Aufgabenblatt 6: Schnelles Sortieren

Praktikum: Algorithmen und Datenstrukturen für technische Informatiker

Martin Witte
Karl-Fabian Witte

10. Mai 2017

Der Algorithmus Quicksort wird für eine spezielle Keyverteilung optimiert. Die Optimierung wird erklärt. Eine empirische Messung wird quantitativ aufbereitet und das Ergebnis dargeboten.

Inhaltsverzeichnis

1 Ausgangslage **1**

2 Problemgrößen und Schlüsselverteilung **1**

3 Verbesserungskonzept **1**
3.1 Quicksort mit Insertionsort für kleine Teillisten 2
3.2 Ausnutzung der Keyverteilung 2

4 Messung **3**

5 Ergebnis **8**

1 Ausgangslage

Quicksorts Aymptotische Komplexität liegt bei $O(n) = n\log(n)$ im `best` und im `average case`. Im `worst case` ist die Komplexität bei $O(n) = n^2$. Dieser Fall liegt vor, wenn als Pivotelement immer ein Extrema der zu sortierenden Liste ausgewählt wird und somit die Rekursionstiefe wächst. Für kleine Listen ist Quicksort durch das **Teile und Herrsche Prinzip** nicht geeignet, da es mehr konstanten Aufwand kostet, kleinere Listen zu unterteilen und das Pivotelement mit höherer Wahrscheinlichkeit ein Extrema ist, sodass der `worst case` hier öfters eintreten kann. Zudem summieren sich die Konstanten Operationen (Pivotsuche und Swaping des Pivots) auf, und bei einer Listengröße von 1 ist dies ein Aufwand, der keinen Nutzen hat.

2 Problemgrößen und Schlüsselverteilung

Es soll eine Verbesserung des Quicksortalgorithmus gefunden werde, der die Anzahl der Operationen verkleinert.

Die Elemente in den Listen, besitzen einen Schlüssel k, der aufsteigend sortiert wird. In Abhängigkeit der Anzahl der Elemente N, sind die Schlüssel wie folgt zu wählen:

$$700N \leq k \leq 800N$$

Zudem sind die Schlüssel in diesem Wertebereich zufällig sortiert und es können auch doppelte Schlüssel vorkommen. Es soll aber davon ausgegangen werden, dass die Schlüssel einigermaßen gleichverteilt sind. Die Listengrößen

$$N = 10^i, i = 1, 2, \cdots, 6$$

werden hier untersucht.

3 Verbesserungskonzept

Es werden mehrere Verbesseungskonzepte implementiert.

3.1 Quicksort mit Insertionsort für kleine Teillisten

Um das unnötige Aufteilen von kleineren Listen zu verhindern, wird ab einer gewissen Mindestgröße der rekursive Algorithmus (**Teile**) abgebrochen und durch einen anderen Sortieralgorithmus, hier Insertionsort, ersetzt. [Pareigis, 2017]

Obwohl Insertionsort eine Komplexität von $O(n) = n^2$ hat, ist der Aufwand für kleine Problemgrößen N meist geringer, da der konstante Faktor von Operationen im Quicksortverfahren hier entfällt.

Die perfekte Abbruckgröße S für die Quicksortrekursion ist nicht leicht zu bestimmen, da diese von der Problemgröße und von der Sortiertheit der Liste anhängt. Es wurde daher keine mathematische Formel für die Berechnung dieser gefunden, sodass mehrere Abbruchgrößen ($S = 10, 20, \cdots 50$) für die Messung verwendet werden.

3.2 Ausnutzung der Keyverteilung

Ziel ist es, eine Komplexität von $O(N)$ zu erhalten. Die Informationen über die Schlüsselverteilung wird hier verwendet, um einen möglichst effektiven, jedoch speicherintensiven Algorithmus zu entwerfen.

Der Schlüssel wird als Index für eine extra angelegtes Array eingesetzt [Pareigis, 2017]. Dafür wird der Offset des Wertebereiches subtrahiert.

$$arrIdx = key - N * 700;$$

Es wird hierfür ein Array anlegen, welches auch entsprechend viel Platz zur Verfügung stellt. Dies ist nach der Schlüsseleigenschaft.

$$700N \leq k \leq 800N \rightarrow arrSize = (801 - 700) * N$$

(801 da sowohl $800N$ als auch $700N$ mit eingeschlossen sind)

Da davon ausgegangen wird, dass trotz Gleichverteilung doppelte Schlüssel existieren, wurde das Array um den Faktor 2 erweitert, um im Fall, dass ein Schlüssel vorkommt, dieser auf das nächsten Arrayplatz rutscht. Somit folgt:

$$arrSize = 101 * N * 2, arrIdx = o.key - 700 * N, arr[arrIdx] = o;$$

wobei o ein Objekt der zu sortierenden Liste ist.

Das Ergebnis ist eine fragmentierte Liste. Um die Lücken zu eleminieren, wird die zu sortierende Liste mit den richtigen Werten überschrieben. Es wird durch das Hilfsarray arr durchiteriert und falls ein Plaz belegt ist, wird dieses Element in die Liste übertragen.

Der Pseudocode sieht wie folgt aus:

```
1  sort (list)
2  {
3    N = list.length;
4    arrLen = 2*101*N
5    obj[] arr = obj[arrLen];
6
7    for (i = 0; i < N; i++)
8    {
9      idx = (list[i].key - 700*N)*2;
10     if (arr[idx] not empty ) idx++;
11     arr[idx] = list[i];
12   }
```

```
13
14    pos = 0;
15    for (i = 0; pos < N; i++ )
16    {
17      if ( arr[i] not empty ){
18      list[pos] = arr[i];
19      pos++;
20    }
21  }
```

Die erste Schleife iteriert N mal, die zweite iteriert im `worst case` $arrSize = 101 * N * 2$. Somit erhalten wir eine Laufzeit von $T(N) = 203 * N = O(N)$. Im weiteren Verlauf wird dieser Algorithmus Keysort genannt.

4 Messung

Es werden für die Messung folgende Größen betrachtet:

moves Bewegungsoperationen (Speicher wird neu belegt)

compares Vergleichsoperationen (Vergleiche von Schlüsseln)

time Die Zeit der Sortierung (Hardware abhängig)

Die Listen werden zufällig gefüllt. Das Quicksortverfahren sucht sich das Pivotelement zufällig aus der Teilliste. Um auswertbare Werte zu erhalten, werden alle Messungen 20 mal wiederholt und der Mittelwert gebildet. Als Abbruchsgröße für die Rekursion wird eine Listengröße von $N < S_i$ verwendet, wobei:

$$S_i = 5, 10, 15, 20, 25$$

ist.

Die Ergebnisse werden in doppellogarithmischen Diagrammen qualitativ und zudem in Tabellen quantitativ bewertbar dargestellt.

Tabelle 1: Die Anzahl der Zuweisungsbewegungen in Abhängigkeit zur Listengröße N der Implementationen der Implementationen Quicksort (QS), Qicksort mit Isertionsort ab S (QS+IS S) und dem Keysort

N	QS	QS+IS 5	QS+IS 10	QS+IS 15	QS+IS 20	QS+IS 25	Keysort
10	45	34	39	38	39	40	20
100	684	577	603	652	704	770	200
1000	9125	8091	8322	8768	9337	10013	2000
10000	114742	103880	106316	110912	116453	122629	20000
100000	1378250	1269636	1292256	1339679	1394375	1458038	200000
1000000	16073750	14999379	15210308	15714075	16233517	16855441	2000000

In Abbildung 1 ist nicht direkt etwas zu erkennen, jedoch sieht man in Tabelle 1 deutlich, dass der Insertionsort eine Verbesserung um ca. 1% erbracht hat. Dabei unterschieden sich die Verfahren mit den unterschiedlichen Rekursionsabbruchkonstanten kaum von einander. Der Abbruch bei einer Listengröße von $S = 5$ hat die wenigsten Speicherbewegungen. Der Keysort liegt aber hier deutlich am besten, da er einen Aufwand von $T(N) = 2N$ hat.

Das selbe, wie in der Messung der Bewegungen, gilt auch für die Vergleiche zwischen Quicksort und der Quicksortvariante mit Insertionnsort. Abbildung 1 und Tabelle 1 weisen eine Verbessung von ca. 1% auf. Der Keysort ist trotz linearer Komplexität deutlich über dem Quicksort. Wären die Listen noch größer gewählt worden, so würde Keysort Quicksort überholen. Was aber mit einer Sehr starken Hardware erst möglich wäre.

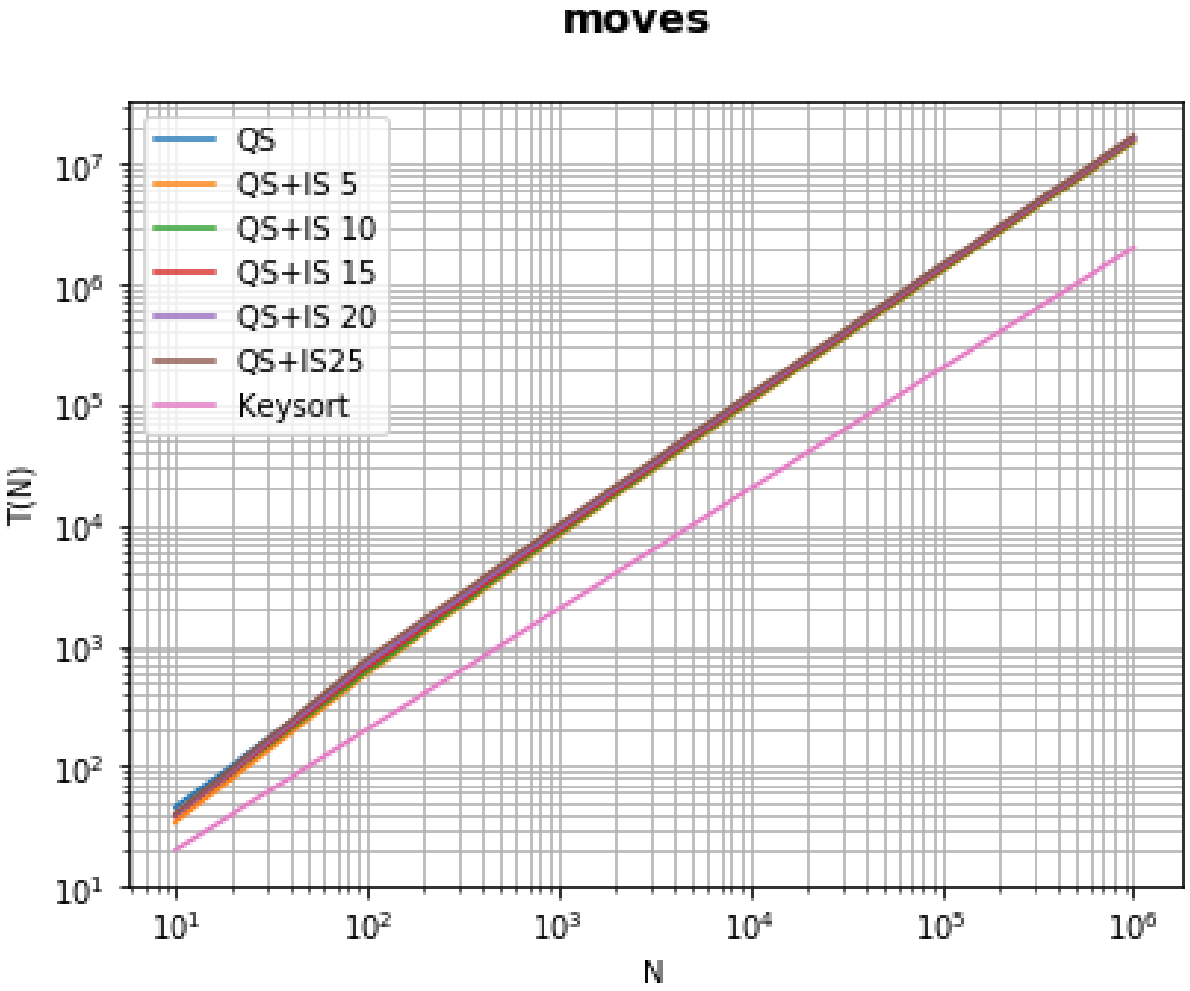

Abbildung 1: Darstellung der Anzahl der Zuweisungen der Listenenlemente $T(N)$ gegen die Problemgröße N

Tabelle 2: Die Anzahl der Schlüsselvergleiche in Abhängigkeit zur Listengröße N der Implementationen Quicksort (QS), Qicksort mit Isertionsort ab S (QS+IS S) und dem Keysort

N	QS	QS+IS 5	QS+IS 10	QS+IS 15	QS+IS 20	QS+IS 25	Keysort
10	41	31	29	27	28	28	1757
100	966	866	850	880	889	948	19821
1000	15677	14592	14671	15020	14924	15352	200867
10000	216537	209437	205745	207648	211078	216255	2009845
100000	2780130	2687801	2683426	2682031	2722770	2752859	20100255
1000000	33817025	33153776	32961227	32891914	33512717	33960675	201004834

Tabelle 3: Die Anzahl der Schlüsselvergleiche und Bewegungen in Abhängigkeit zur Listengröße N der der Implementationen Quicksort (QS), Qicksort mit Isertionsort ab S (QS+IS S) und dem Keysort

N	QS	QS+IS 5	QS+IS 10	QS+IS 15	QS+IS 20	QS+IS 25	Keysort
10	87	66	68	65	68	68	1777
100	1651	1443	1454	1533	1594	1718	20021
1000	24803	22684	22994	23789	24261	25366	202867
10000	331280	313318	312061	318560	327532	338885	2029845
100000	4158381	3957438	3975683	4021711	4117146	4210898	20300255
1000000	49890775	48153155	48171536	48605989	49746235	50816116	203004834

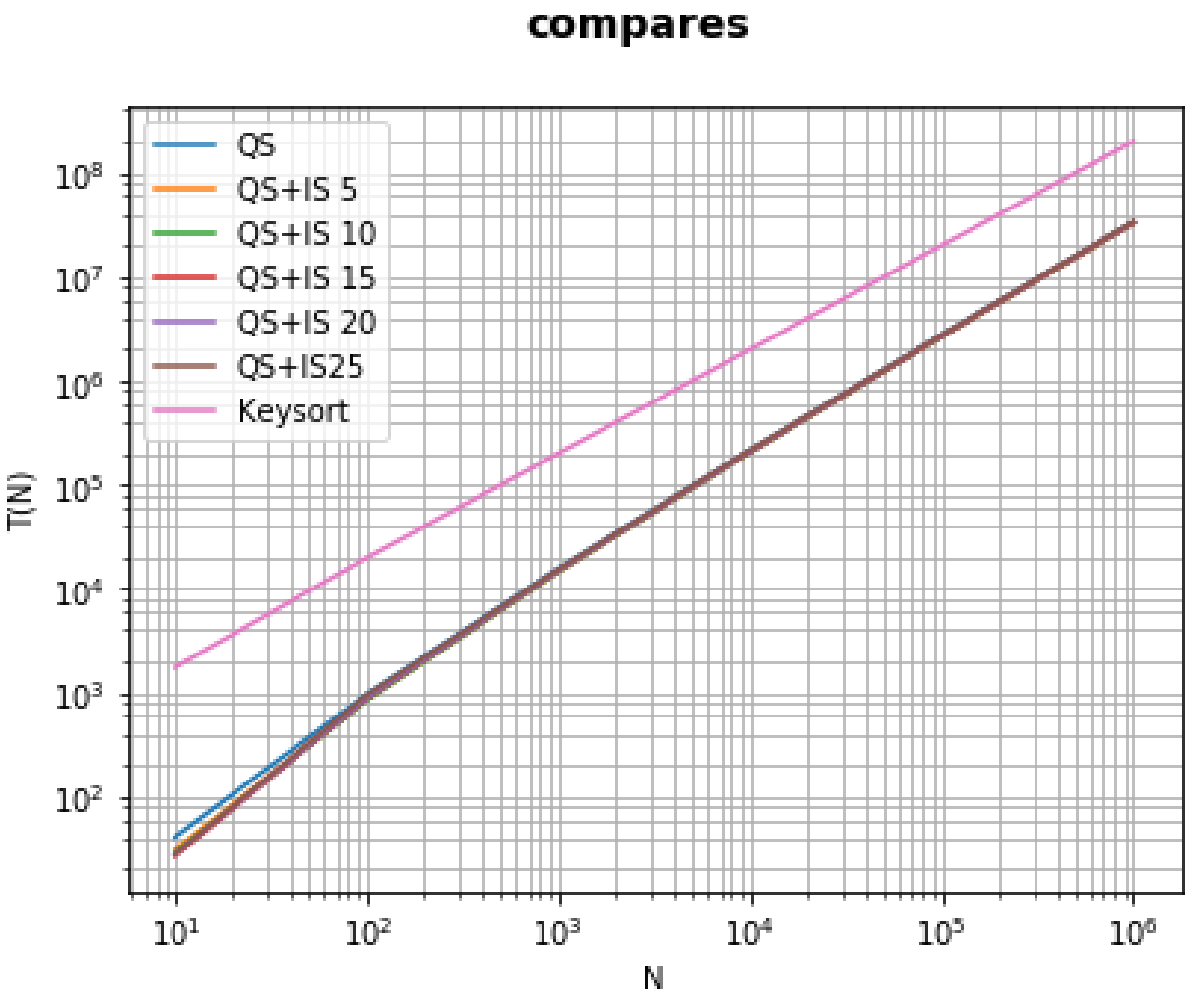

Abbildung 2: Darstellung der Anzahl der Schlüsselvergleiche $T(N)$ gegen die Problemgröße N

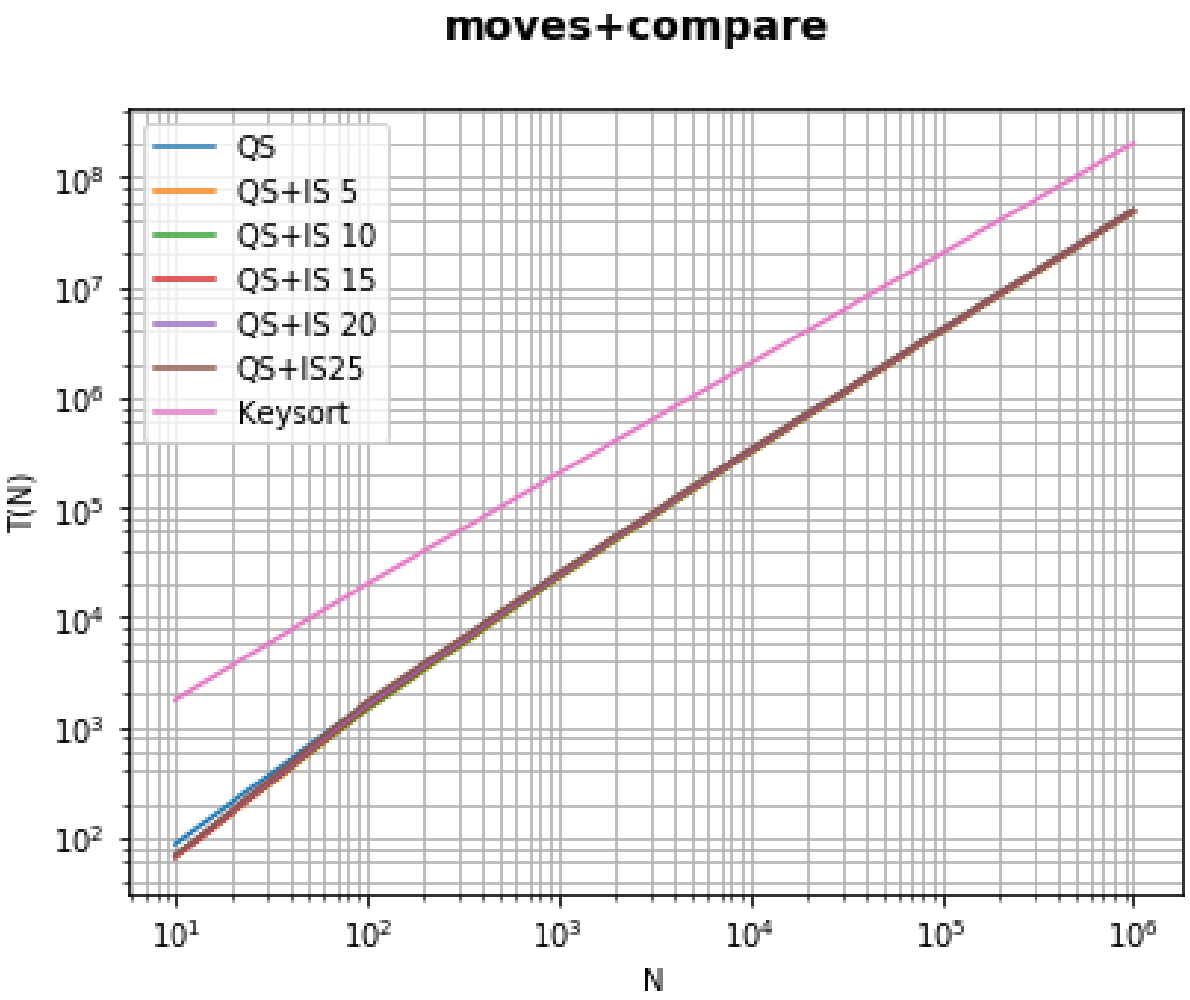

Abbildung 3: Darstellung der Schlüsselvergleiche und Zuweisungen zusammen $T(N)$ die Problemgröße N

Die Summe aus Bewegungen und Vergleichen liegt somit im selben Verhältnis, wie man Abbildung 3 und 3 vernehmen kann. Gerade bei Keysort überwieden deutlich die Vergleiche.

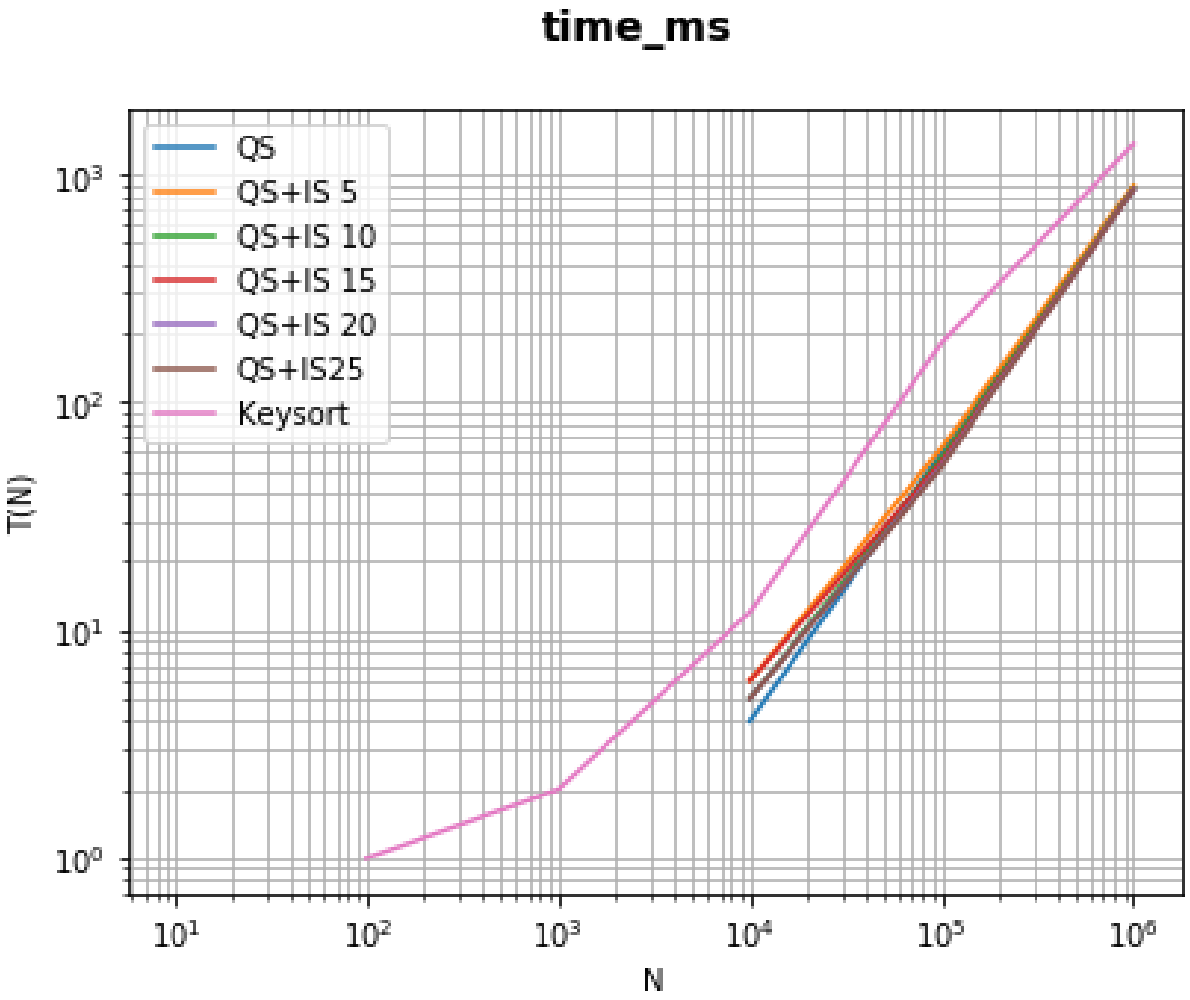

Abbildung 4: Darstellung benötigten Berechnungszeit $T(N)$ in Millisekunden gegen die Problemgröße N

Tabelle 4: Die Berechnungszeit in ms in Abhängigkeit zur Listengröße N der Implementationen Quicksort (QS), Qicksort mit Isertionsort ab S (QS+IS S) und dem Keysort

N	QS	QS+IS 5	QS+IS 10	QS+IS 15	QS+IS 20	QS+IS 25	Keysort
10	0	0	0	0	0	0	0
100	0	0	0	0	0	0	1
1000	0	0	0	0	0	0	2
10000	4	6	5	6	5	5	12
100000	60	63	57	55	53	52	182
1000000	855	890	869	859	859	857	1349

Die Zeitmessung weißt hingegen ein auffälligeres Verhalten auf. Abbildung 4 und Tabelle 4 zeigen, dass die weniger Aufwändige Variante des Quicksorts mit Insertion sort . Da die Rekursion ständig den Stack belastet, hätte den reinen Quicksort mehr Zeit beansprucht, doch hier verhält sich der verbesserte Algorithmus langsamer. Es wird vermutet, dass die Hardware bzw. die übersetzung der Virtuellen Maschine Javas die Rekursion gut optimiert. Beim Keysort ist 200 fache Größe des Hilfsarrays ist hier der entscheidende limitiernede Faktor, da bei sehr großen Listen, der Zugriff auf das Hilfsarray Pagefaults generiert, da dieser Bereich möglicherweise zu Pagefaults führt und zu systemabhängigen Laufzeiten führt.

5 Ergebnis

Eine Verbesserung mit dem Abbruch des rekursiven Verfahrens und der Sortierung der kleinen Teillisten mit Insertionsort ist nicht stark ausgefallen. Lediglich die Operationsanzahl hat sich um 1% verbessert.

Der Keysort ist nur in den Bewegungen schneller als Qicksort. Eine Verbeserung ist er somit nicht wirklich, außer es wird mit Listengrößen $N >> 1M$ bearbeitet, damit die Liniarität des Aufwands den $nlogn$ des Quicksorts überholt.

Eine weitere Überlegung wäre, andere Sortierverfahren auszuprobieren, wie Mergesort.

Literatur

Stephan Pareigis. *Algorithmen und Datenstrukturen für Technische Informatiker*. Hochschule für Angewanste Wissenschaften Hamburg, Department für Informatik, 2017.