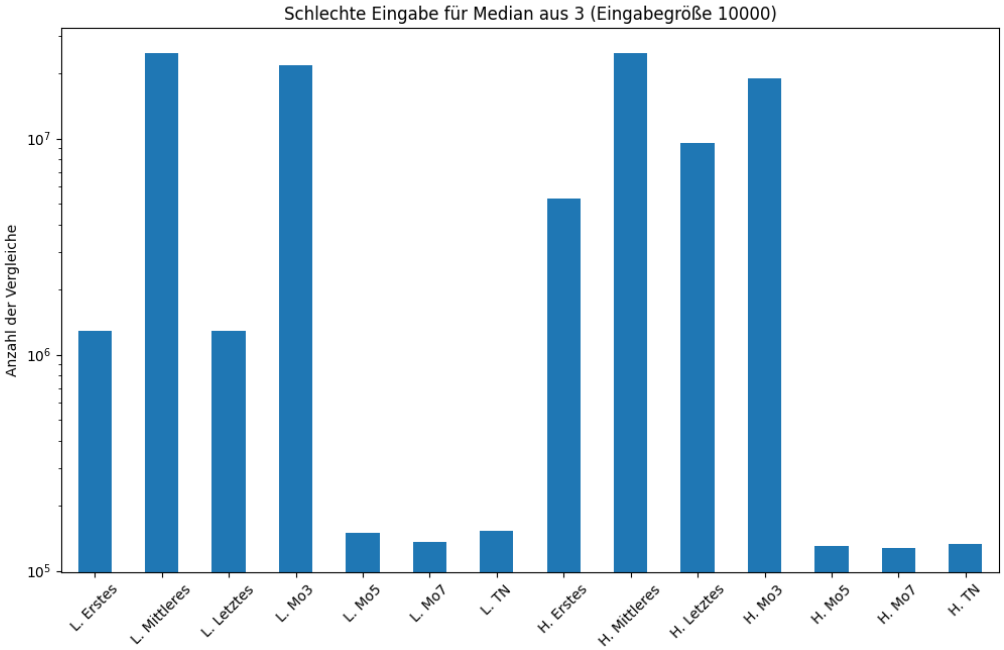

Abbildung 37: Vergleiche - Schlechte Eingabe für Median aus 3

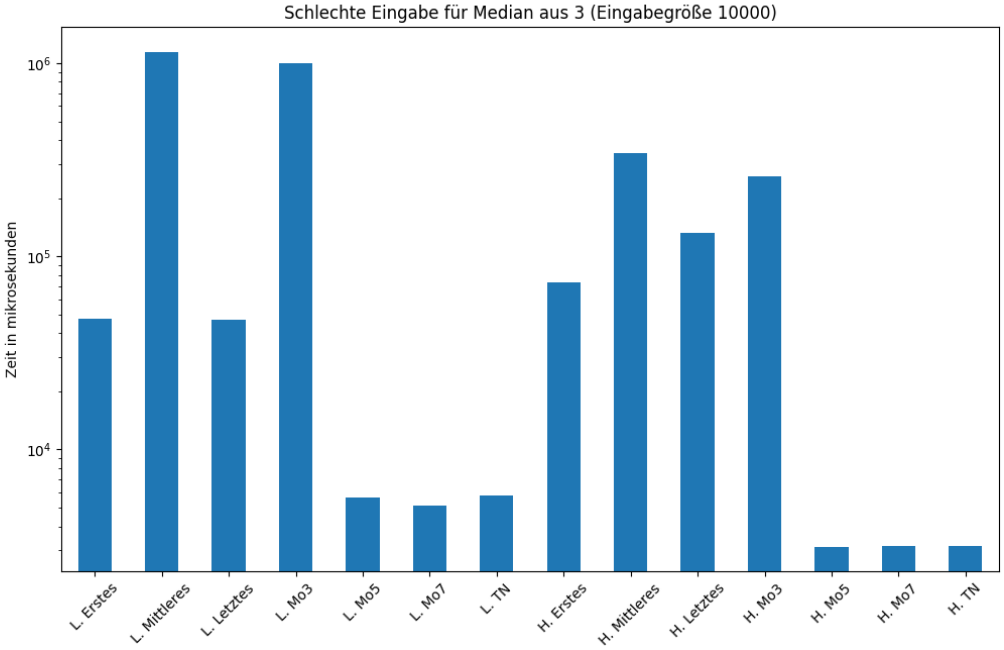

Abbildung 38: Laufzeit - Schlechte Eingabe für Median aus 3

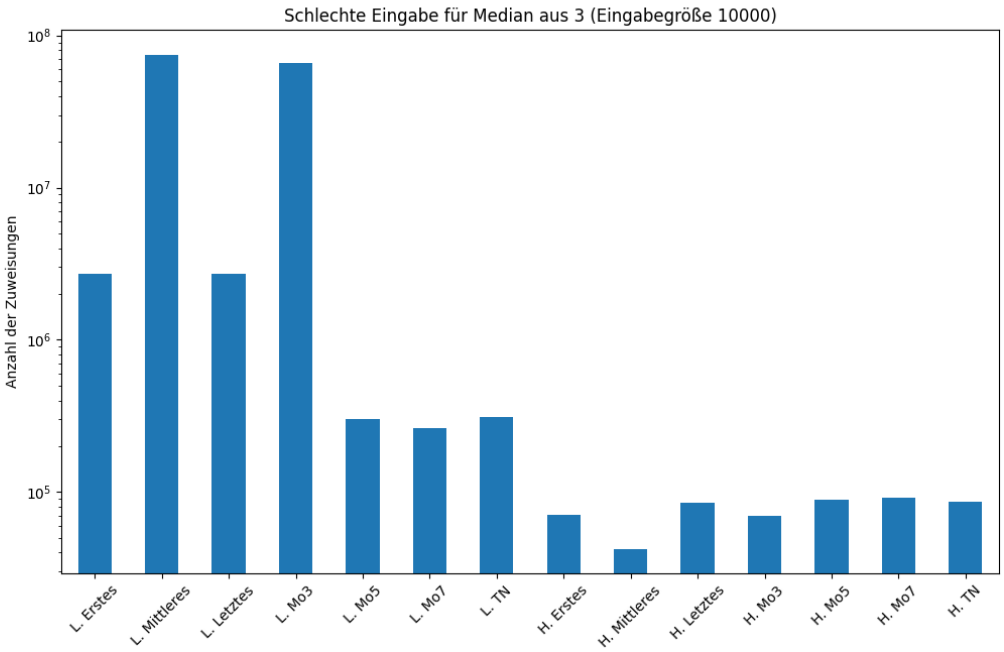

Abbildung 39: Zuweisungen - Schlechte Eingabe für Median aus 3

12.2.10 Zusammenfassung

Bei der Eingabe der „Killersequenz“ für Median aus 3 fällt auf, dass viele Methoden relativ schlecht performen. Da das Schema der Eingabe dem für Hoare/Lomuto Mittleres Element stark ähnelt performen diese hier auch ziemlich schlecht. Für diese beiden Pivotwahlen ergeben sich ungefähr $0.25n^2$ Vergleiche.

Da die Eingabe auch teilweise sortiert ist, benötigen auch die Methoden, welche das Erste und Letzte Element benutzen deutlich mehr Vergleiche als bei einer zufälligen Eingabe.

13 Pivotwahl mit Randomisierung

Um die Auswirkungen von Randomisierung der Pivotwahl zu testen, werden die vorhandenen Methoden zur Pivotwahl, in Verbindung mit Randomisierung getestet.

13.1 Methoden

13.1.1 Zufälliges Element

Als Pivotelement wird hier ein zufälliges Element aus der Eingabeliste gewählt.

13.1.2 Median aus 3 Randomisiert

Der Median aus 3 wird nichtmehr aus dem Ersten, Mittleren und Letzten Element gebildet, sondern aus drei zufälligen Werten der Eingabe.

13.1.3 Median aus 5 Randomisiert

Die komplett randomisierte Median aus 5 Methode berechnet den Median aus fünf zufälligen Elementen der Eingabe.

13.1.4 Median aus 7 Randomisiert

Die Median aus 7 Teil Randomisierung benutzt als feste Eingabe für den Median das Erste, Mittlere und Letzte Element. Die anderen beiden Elemente werden zufällig ausgewählt.

13.1.5 Tukey's Ninther Randomisiert

Der randomisierte Tukey's Ninther benutzt auch das Erste, Mittlere und Letzte Element als feste Eingabe. Zusätzlich werden sechs zufällige Elemente zur Bestimmung des Medians benutzt.

13.2 Vergleich zur Pivotwahl ohne Randomisierung

Die Methoden werden 100 mal auf einer zufälligen Eingabe der Größe 1.000.000 aufgerufen und mit den Methoden ohne Randomisierung verglichen.

13.2.1 Vergleiche

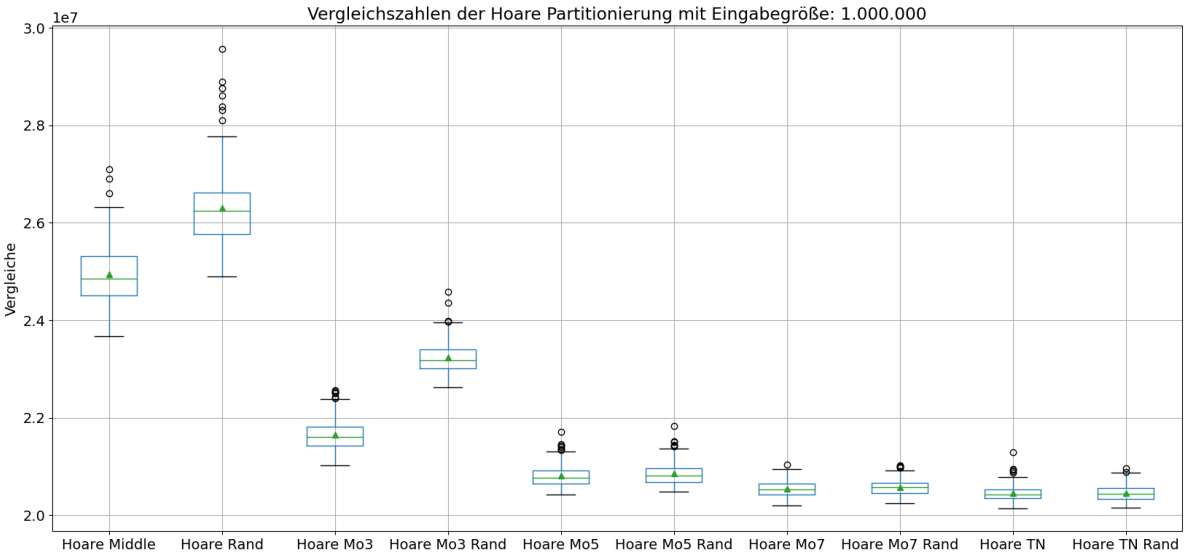

Abbildung 40: Vergleiche - Randomisierte und Normale Methoden

13.2.2 Laufzeit

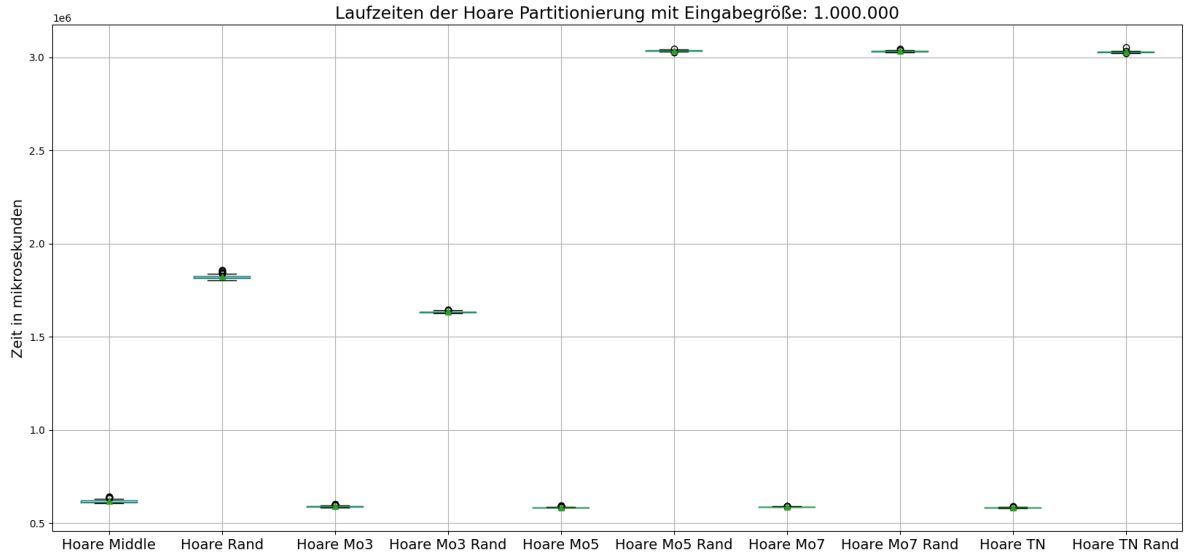

Abbildung 41: Laufzeit - Randomisierte und Normale Methoden

13.2.3 Zuweisungen

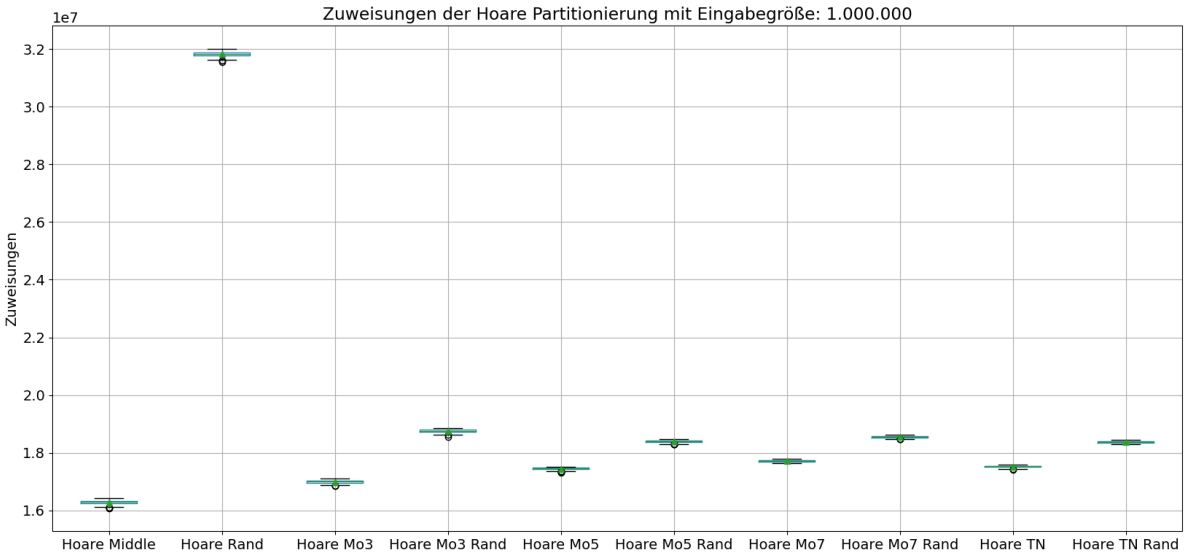

Abbildung 42: Zuweisungen - Randomisierte und Normale Methoden

13.3 Performanz bei speziellen Eingaben

Zuvor wurde gezeigt, dass durch speziell konstruierte Eingaben, bei verschiedenen Methoden eine extrem schlechte/Worst Case Performanz provoziert werden kann. Nun werden die Methoden mit Randomisierung auf den selben Eingaben getestet.

13.3.1 Aufsteigend Sortierte Liste

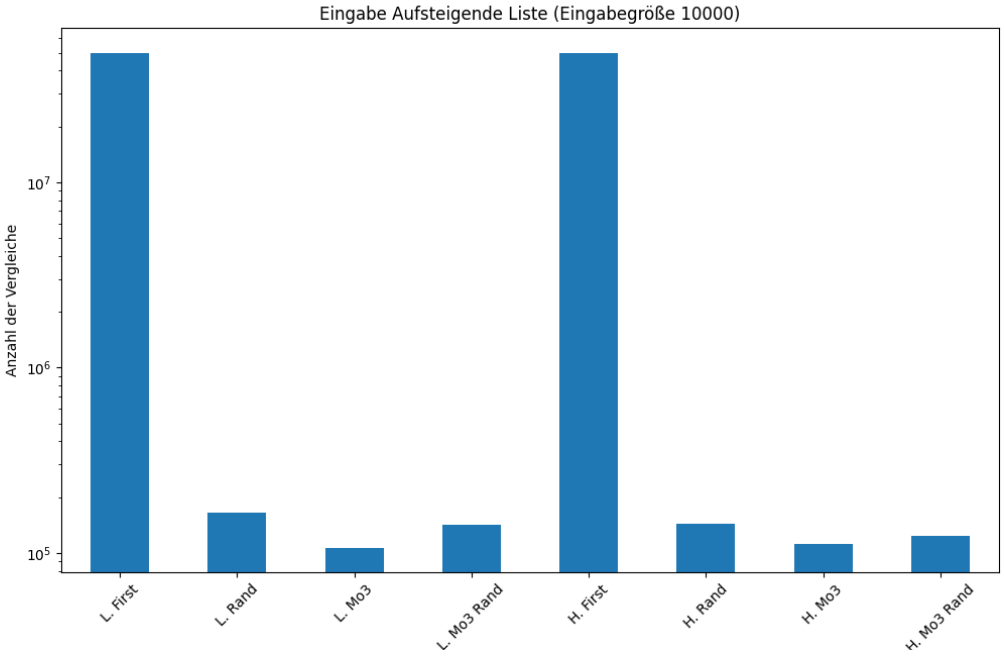

Abbildung 43: Vergleiche - Sortierte Liste

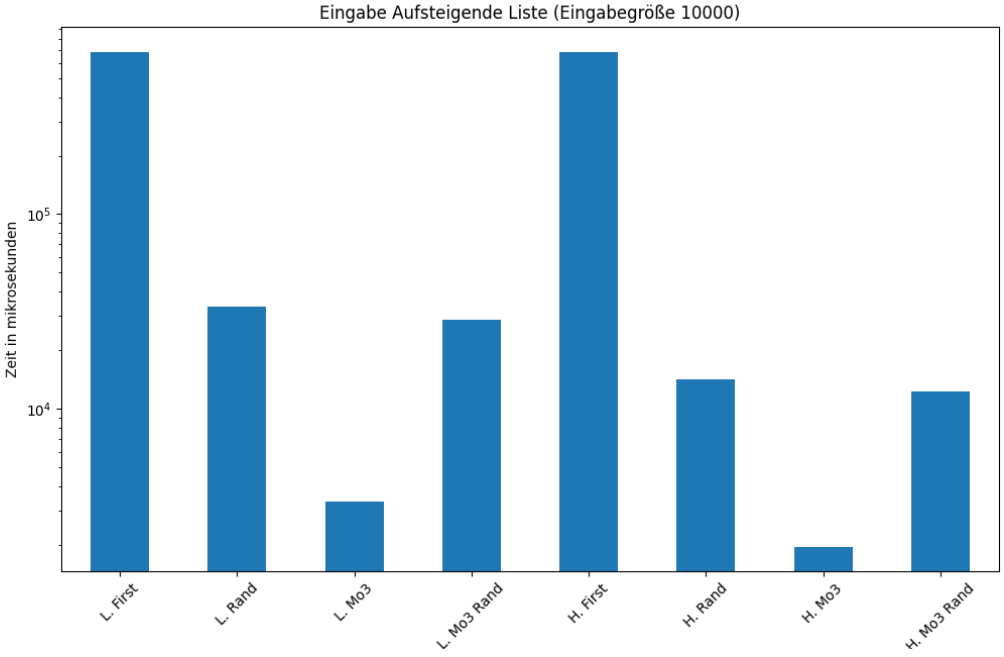

Abbildung 44: Laufzeit - Sortierte Liste

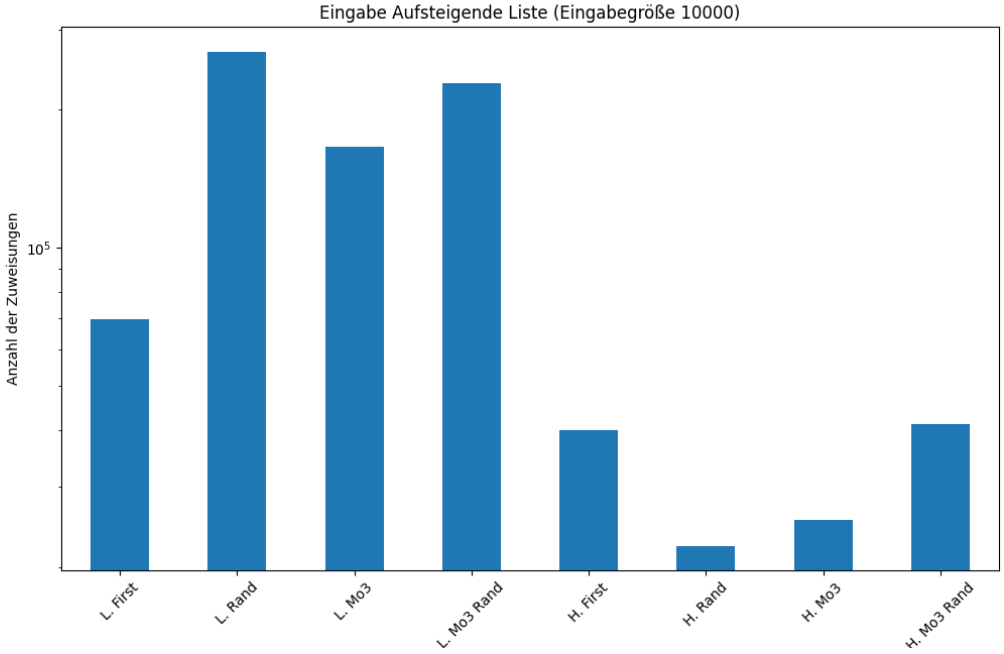

Abbildung 45: Zuweisungen - Sortierte Liste

13.3.2 Spezielle Eingabe für Median aus 3 und Mittleres Element

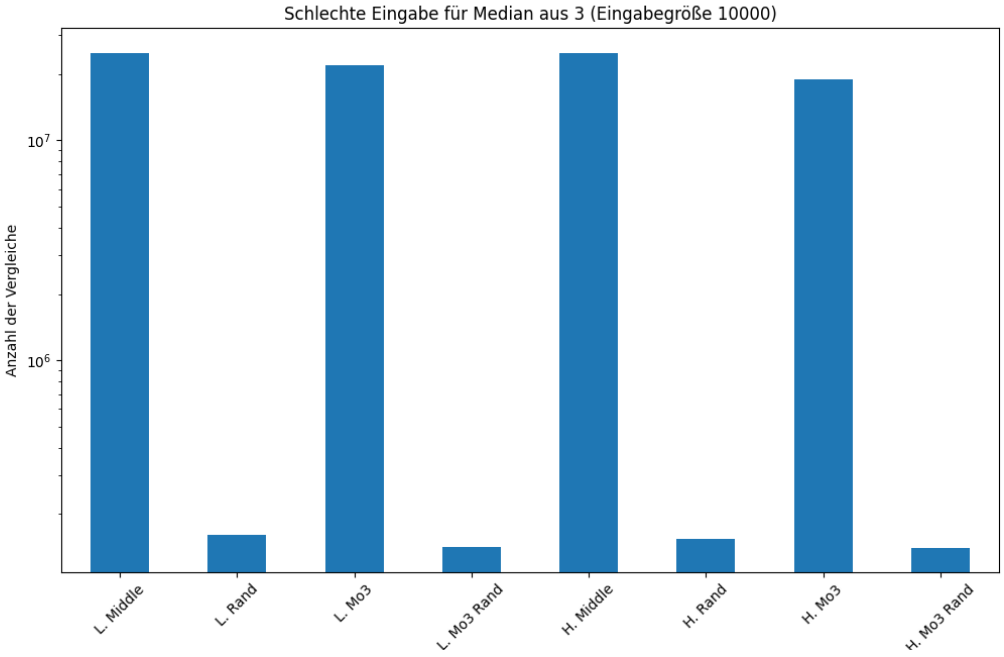

Abbildung 46: Vergleiche - Eingabe für Median aus 3 und Mittleres Element

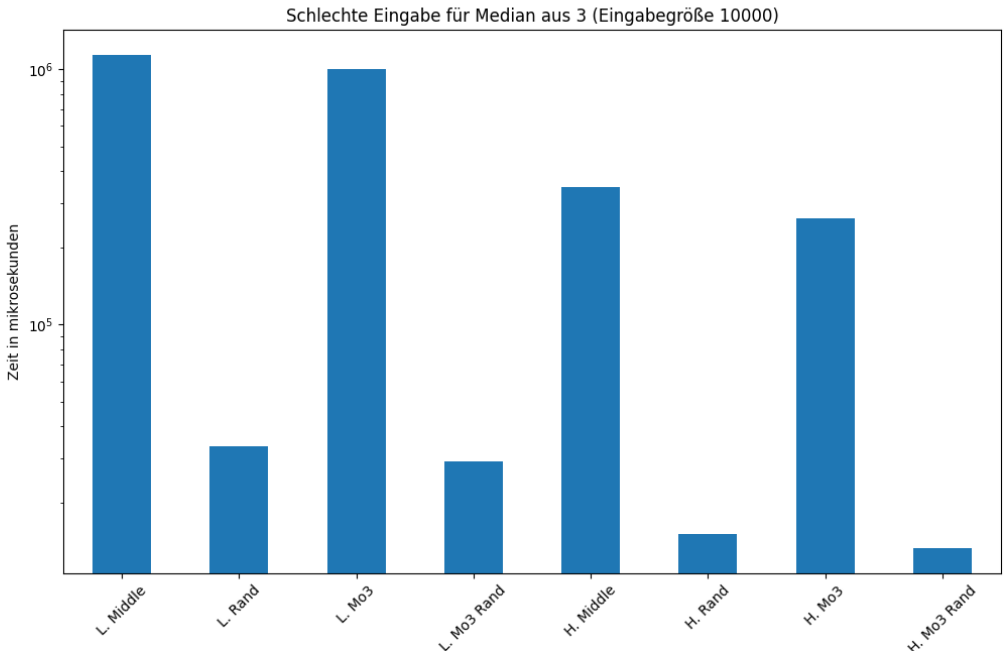

Abbildung 47: Laufzeit - Eingabe für Median aus 3 und Mittleres Element

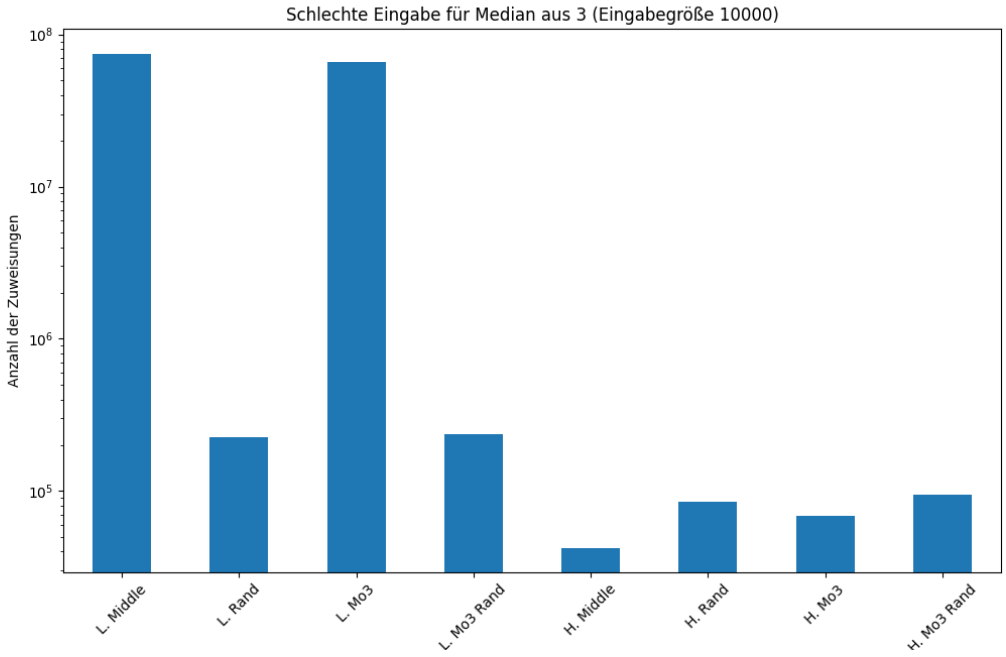

Abbildung 48: Zuweisungen - Eingabe für Median aus 3 und Mittleres Element

13.4 Fazit Randomisierung

Wie erwartet, performen die Methoden mit Randomisierung wesentlich besser auf die Worst Case Eingabe. Durch die randomisierte Wahl der Elemente ist das Muster nicht mehr wirksam, weswegen sie wie auf einer Zufälligen Eingabe performen.

Bei zufälliger Eingabe fällt auf, dass die Erzeugung der randomisierten Indizes sehr kostspielig ist. Hinsichtlich der Vergleichszahlen und der Zuweisungen sind die Methoden nah beieinander, jedoch benötigen die randomisierten Methoden sehr viel mehr Laufzeit.

14 Schlussfolgerung und Ausblick

Abschließend ist festzuhalten, dass bei der Implementierung von Quicksort definitiv das Hoare Partitionierungs Schema benutzt werden sollte, da es deutlich schneller ist und viel weniger Zuweisungen macht.

Bei der Frage nach der Pivotwahl sollte immer der individuelle Anwendungszweck berücksichtigt werden. Wenn die Eingabegröße nicht sehr groß ist (kleiner 5.000 hier), ist durch eine komplexere Pivotwahl kein Vorteil gegeben. Dennoch muss beachtet werden, dass einfache Pivotwahlen, wie die Wahl des ersten/letzten Elements sehr anfällig gegenüber vorsortierten Listen sind, weshalb sich auch bei kleineren Eingabe die Wahl des Median aus 3 Schemas anbietet. Hier ist es quasi nicht möglich, dass zufällige oder normale Eingaben einen Worst Case darstellen.

Bei sehr großen Eingaben (größer 100.000) sollte eine komplexere Pivotwahl gewählt werden, hierbei performt Tukey's Ninther insgesamt am besten. Dabei sollte dann beachtet werden, dass ab einer Eingabegröße von 100-200 auf Median aus 3 und bei sehr kleinen Eingaben (< 10) auf Insertion Sort gewechselt wird, um die Performanz nochmal zu verbessern.

Der Wechsel auf randomisierte Methoden lohnt sich nicht für zufällige Eingaben. Je nach Muster der zu sortierenden Eingabe kann dies aber auch eine verbesserte Performanz bringen.

Die Inklusion des Pivotelements hat bei der Lomuto Partitionierung eine Verbesserung der Laufzeit bewirkt, jedoch bei der ohnehin schnelleren Hoare Partitionierung wurde die Laufzeit dadurch verschlechtert.

15 Weitere Benchmarks

Die Benchmarks aller Methoden wurden für verschiedene Eingabegrößen durchgeführt. Da bisher nur die Ergebnisse für Eingabegröße 16 Millionen dargestellt waren, sind hier die Restlichen Ergebnisse aufgelistet.

15.1 Eingabegröße 1.000

15.1.1 Vergleiche

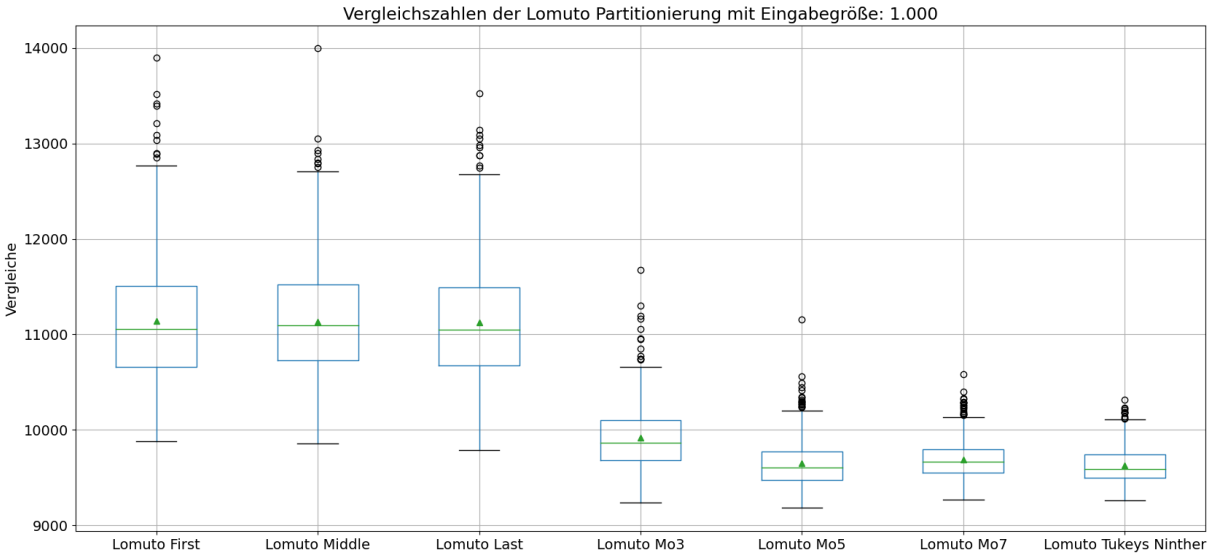

Abbildung 49: Vergleiche Lomuto Funktionen - Benchmarks aller Methoden

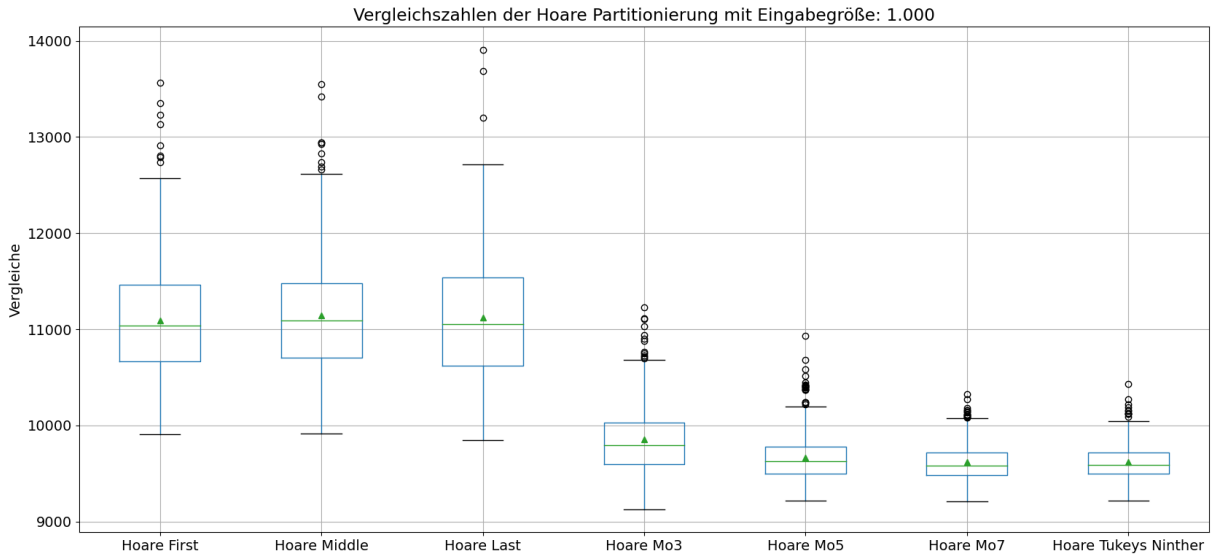

Abbildung 50: Vergleiche Hoare Funktionen - Benchmarks aller Methoden

15.1.2 Laufzeit

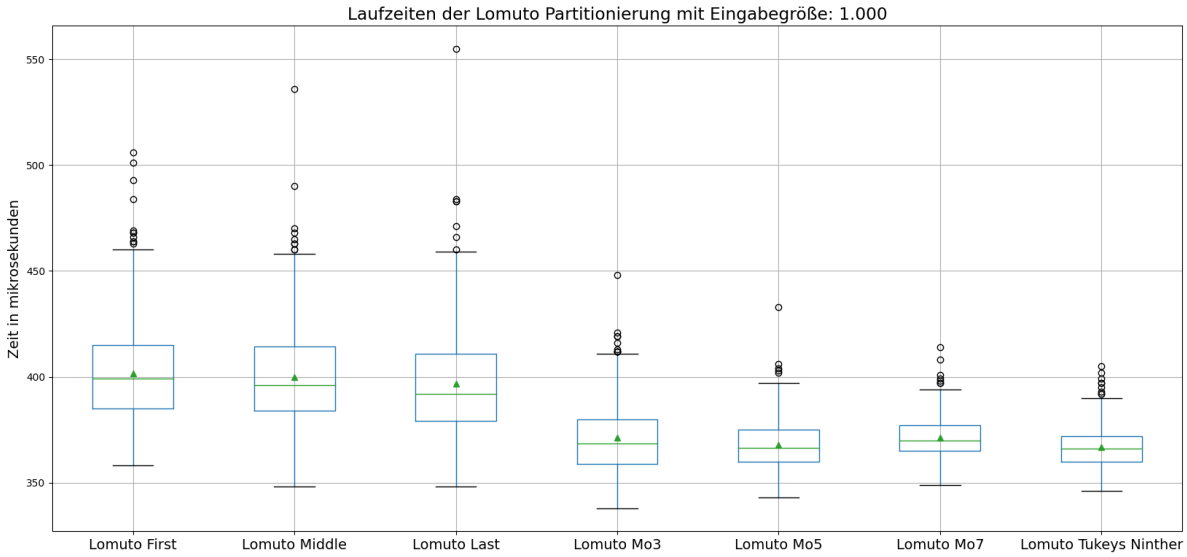

Abbildung 51: Laufzeit Lomuto Funktionen - Benchmarks aller Methoden

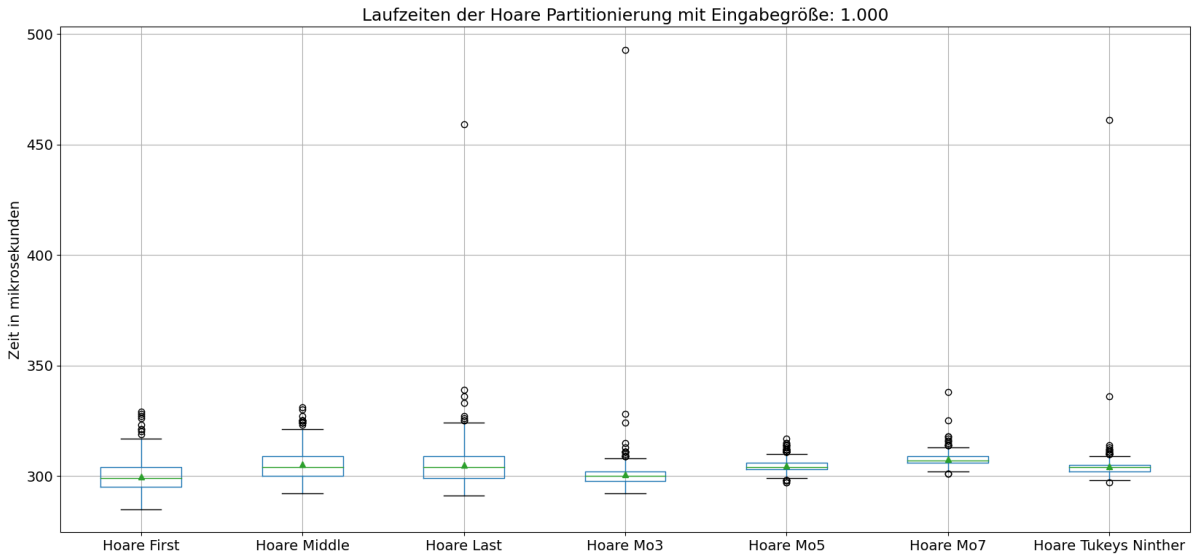

Abbildung 52: Laufzeit Hoare Funktionen - Benchmarks aller Methoden

15.1.3 Zuweisungen

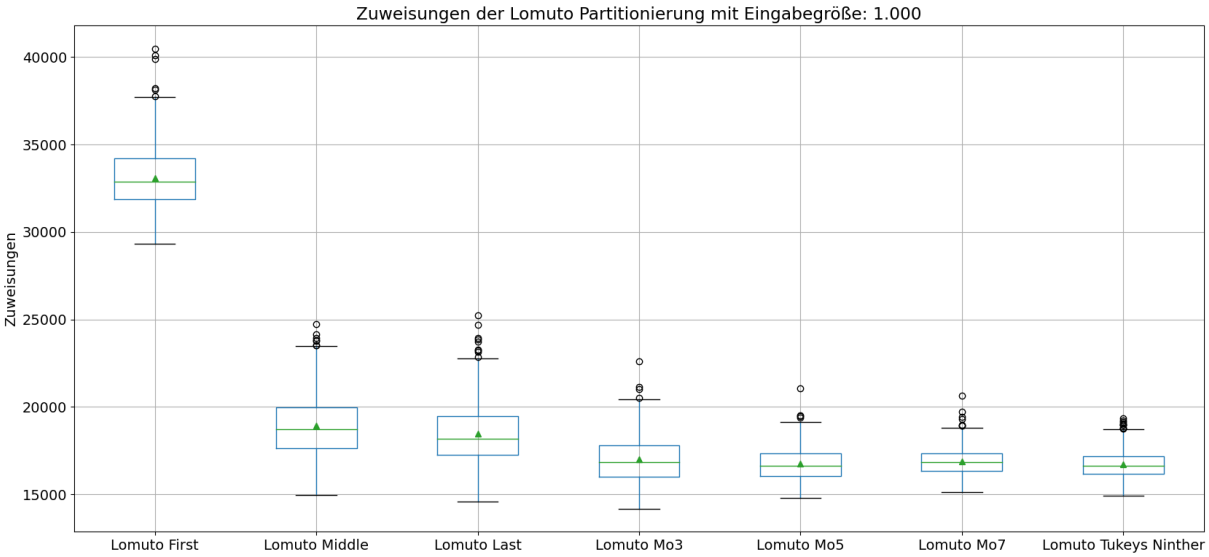

Abbildung 53: Zuweisungen Lomuto Funktionen - Benchmarks aller Methoden

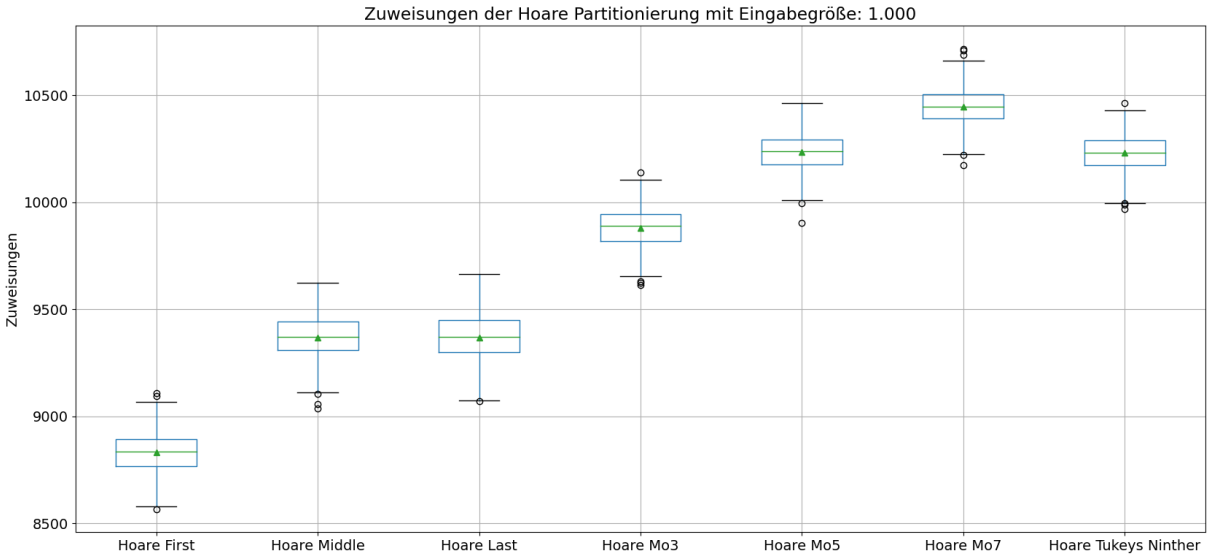

Abbildung 54: Zuweisungen Hoare Funktionen - Benchmarks aller Methoden

15.2 Eingabegröße 5.000

15.2.1 Vergleiche

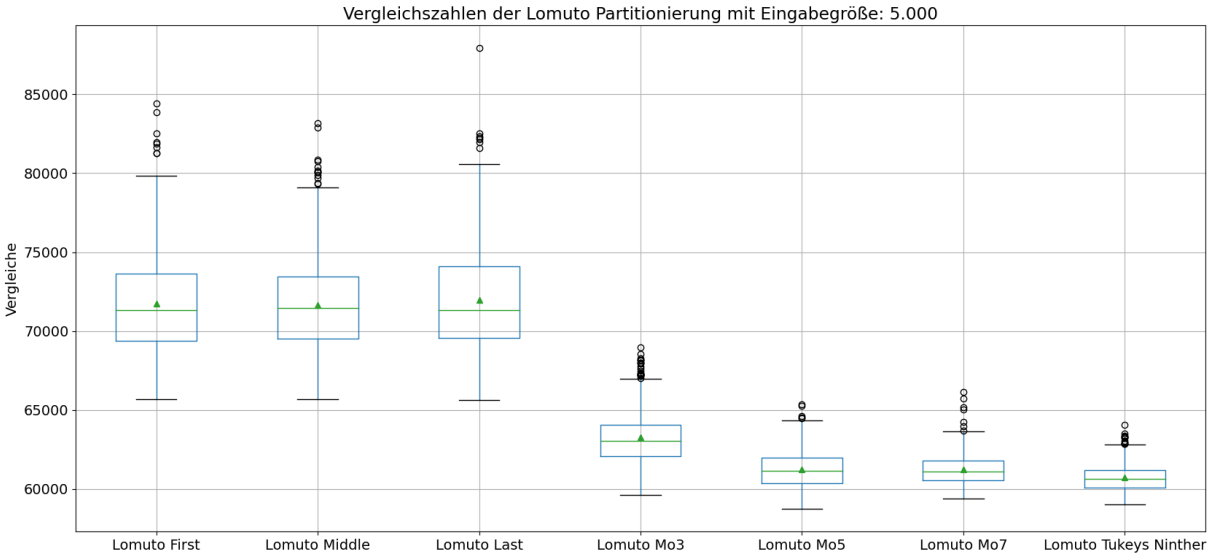

Abbildung 55: Vergleiche Lomuto Funktionen - Benchmarks aller Methoden

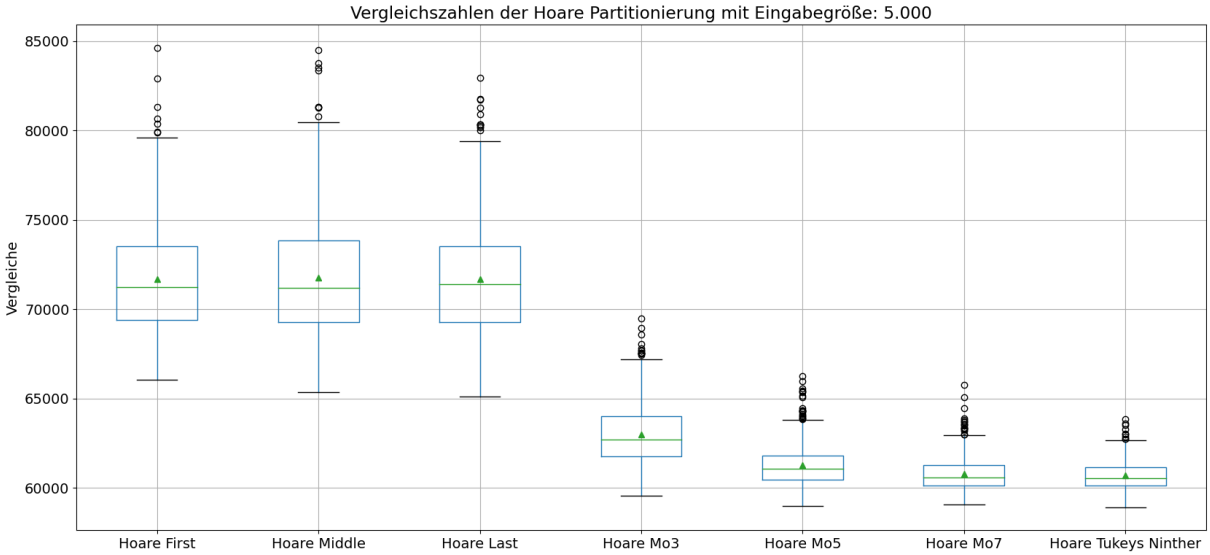

Abbildung 56: Vergleiche Hoare Funktionen - Benchmarks aller Methoden

15.2.2 Laufzeit

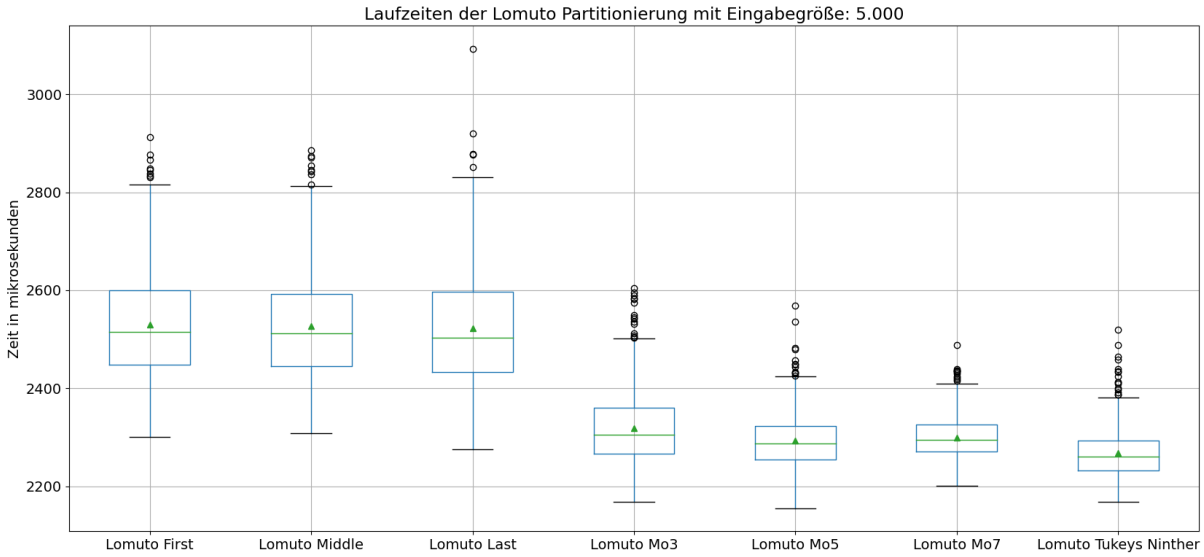

Abbildung 57: Laufzeit Lomuto Funktionen - Benchmarks aller Methoden

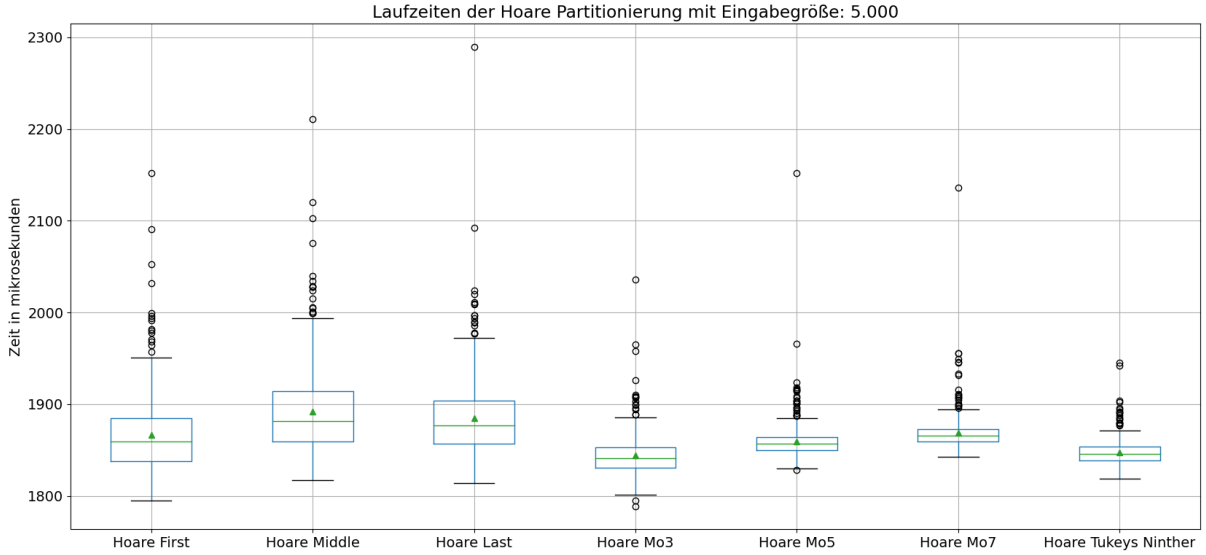

Abbildung 58: Laufzeit Hoare Funktionen - Benchmarks aller Methoden

15.2.3 Zuweisungen

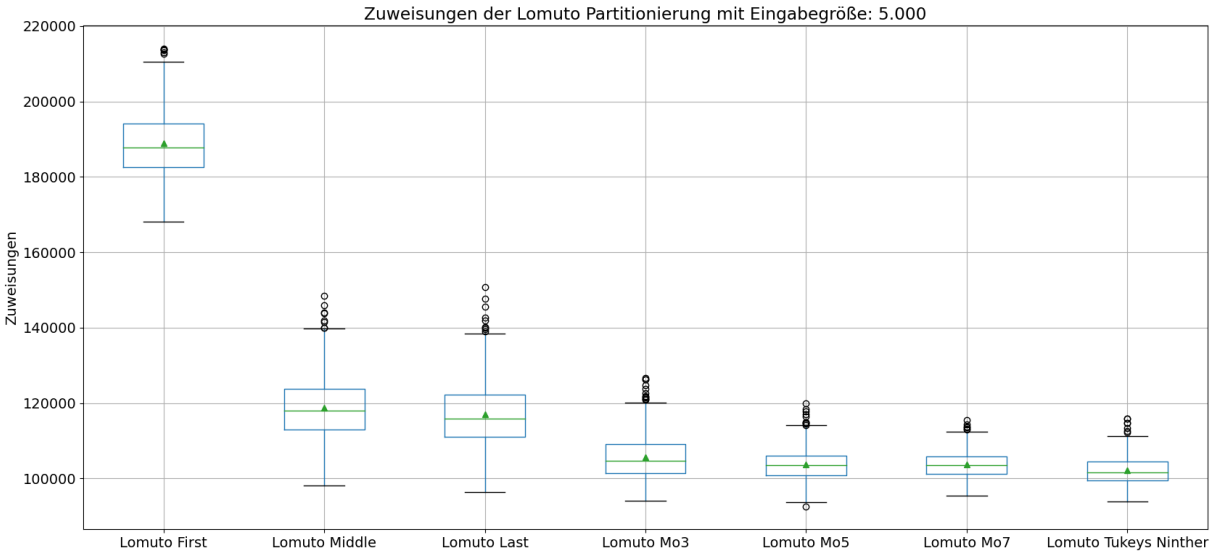

Abbildung 59: Zuweisungen Lomuto Funktionen - Benchmarks aller Methoden

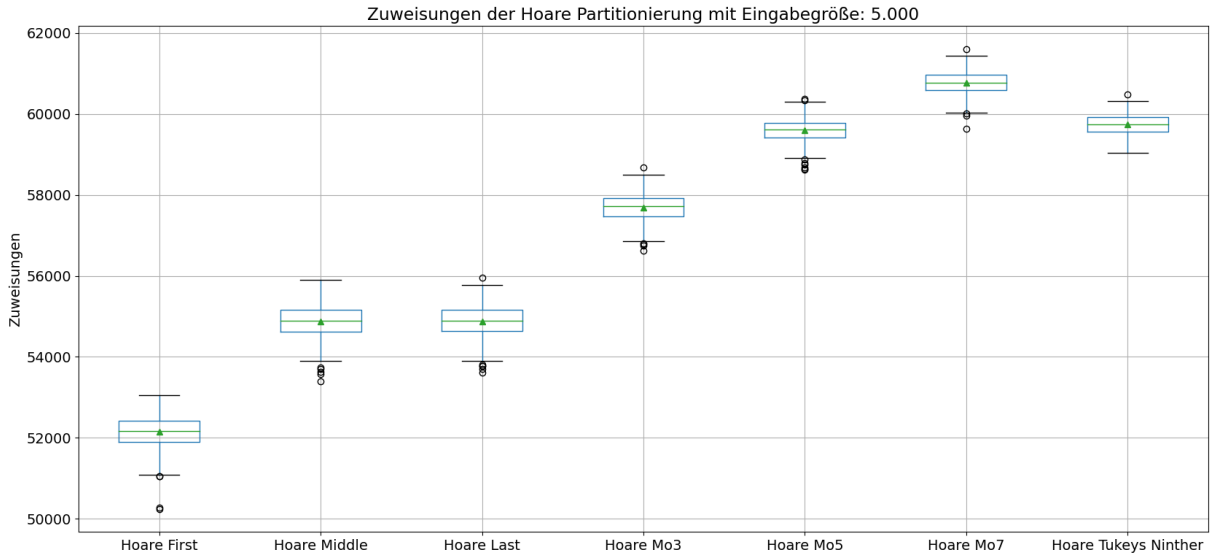

Abbildung 60: Zuweisungen Hoare Funktionen - Benchmarks aller Methoden

15.3 Eingabegröße 10.000

15.3.1 Vergleiche

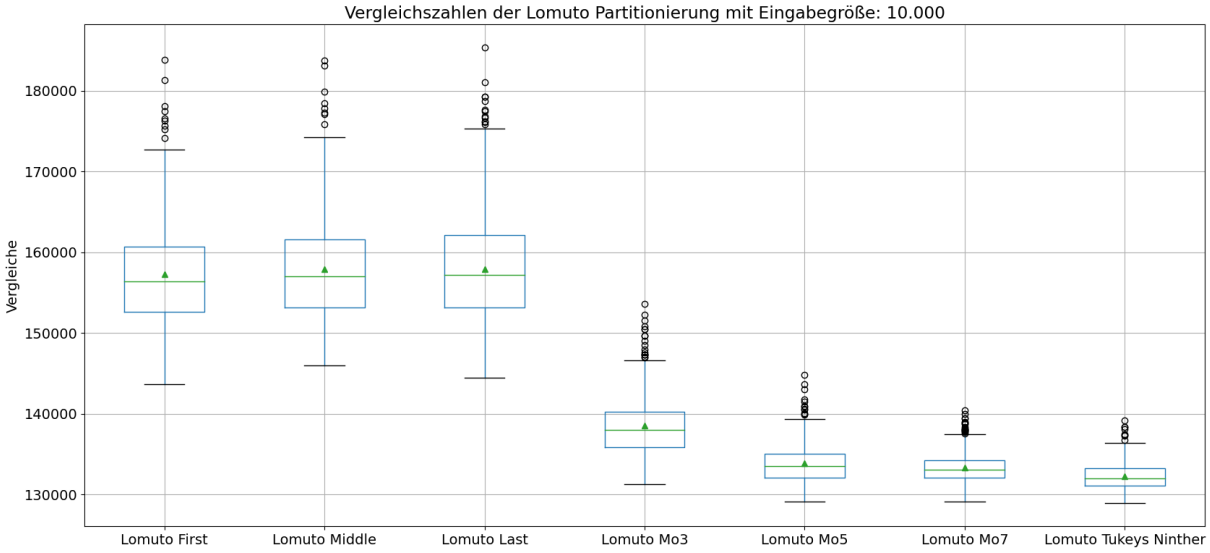

Abbildung 61: Vergleiche Lomuto Funktionen - Benchmarks aller Methoden

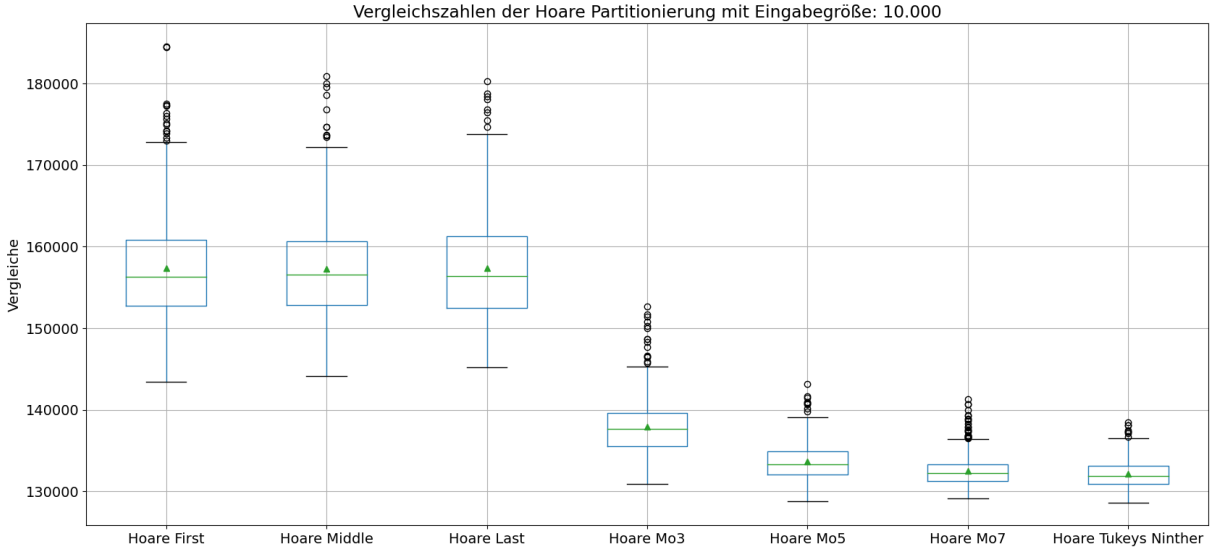

Abbildung 62: Vergleiche Hoare Funktionen - Benchmarks aller Methoden

15.3.2 Laufzeit

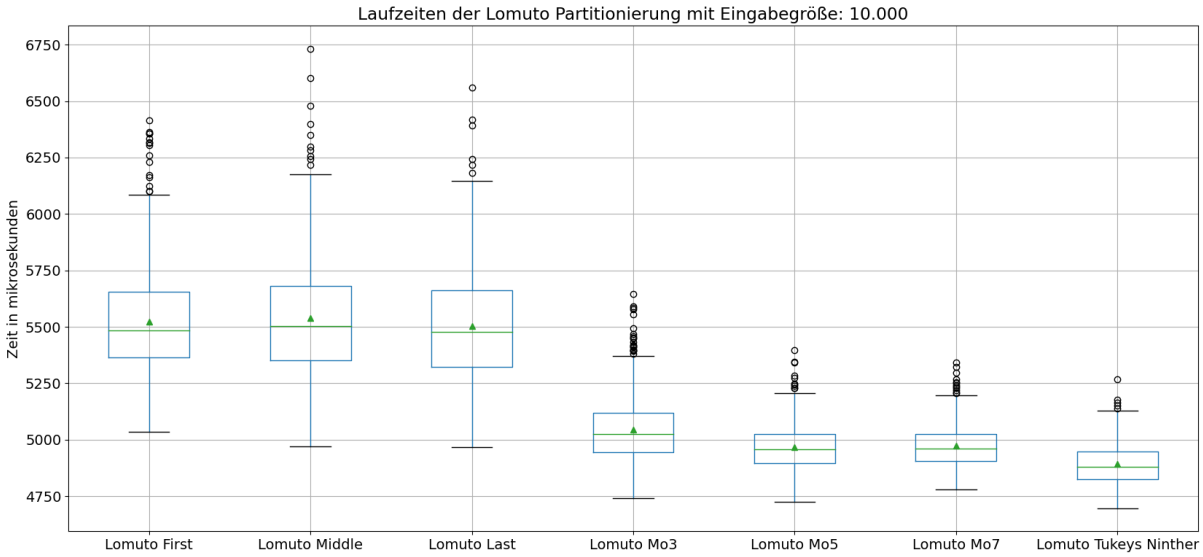

Abbildung 63: Laufzeit Lomuto Funktionen - Benchmarks aller Methoden

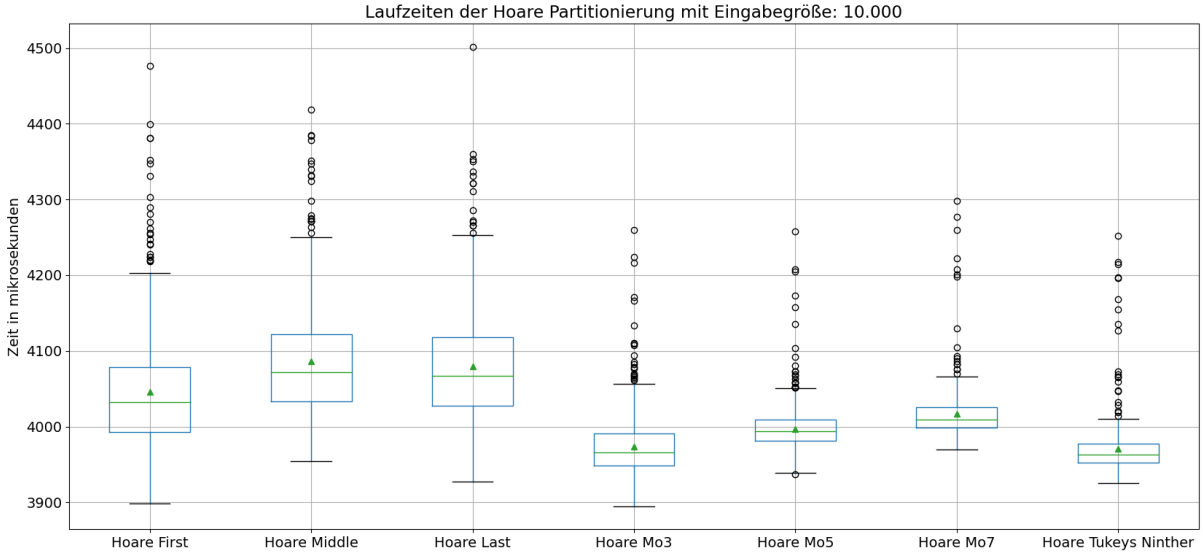

Abbildung 64: Laufzeit Hoare Funktionen - Benchmarks aller Methoden

15.3.3 Zuweisungen

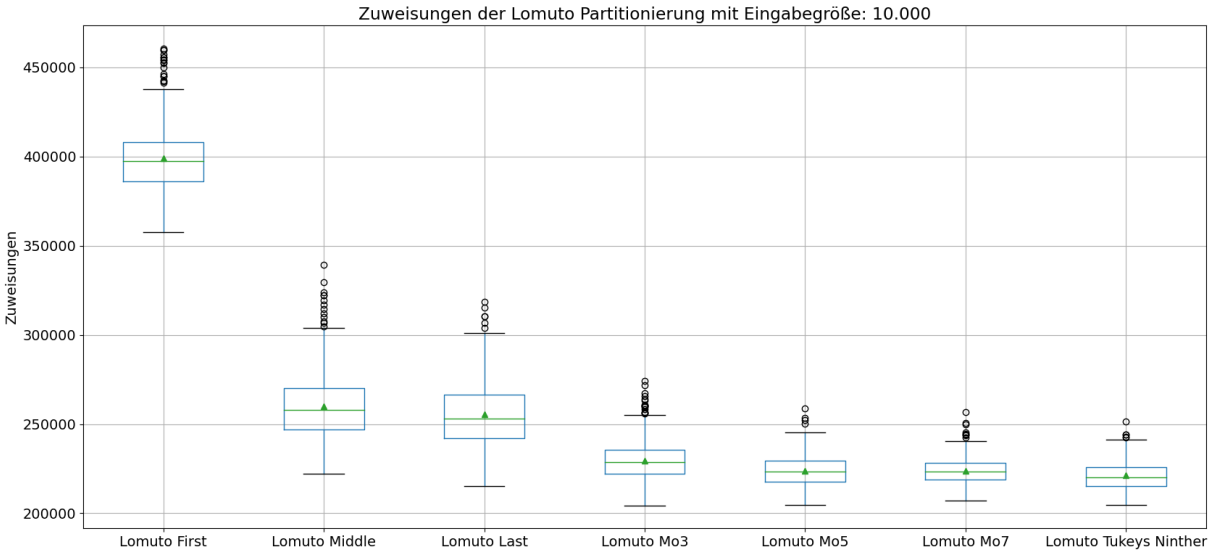

Abbildung 65: Zuweisungen Lomuto Funktionen - Benchmarks aller Methoden

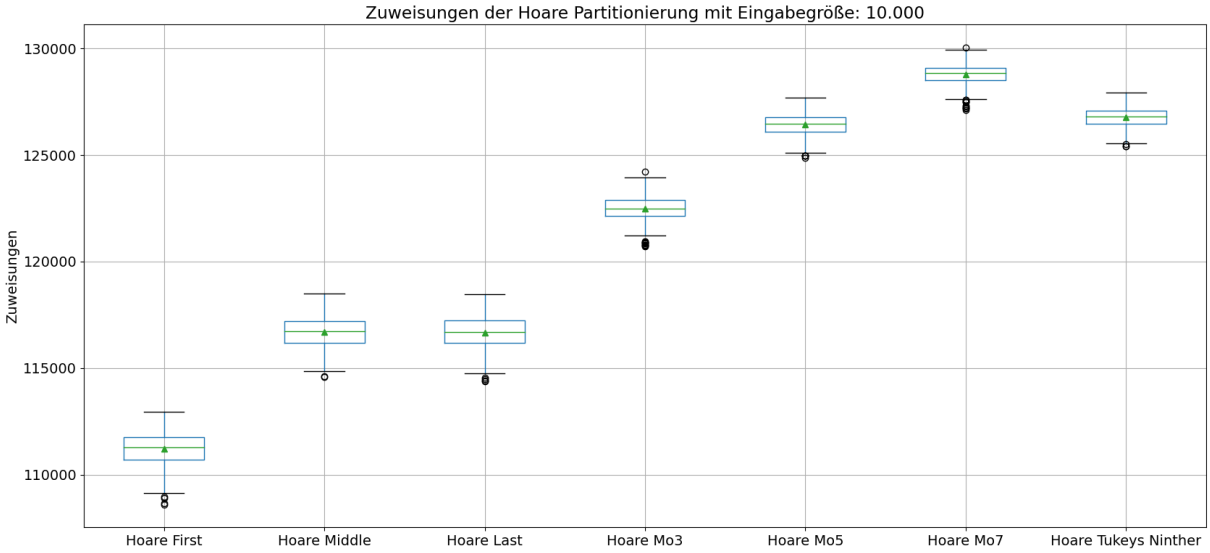

Abbildung 66: Zuweisungen Hoare Funktionen - Benchmarks aller Methoden

15.4 Eingabegröße 50.000

15.4.1 Vergleiche

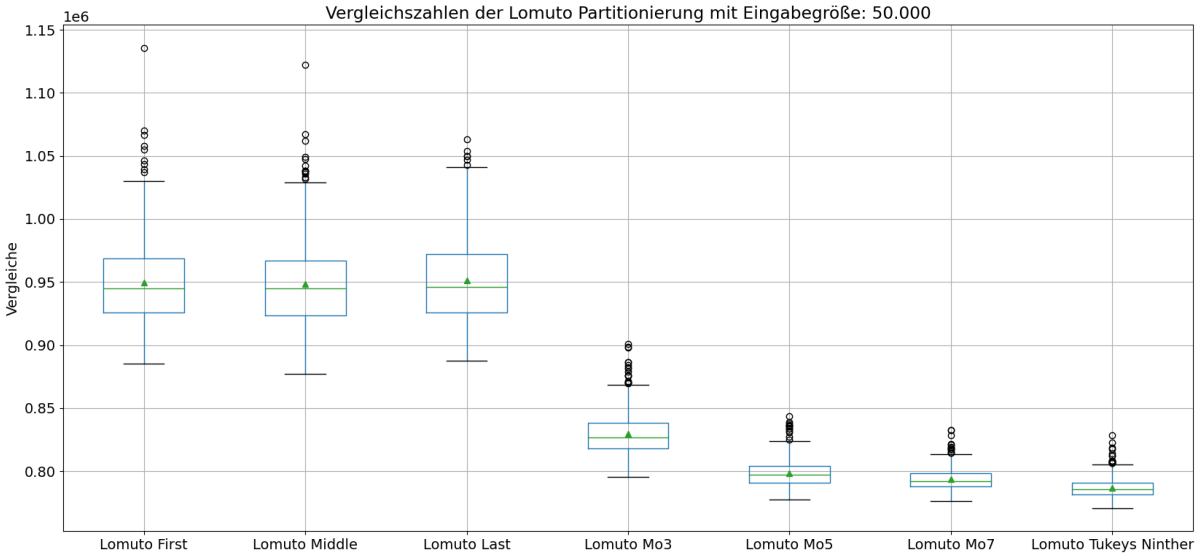

Abbildung 67: Vergleiche Lomuto Funktionen - Benchmarks aller Methoden

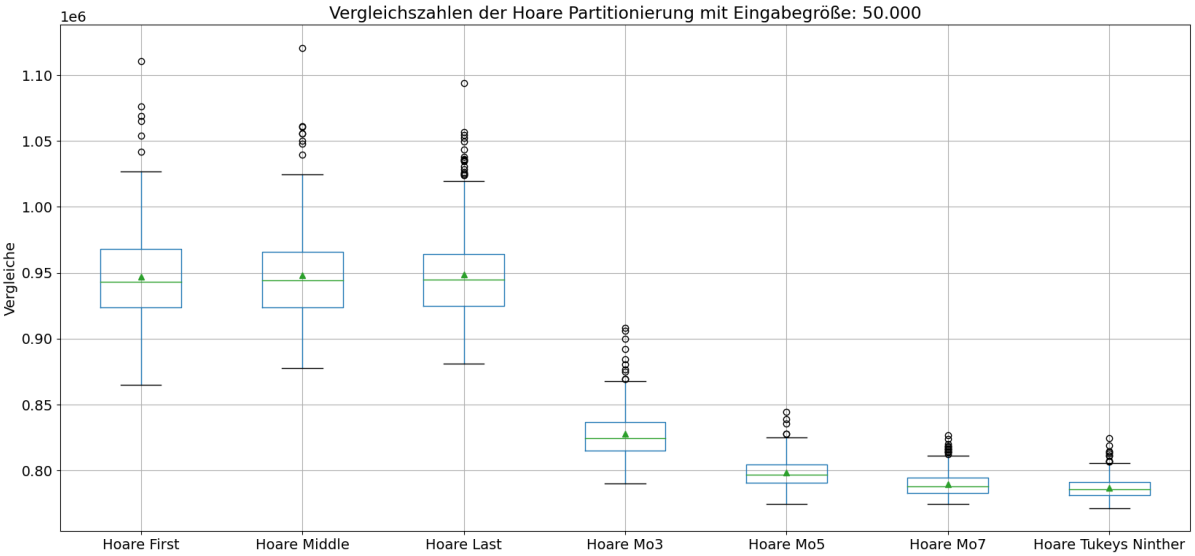

Abbildung 68: Vergleiche Hoare Funktionen - Benchmarks aller Methoden

15.4.2 Laufzeit

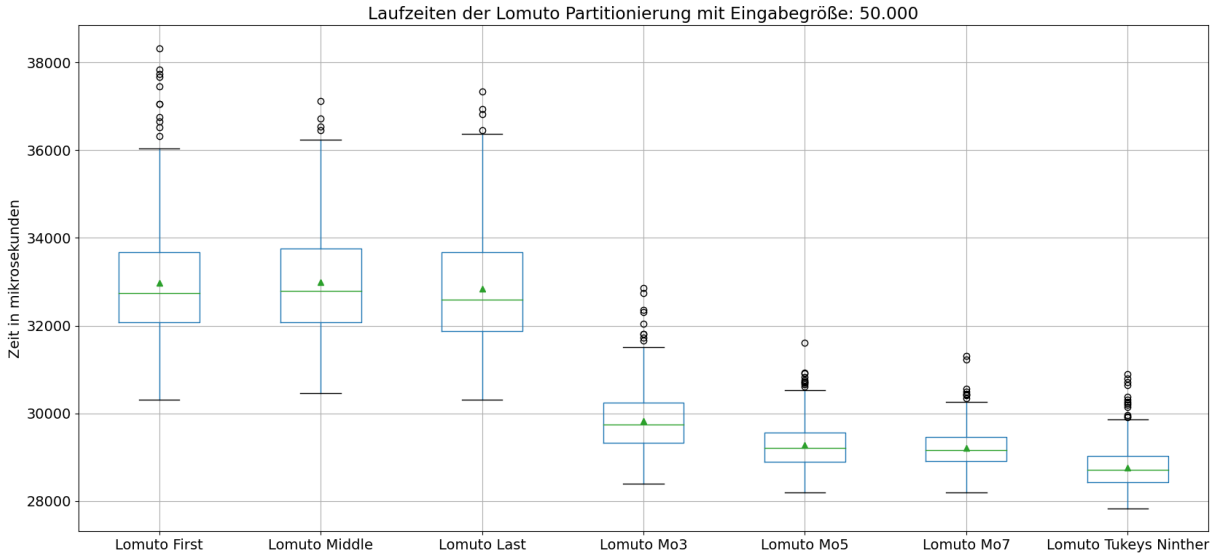

Abbildung 69: Laufzeit Lomuto Funktionen - Benchmarks aller Methoden

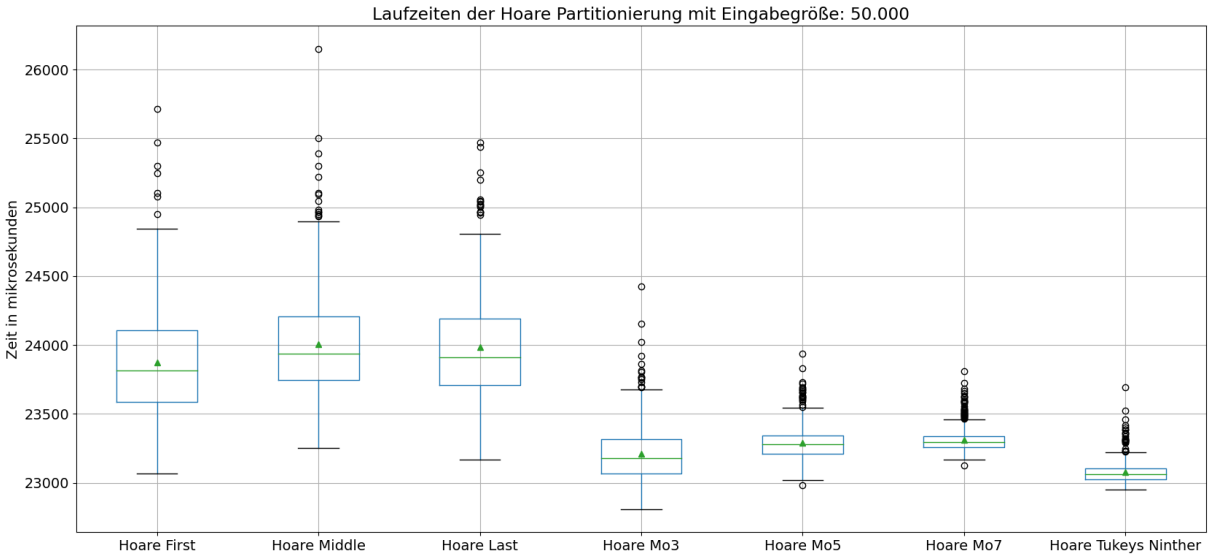

Abbildung 70: Laufzeit Hoare Funktionen - Benchmarks aller Methoden

15.4.3 Zuweisungen

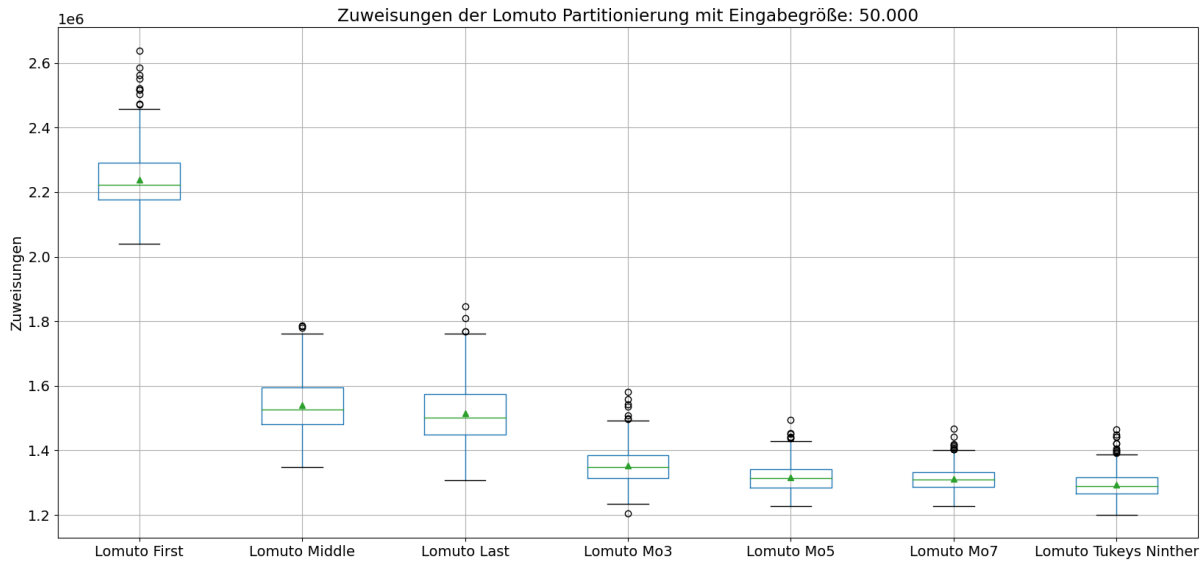

Abbildung 71: Zuweisungen Lomuto Funktionen - Benchmarks aller Methoden

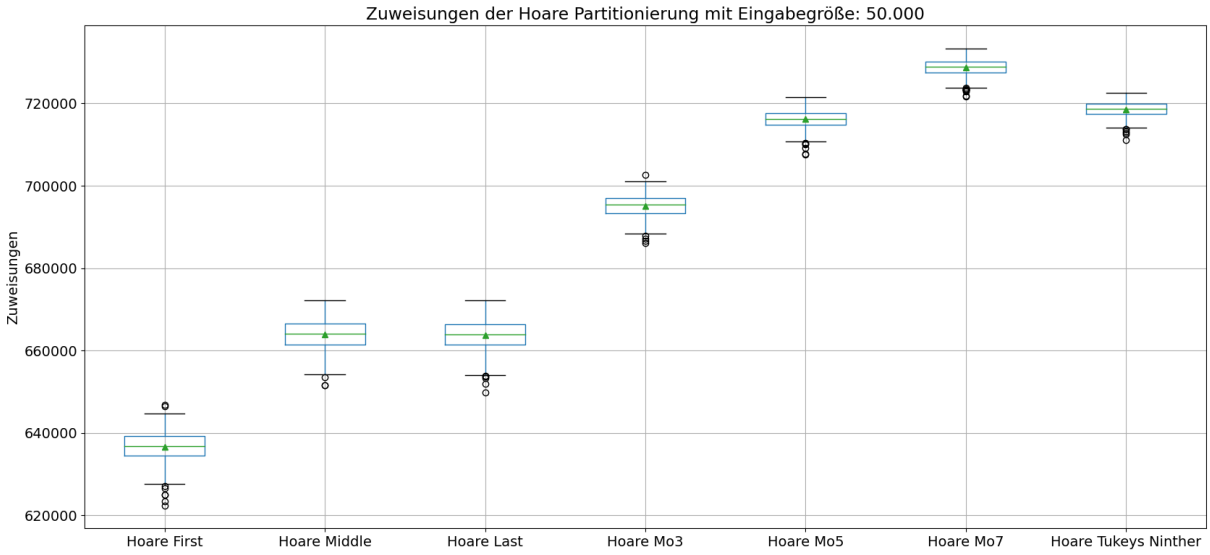

Abbildung 72: Zuweisungen Hoare Funktionen - Benchmarks aller Methoden

15.5 Eingabegröße 100.000

15.5.1 Vergleiche

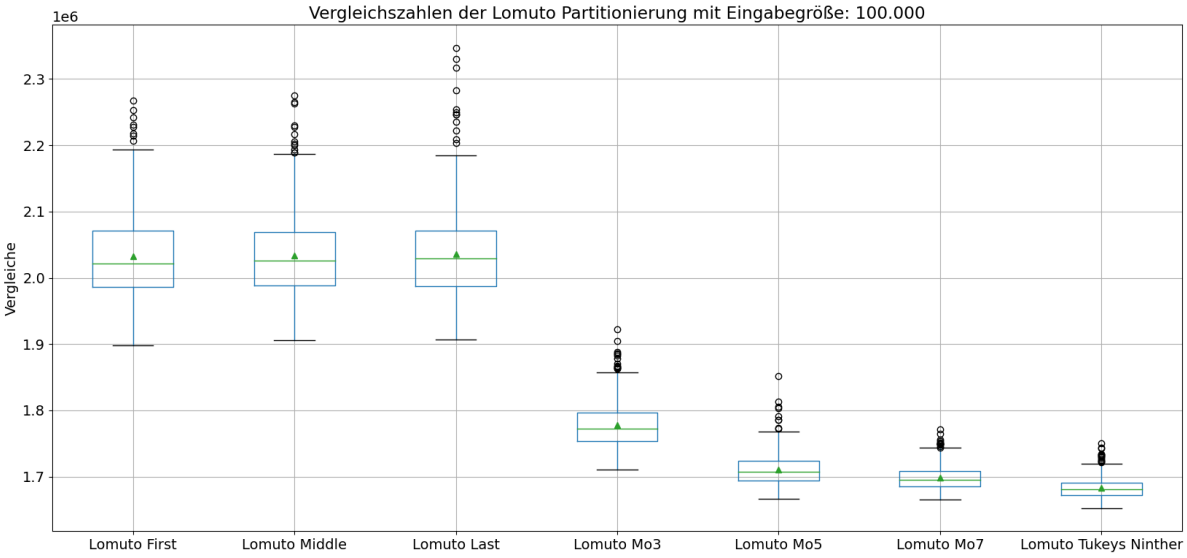

Abbildung 73: Vergleiche Lomuto Funktionen - Benchmarks aller Methoden

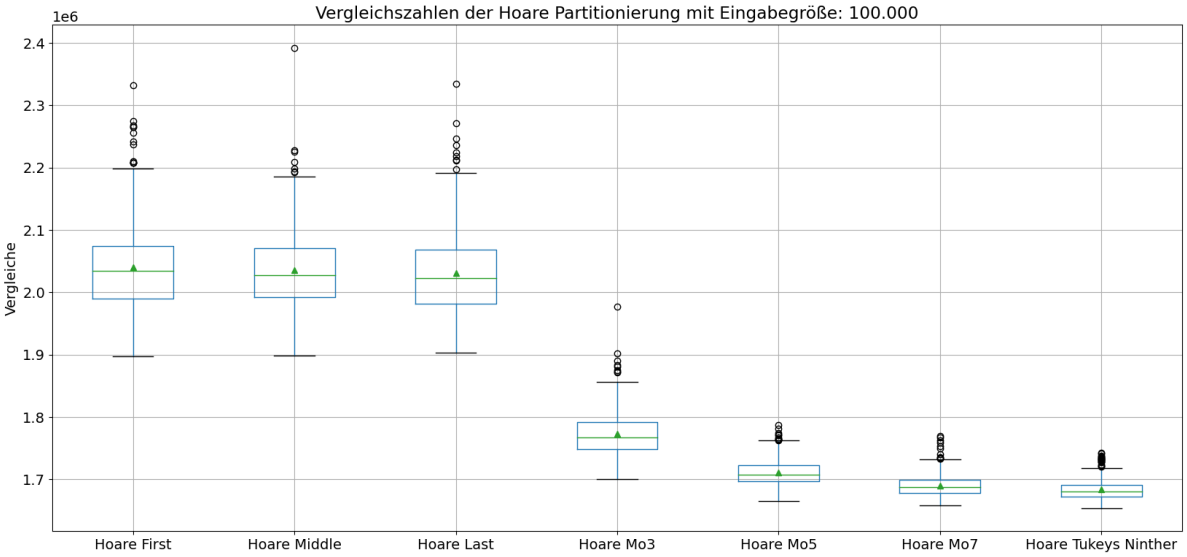

Abbildung 74: Vergleiche Hoare Funktionen - Benchmarks aller Methoden

15.5.2 Laufzeit

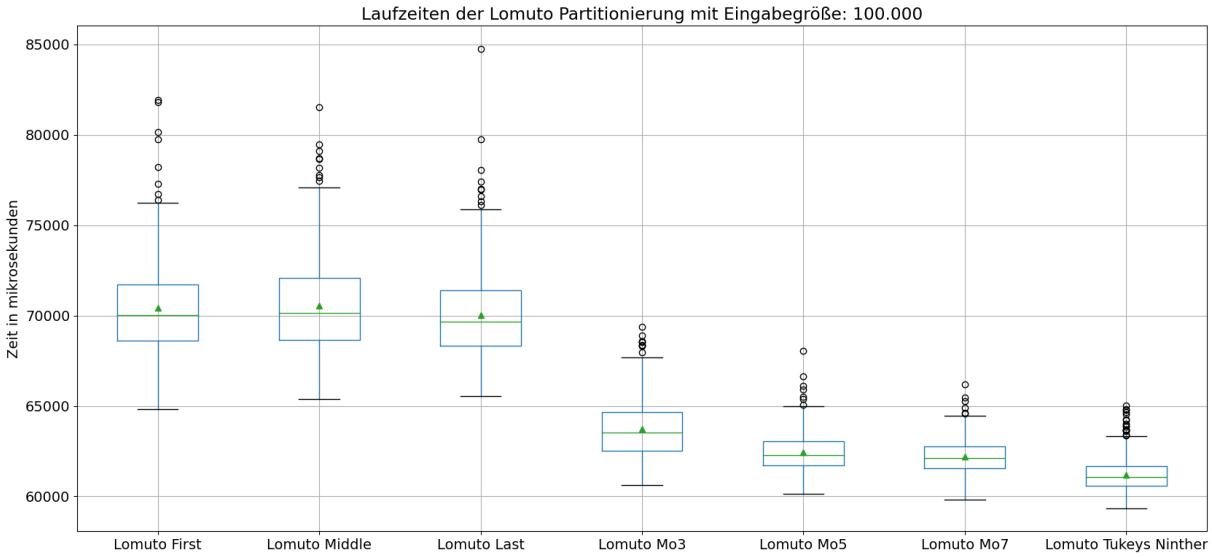

Abbildung 75: Laufzeit Lomuto Funktionen - Benchmarks aller Methoden

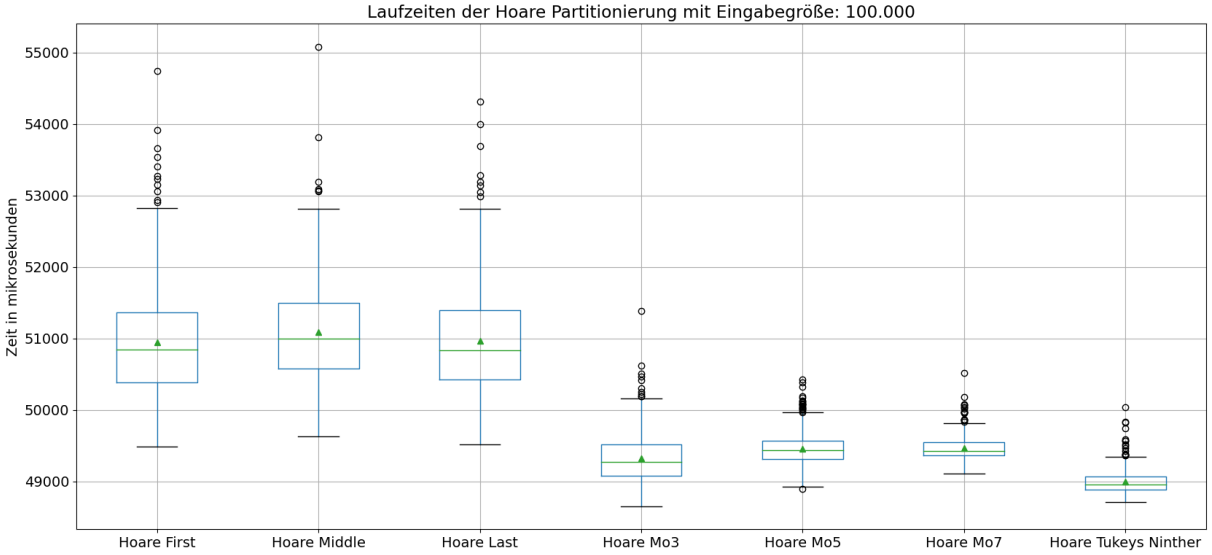

Abbildung 76: Laufzeit Hoare Funktionen - Benchmarks aller Methoden

15.5.3 Zuweisungen

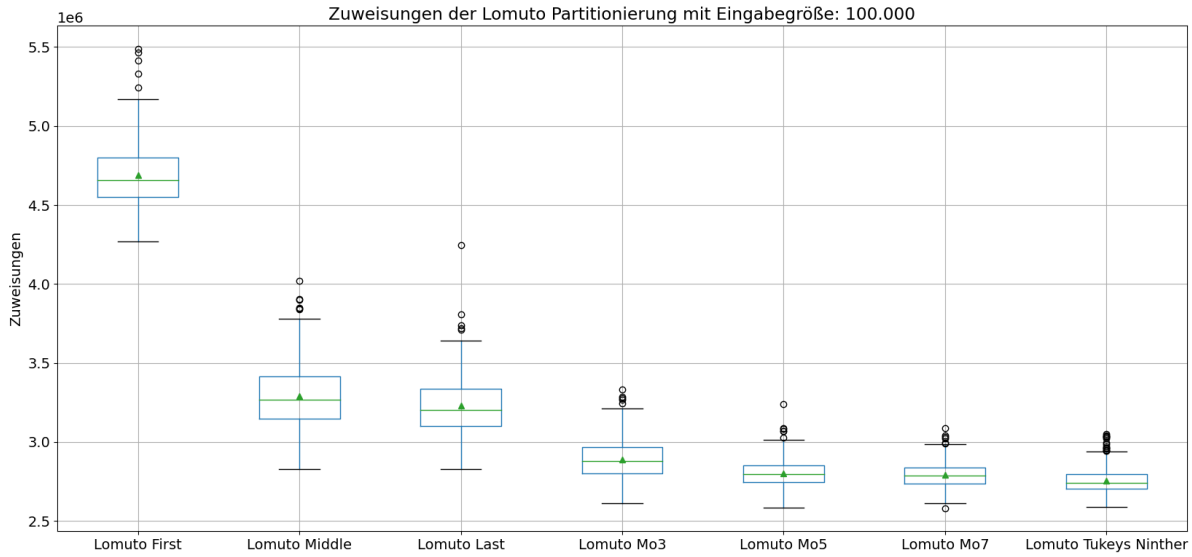

Abbildung 77: Zuweisungen Lomuto Funktionen - Benchmarks aller Methoden

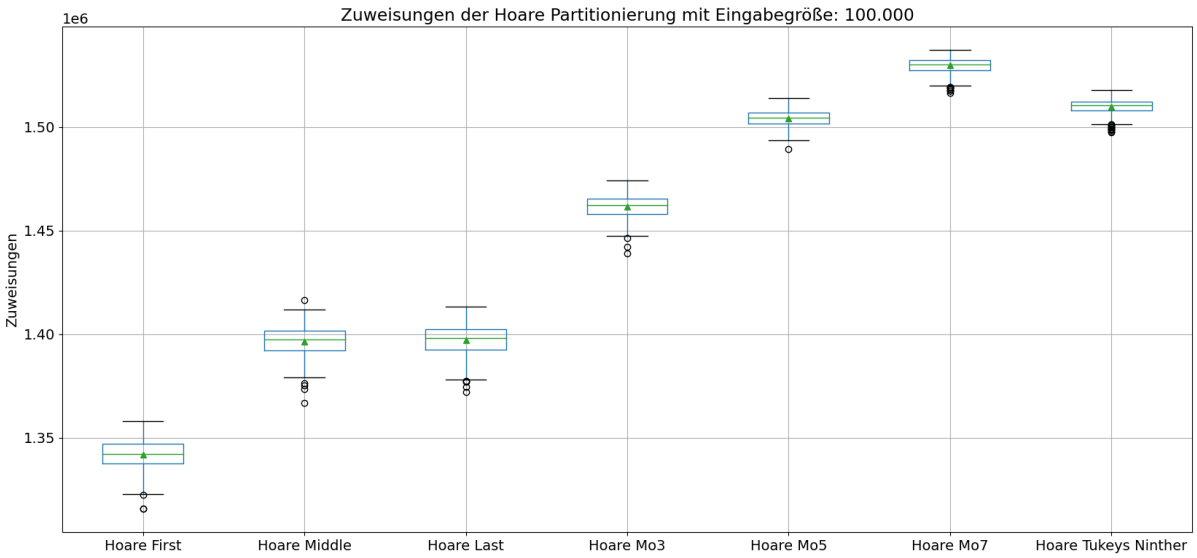

Abbildung 78: Zuweisungen Hoare Funktionen - Benchmarks aller Methoden

15.6 Eingabegröße 250.000

15.6.1 Vergleiche

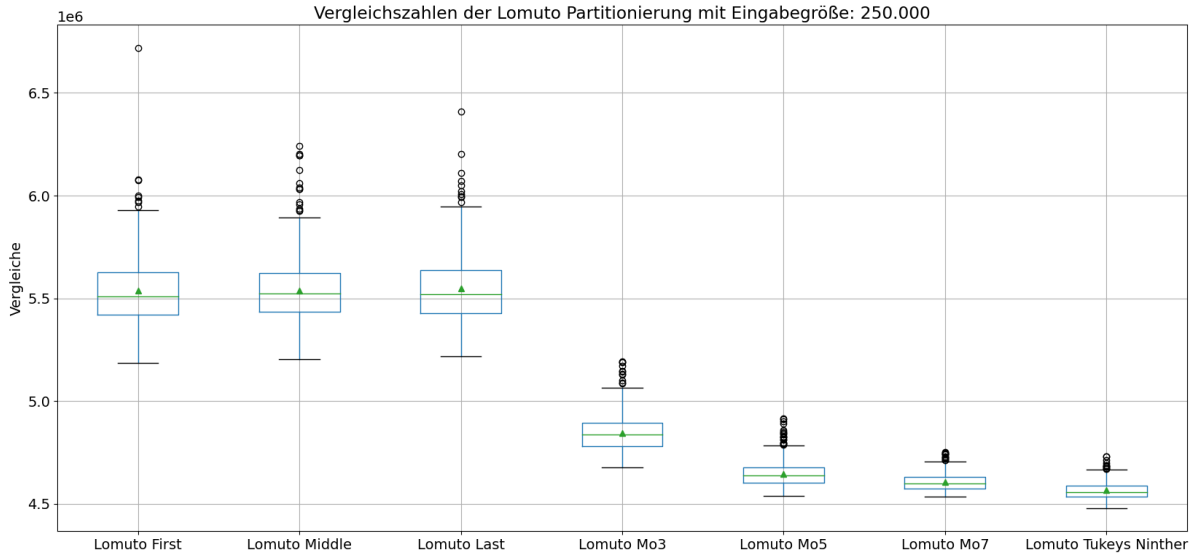

Abbildung 79: Vergleiche Lomuto Funktionen - Benchmarks aller Methoden

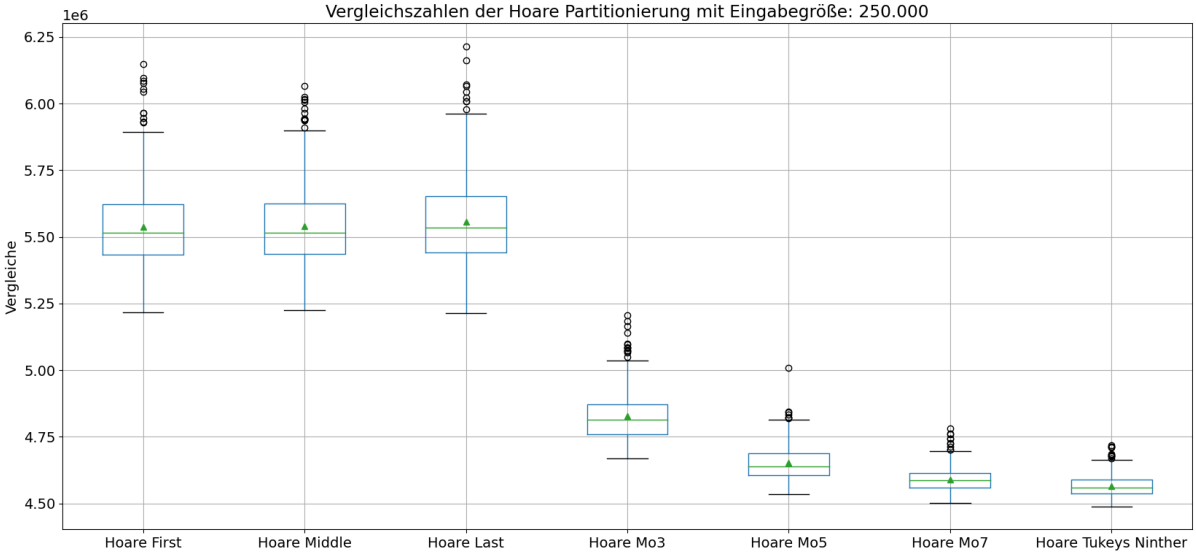

Abbildung 80: Vergleiche Hoare Funktionen - Benchmarks aller Methoden

15.6.2 Laufzeit

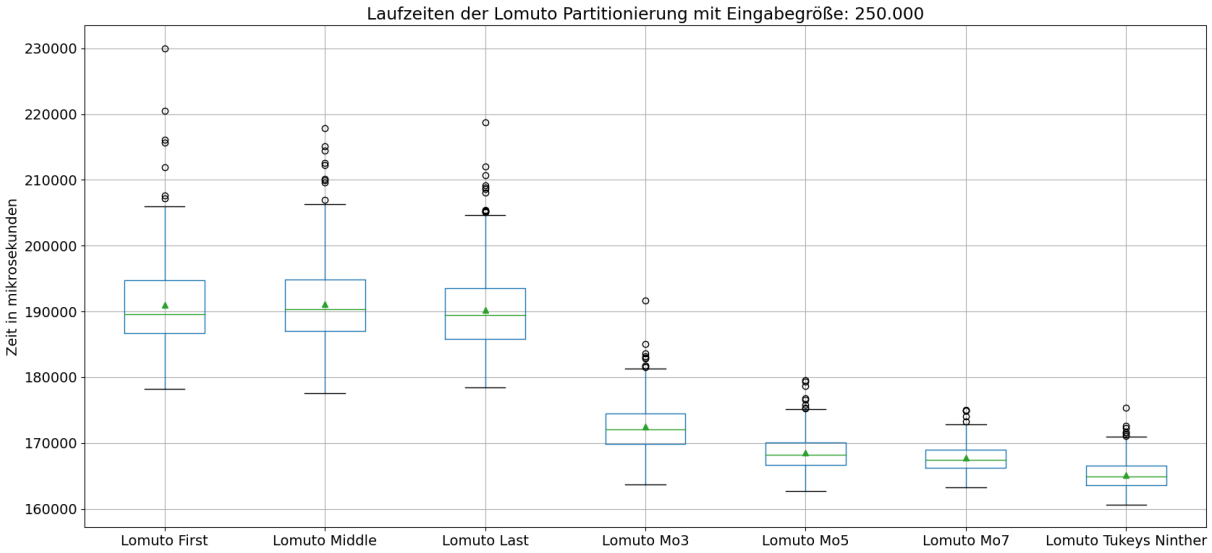

Abbildung 81: Laufzeit Lomuto Funktionen - Benchmarks aller Methoden

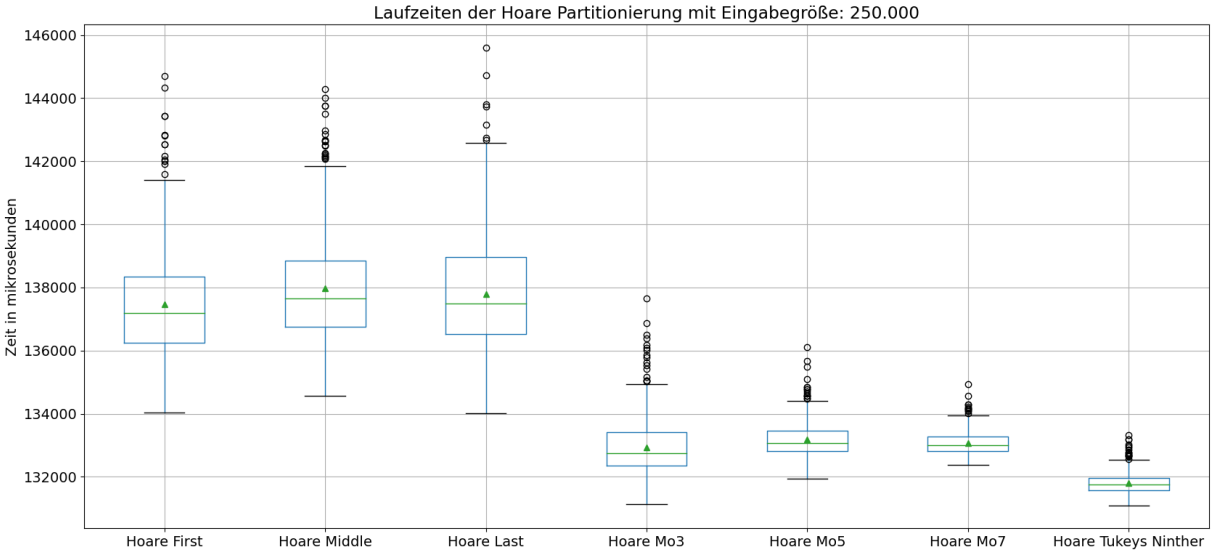

Abbildung 82: Laufzeit Hoare Funktionen - Benchmarks aller Methoden

15.6.3 Zuweisungen

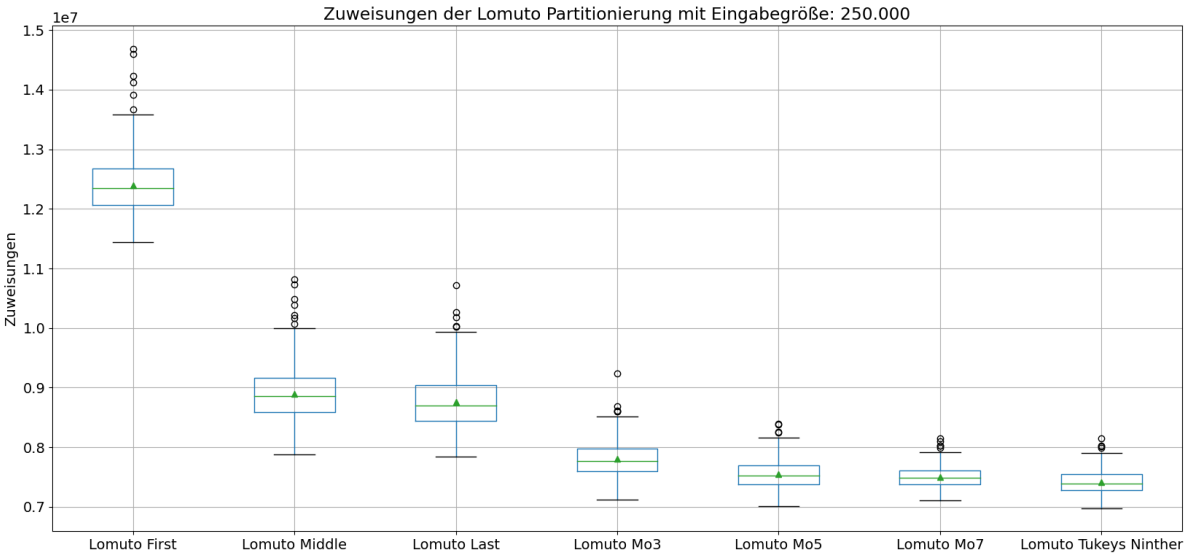

Abbildung 83: Zuweisungen Lomuto Funktionen - Benchmarks aller Methoden

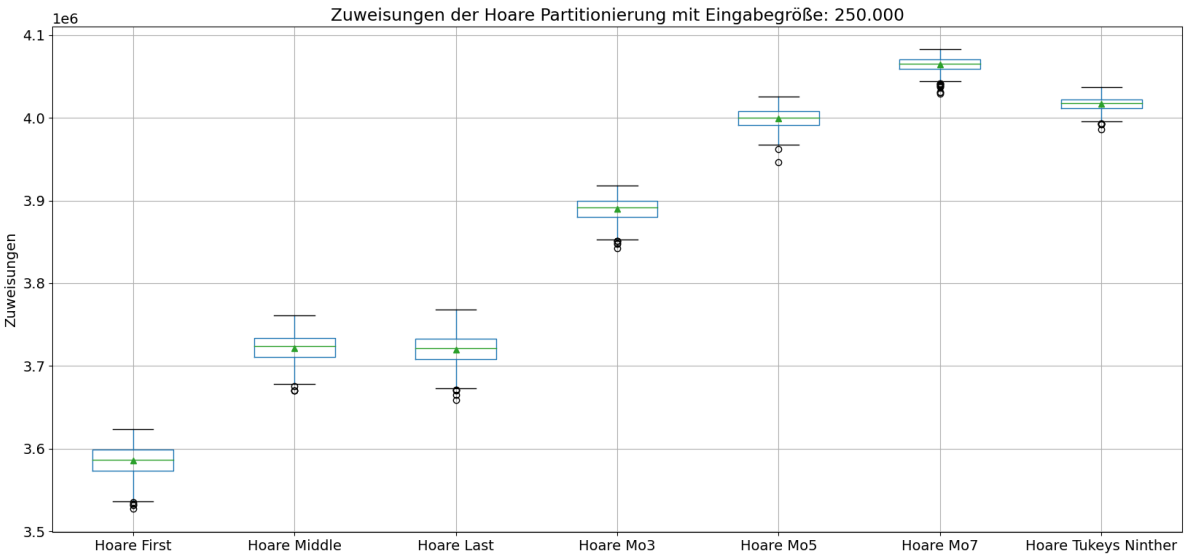

Abbildung 84: Zuweisungen Hoare Funktionen - Benchmarks aller Methoden

15.7 Eingabegröße 500.000

15.7.1 Vergleiche

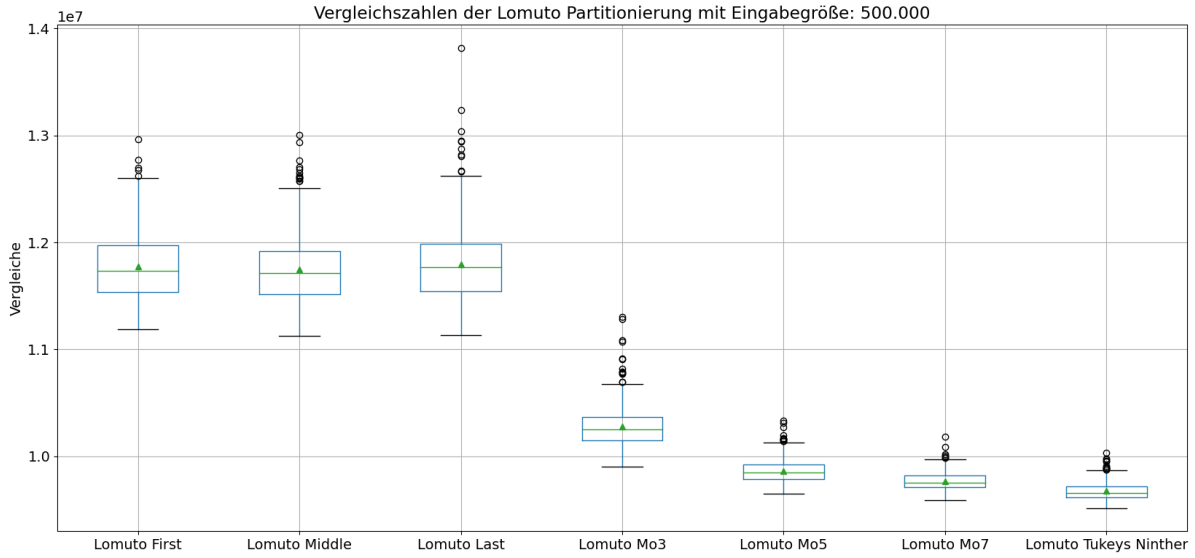

Abbildung 85: Vergleiche Lomuto Funktionen - Benchmarks aller Methoden

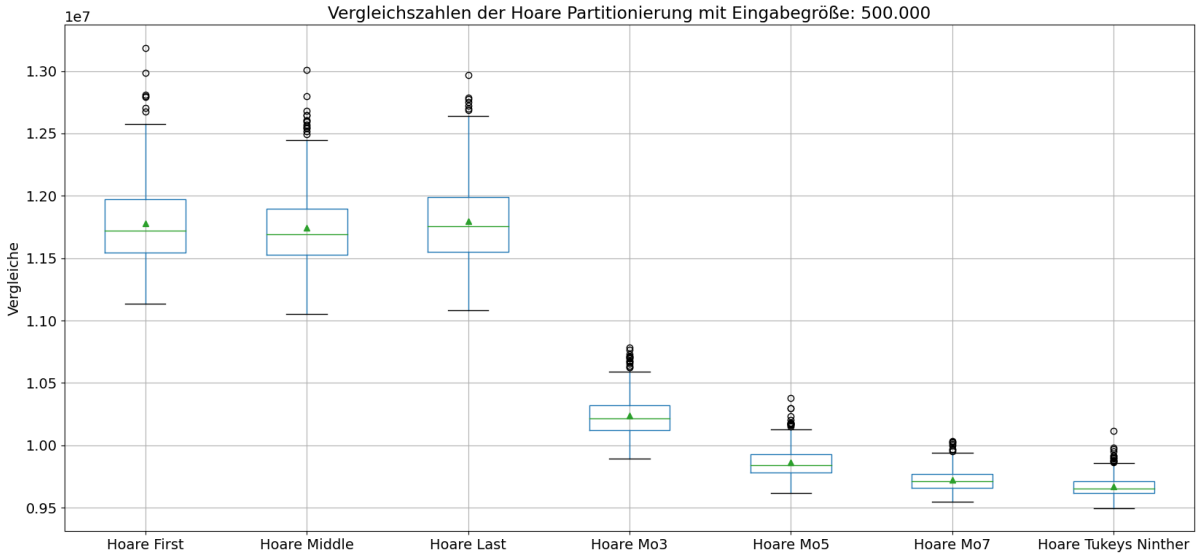

Abbildung 86: Vergleiche Hoare Funktionen - Benchmarks aller Methoden

15.7.2 Laufzeit

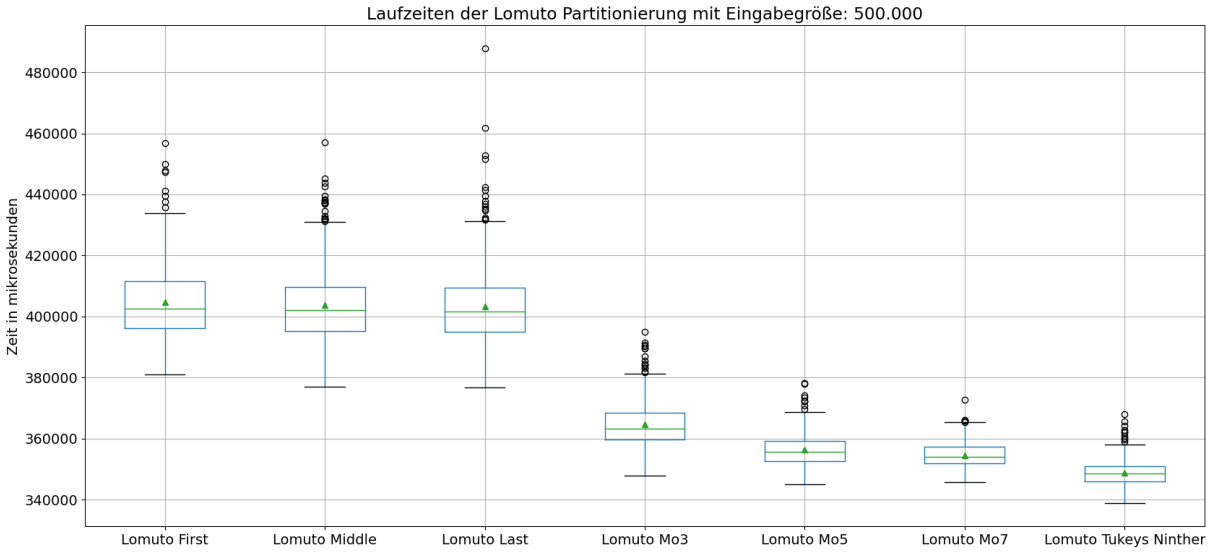

Abbildung 87: Laufzeit Lomuto Funktionen - Benchmarks aller Methoden

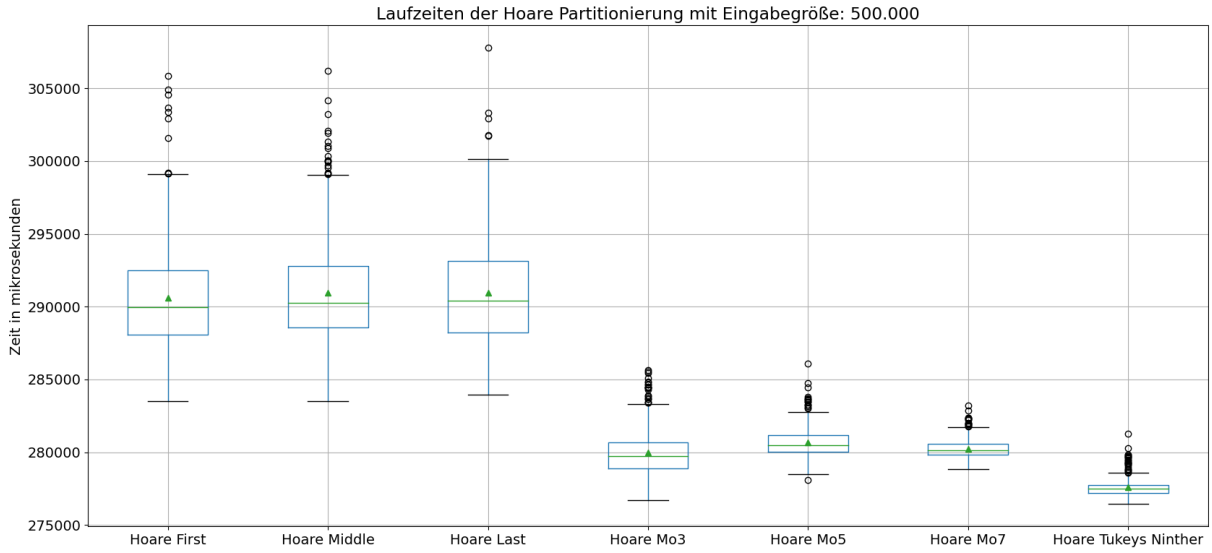

Abbildung 88: Laufzeit Hoare Funktionen - Benchmarks aller Methoden

15.7.3 Zuweisungen

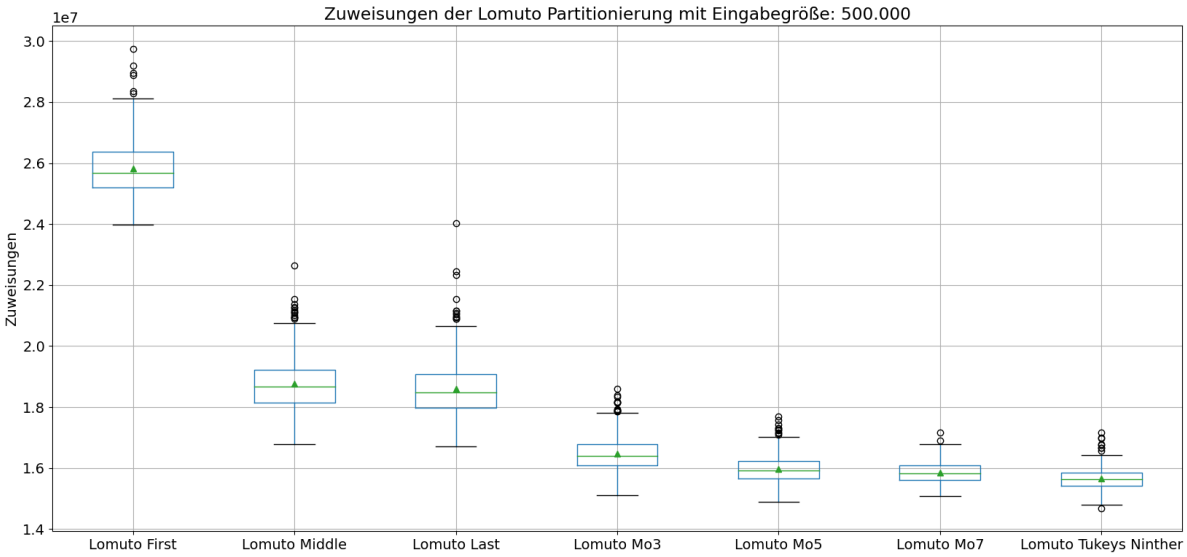

Abbildung 89: Zuweisungen Lomuto Funktionen - Benchmarks aller Methoden

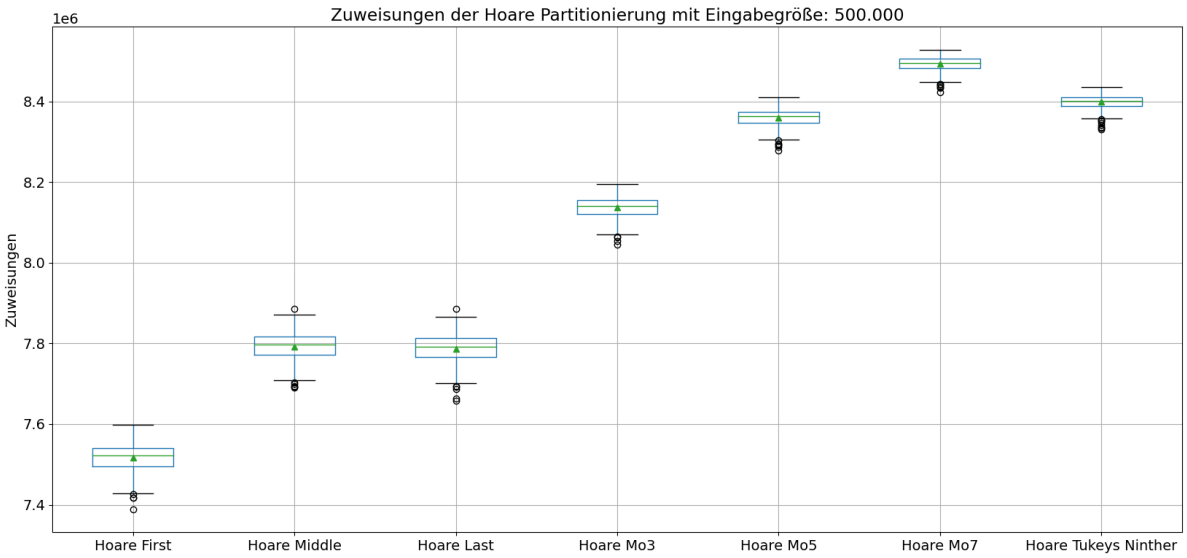

Abbildung 90: Zuweisungen Hoare Funktionen - Benchmarks aller Methoden

15.8 Eingabegröße 1.000.000

15.8.1 Vergleiche

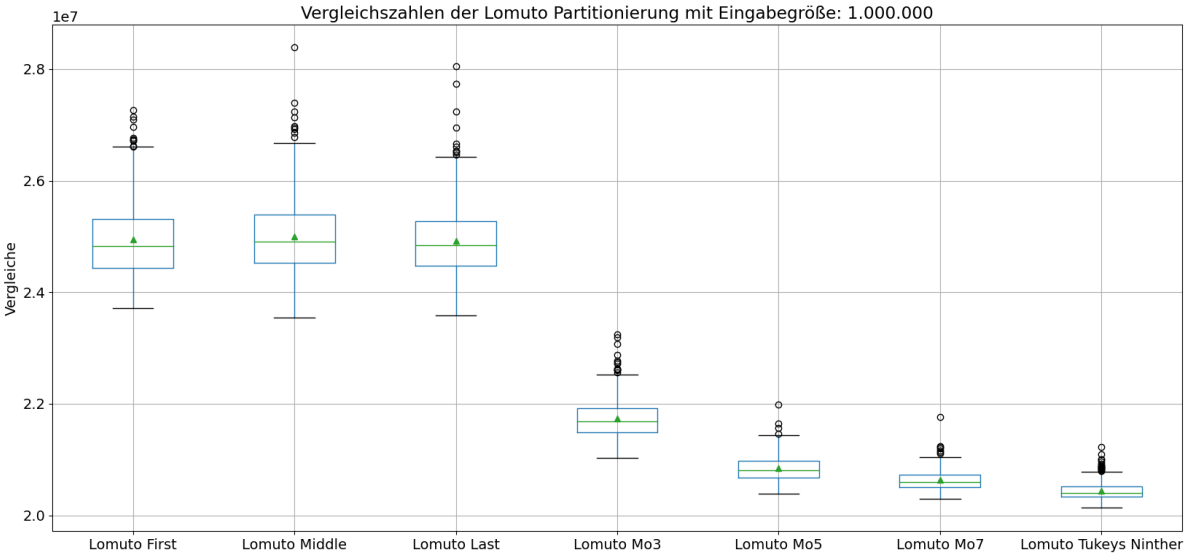

Abbildung 91: Vergleiche Lomuto Funktionen - Benchmarks aller Methoden

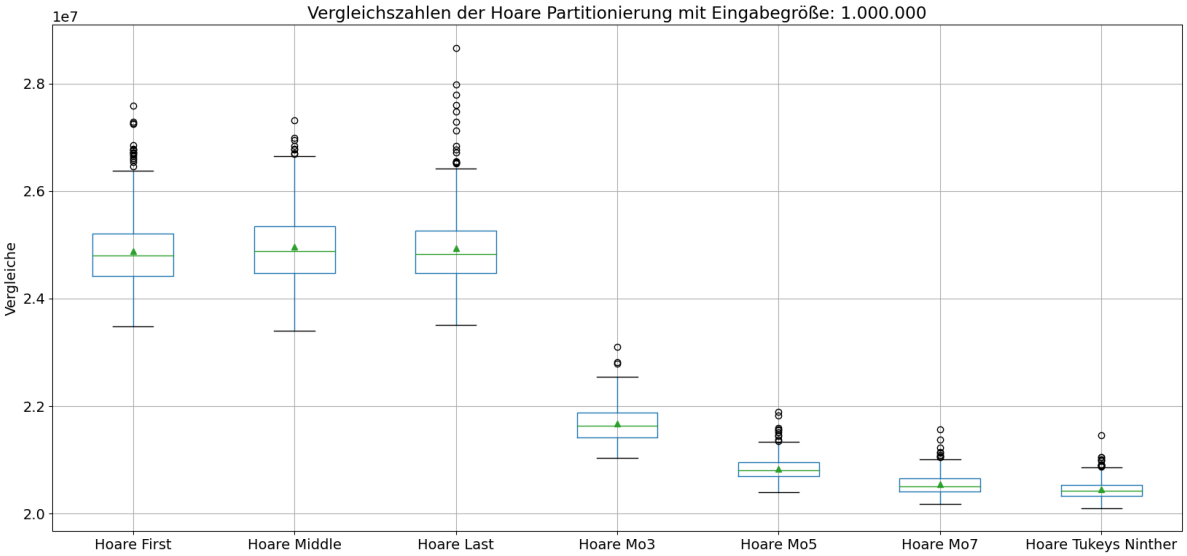

Abbildung 92: Vergleiche Hoare Funktionen - Benchmarks aller Methoden

15.8.2 Laufzeit

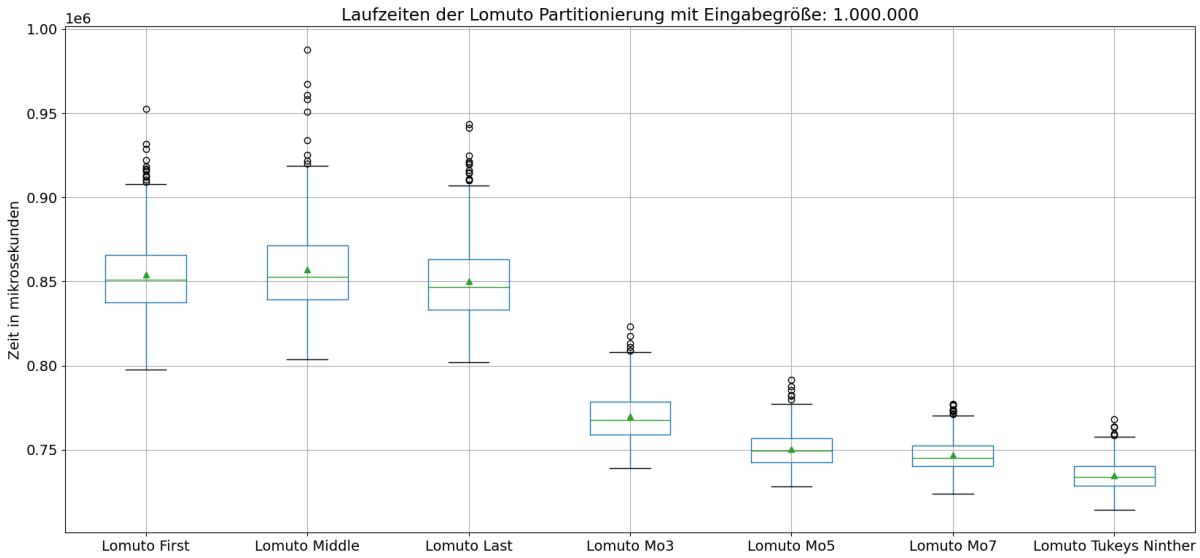

Abbildung 93: Laufzeit Lomuto Funktionen - Benchmarks aller Methoden

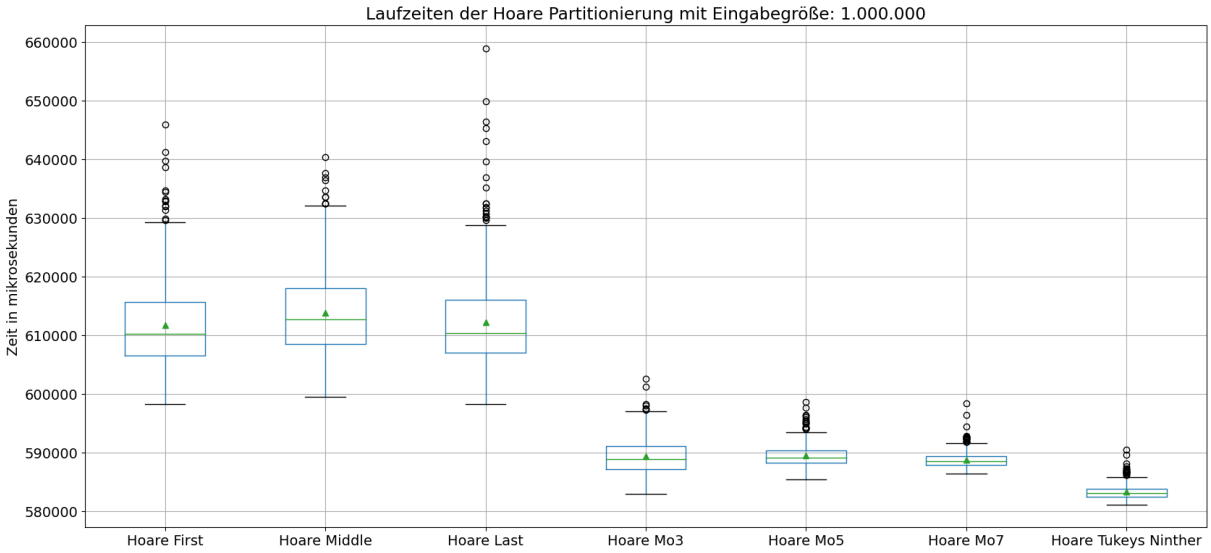

Abbildung 94: Laufzeit Hoare Funktionen - Benchmarks aller Methoden

15.8.3 Zuweisungen

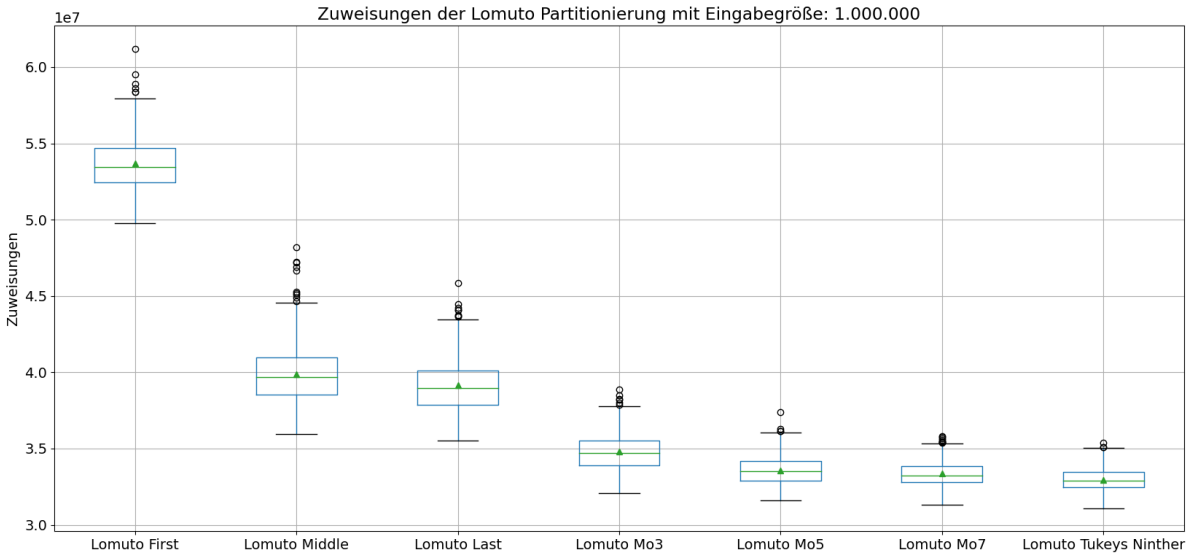

Abbildung 95: Zuweisungen Lomuto Funktionen - Benchmarks aller Methoden

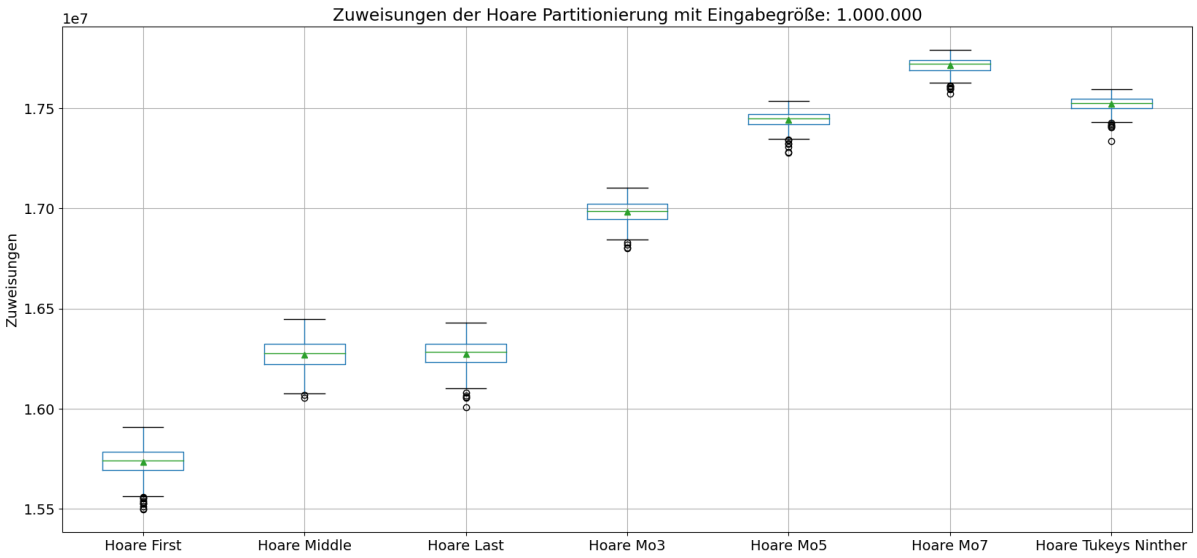

Abbildung 96: Zuweisungen Hoare Funktionen - Benchmarks aller Methoden

Literatur

[1] Robert Sedgewick. "The analysis of Quicksort programs". In: (1977). DOI: http://cslabcms.nju.edu.cn/problem_solving/images/7/73/The_Analysis_of_Quicksort_Programs_%28Sedgewick%29.pdf.

[2] *Performance of Quicksort 1*. Accessed: 22.12.2021. URL: http://homepages.math.uic.edu/~leon/cs-mcs401-r07/handouts/quicksort-continued.pdf.

[3] *Performance of Quicksort 2*. Accessed: 22.12.2021. URL: https://cs.fit.edu/~pkc/classes/writing/hw14/luis.pdf.

[4] *Glib qsort*. Accessed: 28.11.2021. URL: https://code.woboq.org/userspace/glibc/stdlib/qsort.c.html.

[5] *libstdc++ Library Version 4.6.4*. Accessed: 27.11.2021. URL: https://gnu.huihoo.org/gcc/gcc-4.6.4/libstdc++/api/a01058_source.html.

[6] *libstdc++ Library Version 4.7.4*. Accessed: 27.11.2021. URL: https://gnu.huihoo.org/gcc/gcc-4.7.4/libstdc++/api/a01355_source.html#l02310.

[7] *Libc++ algorithm.h*. Accessed: 28.11.2021. URL: https://github.com/llvm-mirror/libcxx/blob/master/include/algorithm.

[8] *Microsoft Standard Library*. Accessed: 28.11.2021. URL: https://github.com/microsoft/STL/blob/main/stl/inc/algorithm.

[9] *Php sort*. Accessed: 28.11.2021. URL: https://www.php.net/manual/de/function.sort.php.

[10] *(Java) Gnu Classpath qsort*. Accessed: 28.11.2021. URL: https://developer.classpath.org/doc/java/util/Arrays-source.html.

[11] *Swift 4.2 sort*. Accessed: 28.11.2021. URL: https://github.com/apple/swift/blob/swift-4.2-branch/stdlib/public/core/Sort.swift.gyb.

[12] *Rust sort*. Accessed: 28.11.2021. URL: https://github.com/rust-lang/rust/blob/master/library/core/src/slice/sort.rs.

[13] *Python listobject*. Accessed: 03.12.2021. URL: https://github.dev/python/cpython/blob/main/Objects/listobject.c.

[14] *Comparison of different Pivot selection choices*. Accessed: 05.12.2021. URL: https://scialert.net/fulltext/?doi=itj.2007.424.427.

[15] *Php zend sort*. Accessed: 22.12.2021. URL: https://github.com/php/php-src/blob/master/Zend/zend_sort.c.

[16] *Quicksort - Wikipedia*. Accessed: 12.12.2021. URL: https://en.wikipedia.org/wiki/Quicksort.

Erklärung

Ich versichere, diese Arbeit selbstständig verfasst zu haben. Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommene Aussagen als solche gekennzeichnet. Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens. Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht. Das elektronische Exemplar stimmt mit allen eingereichten Druck-Exemplaren überein.

Datum und Unterschrift:

Declaration

I hereby declare that the work presented in this thesis is entirely my own. I did not use any other sources and references that the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted hard copies.

Date and Signature: