Proportion Extend Sort

Studienarbeit von
Jens Neugebauer

am
Institut für Informationssysteme
Fachgebiet Programmiersprachen und Übersetzer
Prof. Dr. R. Parchmann
2004

auf Basis des Artikels
Proportion Extend Sort
von Jing-Chao Chen
Siam J. Comput. Vol. 31, No. 1, pp. 323 - 330

1. Einführung

Ich werde mich hier mit dem Sortier-Algorithmus „Proportion Extend Sort“ von Jing-Chao Chen befassen. Die Fragestellung ist, wie man möglichst schnell und mit möglichst wenig Speicherbedarf eine gegebene Menge von Zahlen der Größe nach sortiert.
Die Idee des Algorithmus ist es, aus dem Orginalproblem 3 Subprobleme zu machen und diese rekursiv zu lösen.
Der Algorithmus funktioniert ähnlich wie ein Quicksort. Beim Quicksort ist es entscheidend, dass das Pivot-Element die zu sortierende Menge in zwei möglichst gleich große Teile aufteilt. Beim Quicksort wird das Pivot zufällig gewählt. Die Frage ist also, wie kann man das besser machen.

Der Trick beim „Proportion Extend Sort“ ist, dass das dritte Subproblem eine Art „Rest darstellt“. Damit wird erreicht, dass der Quicksort nur auf einen Teil angewendet wird und man dann später aus dem sortierten Teil ein besseres Pivot-Element wählen kann.

Beim „Proportion Extend Sort“ wird vorrausgesetzt, dass die Probleme immer von der selben Struktur sind und zwar ein sortierter Teil gefolgt von einem unsortierten Teil. Das dritte Subproblem ist von der selben Größe wie das Orginalproblem, allerdings ist der sortierte Teil vom dritten Subproblem $(p + 1)$mal so groß wie der sortierte Teil des Orginalproblems.
p ist dabei eine gegebene Konstante und $p \geq 1$.

Im Mittel benötigt Proportion Extend Sort $O(n \cdot \ln(n))$ Vergleiche, ebenso im Worst-Case. Es kann sogar gezeigt werden, dass Proportion Extend Sort im Worstcase weniger als $(1/(log(1 + (1/(2p^2 + 2p - 1))) \cdot n\ log(n)$ Vergleiche benötigt. Damit ist Proportion Extend Sort vergleichbar mit vielen anderen Sortieralgorithmen wie z.B. Quicksort, die im Mittel in einer ähnlichen Laufzeitklasse liegen. Im Worst-Case ist Proportion Extend Sort dem Quicksort $(O(n^2))$ aber deutlich überlegen.

2. Der Algorithmus

Zum besseren Verständnis habe ich den Algorithmus in 6 Schritte aufgeteilt, und jedem Schritt ein Bild zugeordnet. Ich beschreibe hier die rekursive Variante, die ich deutlich intuitiver und leichter verständlich finde. Die Stelle wo sich der Algorithmus wieder selbst aufruft habe ich im Bild rot hervorgehoben.

Schritt 1:
Gegeben sei das zu sortierende Array A, welches folgende Struktur habe:
Und zwar bestehe A aus einem sortierten Teil S, gefolgt von einem unsortierten Teil UA. Wenn S leer ist, dann setzen wir S als das erste Element (eine 1-elementige Menge ist immer sortiert).

Schritt 2:
Als erstes soll erreicht werden, dass die ersten $|SE|$ Elemente sortiert sind, wobei $|SE| = (1 + p)\,|S|$.
Dazu sei U der unsortierte Teil von SE, und mid das mittlere Element von S.

Schritt 3:
Jetzt wird mid als Pivot-Element genommen, um U in UL und UR aufzuteilen: UL sind dabei alle Elemente die kleiner sind als mid und UR alle die, welche größer sind als mid. Sei SL der Teil von S, der links von mid steht und SR der Teil von S, der rechts von mid steht.

Schritt 4:
Jetzt werden SR und UL vertauscht. Damit ist erreicht, dass mid an der „richtige Stelle“ innerhalb von SE steht.

Schritt 5:
Außerdem hat man nun 2 Teil-Probleme von der selben Struktur wie das „orginal-Problem“: nämlich einen sortierten Teil (SL bzw. SR) gefolgt von einem unsortierten Teil (UL bzw. UR). Diese beiden Teilprobleme werden rekursiv gelöst.

Schritt 6:
Nachdem dies geschehen ist, ist SE sortiert. Jetzt hat man wieder ein Teilproblem von der selben Struktur wie das „orginal-Problem“: Ein sortierter Teil (SE), gefolgt von einem unsortierten Teil (UE). UE ist dabei der Teil von UA, der nicht zu U gehört. Oder mit anderen Worten: $SE + UE = A$. Dieses Teilproblem wird auch wieder rekursiv gelöst.

Bild 1)

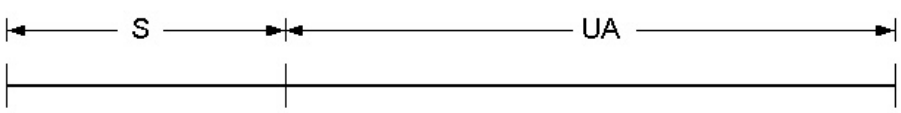
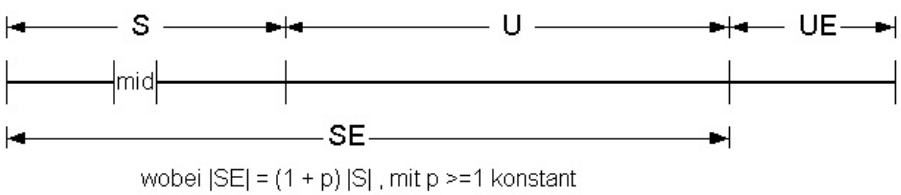


Bild 2)



wobei |SE| = (1 + p) |S| , mit p >=1 konstant

Bild 3)

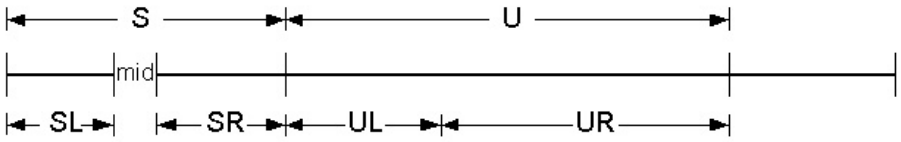


Bild 4)

Bild 5)

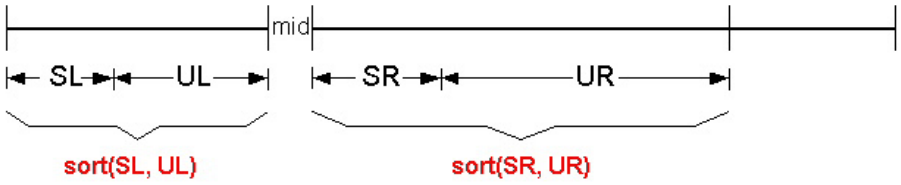


Bild 6)

3. Implementation

An dieser Stelle möchte ich eine Implementation des Algorithmus in einer Java-ähnlichen Sprache zeigen. Ich halte mich hier an die von Jing-Chao Chen gewählte Notation. X bezeichnet dabei ein Array mit der unteren Grenze $x1$ und der oberen Grenze $x2$ (Also der erste Wert des Arrays ist $X[x1]$ und der letzte Wert des Arrays ist $X[x2]$). Bei den Array- und Variablennamen werde ich mich an die Bezeichnungen aus Abschnitt „2) Der Algorithmus“ halten.

sort (S, UA)
if $(s2 < s1)$ *then* $s2 = s1$; // *Überprüfen der Eingabedaten*

if $(s2 \geq ua1)$ *then* $ua1 = s2 + 1$;
if $(ua1 > ua2)$ *then return;* // *Wenn unsortierte Teil leer ist dann fertig!*

(*) $se2 = s1 + (1 + p) \cdot (s2 + 1 - s1) - 1$; // *Die Länge von* $SE = (1 + p) \cdot Länge(Si)$ *berechnen*
// *Wenn das berechnete zulang ist, dann nimm bis Ende*
(**) *if* $((1+p) \cdot (s2+1-s1) > (s2+1-s1)+(ua2+1-ua1))$ *then* $se2 = ua2$;

$mid = (s1 + s2)/2$; // *Die Mitte von S bestimmen = Pivot-Element*
$u1 = s2 + 1$; // *U festlegen = unsortierte Teil von SE*
$u2 = se2$;

// ###### *Partition* ######
// *U in UL (= alles was kleiner als mid) und UR (= alles was größer als mid) aufteilen*
int $j = u1$;
int $k = u2$;
while$(j \leq k)$
    while$(j \leq k$ *AND* $A[j] \leq A[mid])j++$;
    while$(j \leq k$ *AND* $A[k] \geq A[mid])k--$;
    if $(j \geq k)$ *then break;*
    vertausche $A[j]$ *und* $A[k]$;
    $j++$;
    $k--$;

end while;
$ur1 = j$;
//###### *Ende Partition* ######

// SR und UL vertauschen, „s2 − mid“ ist die Länge von ULi
for (*int* $i = 0$; $i \leq s2 - mid$; $i{+}{+}$) *vertausche* $A[s2-i]$ *und* $A[ur1-1-i]$;
$sr1 = ur1 - (s2 - mid)$; *// „s2 − mid“ ist die Länge von UL*
$sr2 = ur1 - 1$;
$ur2 = u2$;
$sl1 = s1$;
$sl2 = mid - 1$;
$ul1 = mid$;
$ul2 = sr1 - 2$; *// Zwischen UL und SR steht nach dem Vertauschen das Element mid! Deshalb -2*
(***) $sort(SL, UL)$;
(***) $sort(SR, UR)$;
$se1 = s1$;
$ue1 = ur2 + 1$;
$ue2 = ua2$;
$sort(SE, UE)$;

end sort;

Anmerkung:

Wenn man anstatt der Zeile (*) „$se2 = ua2$;“ schreibt (Also wenn man im Schritt 2 des Algorithmus für SE ganz A nimmt), so erhält man den Quicksort-Algorithmus. Dann entspricht das „mid“-Element dem Pivot-Element im Quicksort. Die beiden rekursiven Aufrufe sort(SL, UL) und sort(SR, UR) entsprechen den rekursiven Aufrufen beim Quicksort mit dem linken bzw. rechten Teil und der dritte rekursive Aufruf macht nichts (klar, da ja der an den dritten rekursiven Aufruf übergebene Teil dann bereits sortiert ist).

Der Fall, dass der Proportion Extend Sort dem Quicksort entspricht, tritt auch dann ein, wenn die Konstante $p \geq n$ ist (n ist die Anzahl der zu sortierenden Objekte). Dann ist nämlich die „if-Bedingung“ in Zeile (**) immer wahr, und damit ist „$se2 = ua2$;“.

4. Laufzeitverhalten - Theorie

Bemerkung:

Alle Logarithmen sind zur Basis 2.

Theorem 4.1

Der Algorithmus Proportion Extend Sort benötigt im schlechtesten Fall $O(n \cdot \log(n))$ Vergleiche um n Elemente zu sortieren.

Beweis:

p sei eine Konstante, und $p \geq 1$

Vergleiche finden nur an einer Stelle statt und zwar dort, wo U aufgeteilt wird in UL und UR (ich habe den Bereich in der Implementation als Partition gekennzeichnet). Dafür benötigt man genau $|U|$ Vergleiche.

Sei q die Anzahl der Rekursionen von sort, und $W(n)$ die Anzahl der Vergleiche im Worstcase. Dann gilt:

$$W(n) = \sum_{i=1}^{q} |U_i|$$

mit

$$b(A[j] \in U_i) = \begin{cases} 1, & \text{falls } A[j] \in U_i \\ 0, & \text{falls } A[j] \notin U_i \end{cases}$$

erhält man

$$\begin{aligned} W(n) &= \sum_{i=1}^{q} \sum_{j=1}^{n} b([Aj] \in U_i) \\ &= \sum_{j=1}^{n} \sum_{i=1}^{q} b([Aj] \in U_i) \\ &\leq n \cdot max_{j=1}^{n} \left(\sum_{i=1}^{q} b([Aj] \in U_i) \right) \end{aligned}$$

Sei

$$k = max_{j=1}^{n} \left(\sum_{i=1}^{q} b([Aj] \in U_i) \right)$$

Damit erhalten wir:

$$W(n) \leq n \cdot k \tag{1}$$

Aus Schritt 2 des Algorithmus folgt:

$$|U_i| = \begin{cases} |UA_i| & \text{, falls } (p+1)\,p\,|S_i| > |S_i| + |UA_i| \\ p\,|S_i| & \text{, sonst} \end{cases} \tag{2}$$

Aus

$$(p+1)\,p\,|S_i| > |S_i| + |UA_i|$$

folgt

$$|UA_i| < \left(p^2 + p - 1\right)|S_i|$$

und zusammen mit (2):

$$|U_i| \leq \left(p^2 + p - 1\right)|S_i|$$

Jetzt wird auf beiden Seiten $(p^2 + p - 1)|U_i|$ addiert und man erhält

$$|U_i|\left(p^2 + p\right) \leq (p^2 + p - 1)(|S_i| + |U_i|)$$

und damit:

$$|U_i| \leq \frac{(p^2 + p - 1}{p^2 + p}(S_i| + |U_i|) \tag{3}$$

Jetzt benötigt man noch folgende 4 Beziehungen, die sofort aus dem Algorithmus folgen:

$$|U_{i+1}| \leq |UA_{i+1}| \tag{4}$$

$$\begin{aligned} S_{i+1} = SL_i \text{ und } UA_{i+1} = UL_i, \text{ oder} \\ S_{i+1} = SR_i \text{ und } UA_{i+1} = UR_i \end{aligned} \tag{5}$$

$$|UL_i| \leq |U_i| \text{ und}$$
$$|UR_i| \leq |U_i| \tag{6}$$

$$|SL_i| \leq \frac{|S_i|}{2} \text{ und}$$
$$|SR_i| \leq \frac{|S_i|}{2} \tag{7}$$

Aus (4) folgt dann

$$\begin{aligned}
|S_{i+1}| + |U_{i+1}| &\leq |S_{i+1}| + |UA_{i+1}| \\
&\text{mit (5)} \\
&\leq \max(|SL_i| + |UL_i|, |SR_i| + |UR_i|) \\
&\text{mit (6) und (7)} \\
&\leq \frac{|S_i| + |U_i|}{2} + |U_i| \\
&\leq \frac{|S_i| + |U_i|}{2} + \frac{|U_i|}{2} \\
&\text{mit (3)} \\
&\leq \frac{|S_i| + |U_i|}{2} + \frac{(p^2+p-1)}{2\,(p^2+p)}\,(|S_i| + |U_i|) \\
&\leq \frac{p^2+p}{2(p^2+p)}(|S_i| + |U_i|) + \frac{(p^2+p-1)}{2\,(p^2+p)}\,(|S_i| + |U_i|) \\
&\leq \frac{(2p^2+2p-2)}{2(p^2+p)}(|S_i| + |U_i|) \qquad (8)
\end{aligned}$$

Es dürfte klar sein, dass

$$|S_1| + |U_1| \leq n \text{ und } 2 \leq |S_k| + |U_k| \tag{9}$$

(Erinnerung: $k = max_{j=1}^{n} \left(\sum_{i=1}^{q} b([Aj] \in U_i)\right)$)

Daraus folgt:

$$\begin{aligned}
2 &\leq |S_k| + |U_k| \\
&\text{nach (8)} \\
&\leq \frac{(2p^2+2p-1)}{2(p^2+p)}(|S_{k-1}| + |U_{k-1}|) \\
&\leq \cdots \\
&\leq \left(\frac{(2p^2+2p-1)}{2(p^2+p)}\right)^{k-1} \cdot (|S_1| + |U_1|) \\
&\text{nach (9)} \\
&\leq \left(\frac{(2p^2+2p-1)}{2(p^2+p)}\right)^{k-1} \cdot n
\end{aligned}$$

Diese Ungleichung wird jetzt nach k aufgelöst

$$\begin{aligned}
2 &\leq \left(\frac{(2p^2+2p-1)}{2(p^2+p)}\right)^{k-1} \cdot n \\
2 \cdot \left(\frac{2(p^2+p)}{(2p^2+2p-1)}\right)^{k-1} &\leq n \\
log(2) + log\left(\left(\frac{2(p^2+p)}{(2p^2+2p-1)}\right)^{k-1}\right) &\leq log(n) \\
1 + (k-1) \cdot log\left(\frac{2(p^2+p)}{(2p^2+2p-1)}\right) &\leq log(n) \\
k \cdot log\left(\frac{2(p^2+p)}{(2p^2+2p-1)}\right) - log\left(\frac{2(p^2+p)}{(2p^2+2p-1)}\right) &\leq log(n) \\
k \cdot log\left(\frac{2(p^2+p)}{(2p^2+2p-1)}\right) &\leq log(n), \text{ da } \left(\tfrac{2(p^2+p)}{(2p^2+2p-1)}\right) \leq 1 \\
k &\leq \frac{log(n)}{log\left(\frac{2(p^2+p)}{(2p^2+2p-1)}\right)} \\
k &\leq \frac{log(n)}{log\left(1 + \frac{1}{(2p^2+2p-1)}\right)}
\end{aligned}$$

Und schließlich mit (1):

$$W(n) \leq \frac{n \cdot log(n)}{log\left(1 + \frac{1}{(2p^2+2p-1)}\right)}$$

❑

5. Laufzeitverhalten - Messung

Alle Algorithmen wurden in Java 1.4.0 implementiert und alle Algorithmen sortieren Objekte vom Typ Comparable. Die Messungen liefen auf einem Pentium III/1100MHz mit 448MB Ram. Als Betriebssystem lief ein Windows XP Professional, Version 2002. In Diagramm 5.1 ist die Laufzeit in abhängingkeit von der Anzahl der zu sortierenden Objekte dargestellt. Die Einheiten der y-Achse sind Millisekunden, die Einheit der x-Achse sind 1000 zufällig erzeugte Objekte. Alle Messungen wurden 10mal wiederholt, die dargestellten Werte sind der Mittelwert aus den Ergebnissen. Ich habe die Messungen mit maximal 1.000.000 Objekten ausgeführt. Bei mehr Objekten gab es Speicherprobleme, sodass Windows anfing auf Festplatte auszulagern.
„PES10“ steht für „Proportion Extend Sort mit p=10“, „PES16“ steht für „Proportion Extend Sort mit p=16“, „Quick“ steht für „Quicksort“ und „Clever Quick“ für „Clever Quicksort“. „PES v2 16“ steht für „Proportion Extend Sort Verion2 mit p=16“, dass ist ein leicht veränderter Proportion Extend Sort. Die Erklärung und Funktionsweise erfolgt in Abschnitt „6) Verbesserungen“.

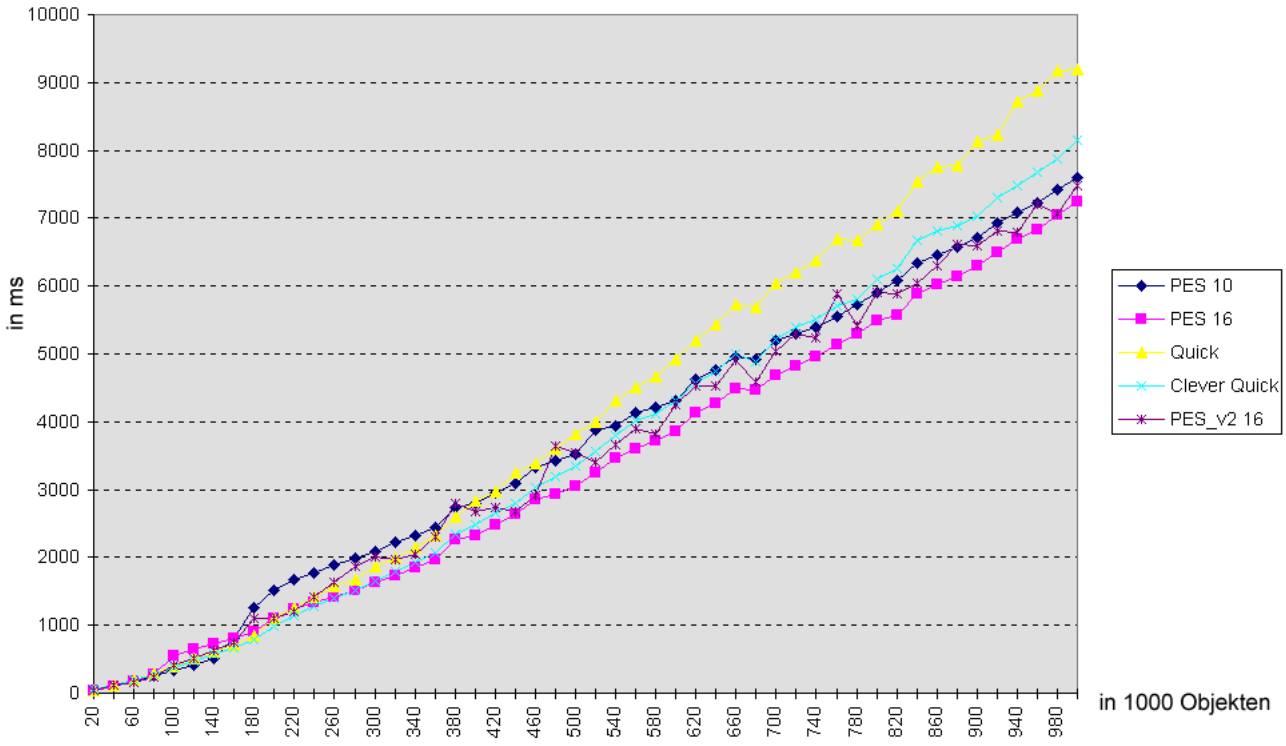


Diagramm 5.1) Laufzeiten bei zufällig erzeugten Objekten

Man erkennt, dass Proportion Extend Sort für zufällig erzeugte Objekte schneller ist als Quicksort und auch geringfügig schneller als Cleverquicksort. Tabelle 5.2) enthält einige der Messwerte, die Diagramm 5.1) zugrunde liegen.

n	PES10	PES16	Quick	Clever Quick	PES v2 16
100.000	327ms	557ms	396ms	367ms	404ms
200.000	1.512ms	1.105ms	1.095ms	994ms	1.102ms
300.000	2.080ms	1.626ms	1.872ms	1.658ms	2.013ms
400.000	2.793ms	2.324ms	2.833ms	2.472ms	2.678ms
500.000	3.532ms	3.042ms	3.828ms	3.346ms	3.546ms
600.000	4.317ms	3.851ms	4.916ms	4.303ms	4.259ms
700.000	5.197ms	4.678ms	6.046ms	5.210ms	5.041ms
800.000	5.912ms	5.492ms	6.909ms	6.110ms	5.908ms
900.000	6.707ms	6.292ms	8.129ms	7.030ms	6.593ms
1.000.000	7.600ms	7.243ms	9.189ms	8.143ms	7.488ms

Tabelle 5.2) Laufzeiten bei n zufällig erzeugten Objekten

Was aus der Tabelle und dem Diagramm nicht hervorgeht ist, dass sowohl Proportion Extend Sort als auch Quick- und Clever Quicksort immer relativ nah am Mittlerwert liegen. Proportion Extend Sort Version2 hingegen streut doch ziemlich stark, ist also sehr abhängig von den Zufallszahlen (Bis hin zu Ausreißern mit 30% Differenz zum Mittelwert). Vermutlich kommt daher auch diese etwas unruhige Kurve von Proportion Extend Sort Version2 in Diagramm 5.1.
In Tabelle 5.3 ist zu sehen, wieviele Vergleich die einzelnen Algorithmen im Mittel benötigen (alle Messungen wurden 10mal wiederholt). Auch hier ist Proportion Extend Sort dem Quick- und dem Clever Quicksort überlegen.

n	PES10	PES16	Quick	Clever Quick	PES v2 16
100.000	1.805.721	1.902.543	2.496.087	2.225.420	1.819.879
200.000	3.998.328	3.836.010	5.367.451	4.730.734	3.828.752
300.000	6.009.527	5.851.713	8.418.881	7.326.198	5.949.477
400.000	8.064.123	7.946.810	11.431.532	9.910.759	7.995.516
500.000	10.178.374	10.101.322	14.468.936	12.577.679	10.138.632
600.000	12.342.174	12.306.748	17.558.789	15.223.098	12.244.992
700.000	14.548.340	14.549.400	20.925.489	18.063.700	14.443.680
800.000	16.790.306	16.831.442	23.587.849	20.614.158	16.628.059
900.000	19.069.005	19.148.228	27.130.960	23.477.182	18.895.718
1.000.000	21.377.617	21.503.183	30.037.749	26.383.179	21.166.944

Tabelle 5.3) Anzahl Vergleiche bei n zufällig erzeugten Objekten

In Tabelle 5.4 ist dargestellt, wie oft sich die Algorithmen selbst aufrufen müssen, um n zufällige Objekte zu sortieren (Wieder als Mittelwert über 10 Messungen). Dabei fiel auf, dass Proportion Extend Sort deutlich mehr Rekursionen benötigt als Quick- oder Clever Quicksort.

n	PES10	PES16	Quick	Clever Quick	PES v2 16
100.000	382.148	555.445	92.927	87.797	293.698
200.000	1.127.681	827.741	185.833	175.559	603.085
300.000	1.423.152	1.059.599	278.862	263.298	933.913
400.000	1.670.245	1.326.007	371.785	351.120	1.100.587
500.000	1.938.468	1.637.567	464.714	438.958	1.334.247
600.000	2.239.090	1.990.621	557.607	526.634	1.512.418
700.000	2.571.360	2.382.026	650.568	614.540	1.744.320
800.000	2.936.427	2.8163.94	743.543	702.377	1.957.302
900.000	3.337.056	3.293.931	836.485	789.977	2.209.968
1.000.000	3.767.821	3.819.771	929.387	877.914	2.473.784

Tabelle 5.4) Anzahl Rekursionen bei n zufällig erzeugten Objekten

5.1 Messungen im Worst-Case

An dieser Stelle werde ich die obigen Messungen nochmal für den Worstcase, also dass die Objekte bereits sortiert sind, wiederholen.

Anmerkung:

Für den Clever Quicksort ist dies nicht der Worstcase, da Clever Quicksort das Pivot-Element zufällig wählt. Und für Proportion Extend Sort Version2 ist dies sogar der Bestcase!

In Diagramm 5.5 ist deutlich zu erkennen, dass Quicksort im Worstcase in einer anderen Laufzeitklasse liegt, nämlich $O(n^2)$.

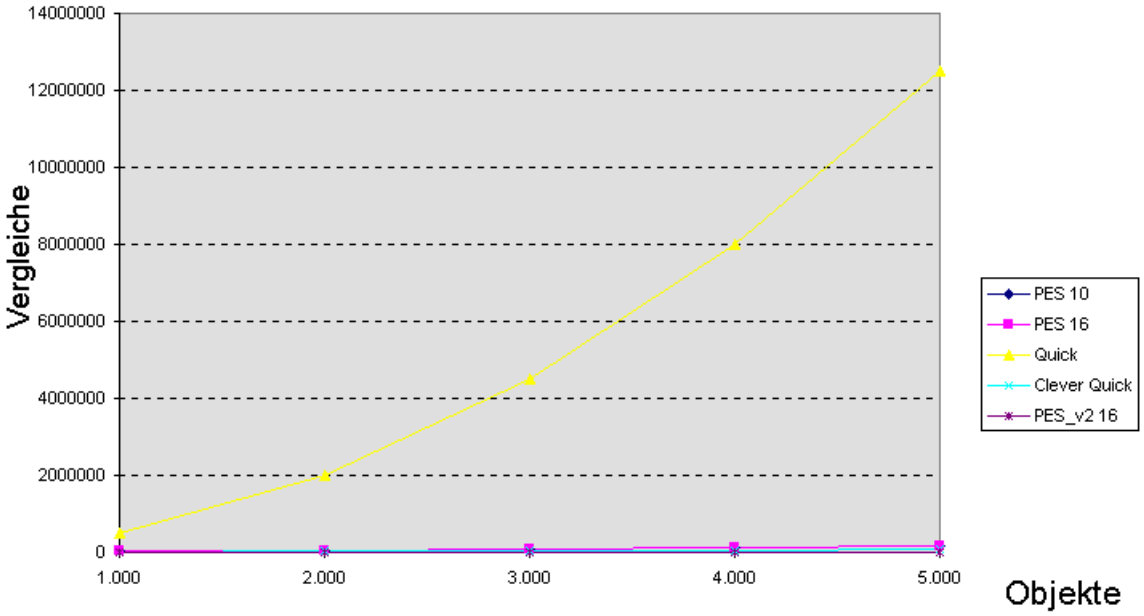


Diagramm 5.5) Laufzeiten im „Worstcase“

Bereits bei 5.000 Objekten benötigt der Quicksort über 12.000.000 Vergleiche, während der Proportion Extend Sort bei ungefähr 150.000 Vergleichen lag. Mit 10.000 Objekten kam es dann beim Quicksort zu Überlauf-Fehlern. Um aber sinnvoll Zeiten messen zu können, benötigt man bei den anderen Algorithmen mindestens 20.000 Objekte. Deshalb habe ich den Quicksort bei den weiteren „worstcase“ Messungen weggelassen.

Diagramm 5.6 zeigt die Zeit, die die Algorithmen benötigen um n „worstcase“ Objekte zu sortieren.

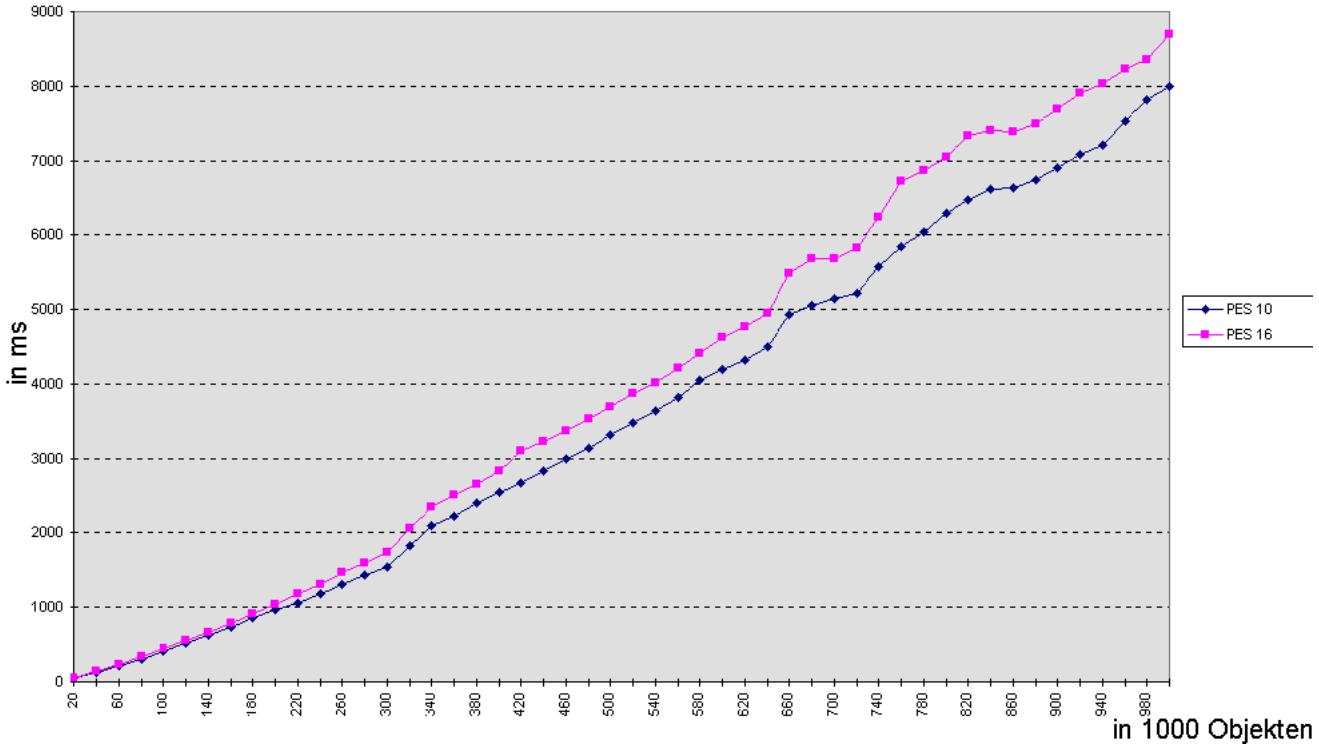


Diagramm 5.6) Laufzeiten im „Worstcase“

Aus dem Diagramm kann man gut erkennen, dass der Proportion Extend Sort im Worstcase etwas schlechter als „Lineare Laufzeit“ ist. Man kann also schon erahnen, dass er in der Laufzeitklasse $O(n \cdot log(n))$ liegt (Was uns die Theorie ja bereits gesagt hatte).
Bemerkenswert ist, dass die Zeit, die der Proportion Extend Sort im Worstcase benötig um n Objekte zu sortieren teilweise sogar geringer ist als die, die er im Mittel benötigt. Das liegt aber vermutlich daran, dass für die theoretische Betrachtung nur die Anzahl der Vergleiche eine Rolle spielt, und nicht das Kopieren der Daten. Im Worstcase werden zwar die meisten Vergleiche benötigt aber es müssen nie irgendwelche Objekte im Array verschoben werden.

In Tabelle 5.7 habe ich für einige ausgewählte Anzahlen von Objekten die gemessenen Laufzeiten, in Tabelle 5.8 die Anzahl der benötigten Vergleiche und in Tabelle 5.9 die Anzahl der benötigten Rekursionen aufgeführt.

n	PES10	PES16
100.000	4.17ms	4.48ms
200.000	962ms	1.042ms
300.000	1.547ms	1.739ms
400.000	2.541ms	2.830ms
500.000	3.312ms	3.688ms
600.000	4.196ms	4.624ms
700.000	5.150ms	5.691ms
800.000	6.285ms	7.037ms
900.000	6.911ms	7.683ms
1.000.000	8.002ms	8.704ms

Tabelle 5.7) Laufzeiten bei n „worstcase“ Objekten

n	PES10	PES16
100.000	4.042.085	4.640.006
200.000	8.508.317	9.913.606
300.000	13.219.201	15.460.970
400.000	18.109.129	21.287.488
500.000	23.224.686	26.936.820
600.000	28.194.378	32.868.850
700.000	33.299.286	38.881.218
800.000	38.555.652	44.737.348
900.000	43.798.827	50.563.128
1.000.000	48.984.703	56.543.488

Tabelle 5.8) Anzahl Vergleiche bei n „worstcase“ Objekten

n	PES10	PES16
100.000	423.910	393.478
200.000	847.462	783.700
300.000	1.270.723	1.173.400
400.000	1.693.756	1.562.875
500.000	2.116.462	1.952.605
600.000	2.539.378	2.342.152
700.000	2.962.105	2.731.762
800.000	3.384.556	3.121.753
900.000	3.807.028	3.511.201
1.000.000	4.229.527	3.900.775

Tabelle 5.9) Anzahl Rekursionen bei n „worstcase“ Objekten

5.2 Messungen zum Parameter p

Der Proportion Extend Sort teilt das ursprüngliche Problem in 3 Subprobleme auf, wobei der sortierte Teil vom dritten Subproblem $(p+1)$mal so groß ist wie der sortierte Teil des Orginalproblems. p ist dabei ein frei wählbarer Parameter mit der Bedingung, dass $p \geq 1$. An dieser Stelle soll untersucht werden, wie sich in der Praxis die Laufzeit des Proportion Extend Sort in Abhängigkeit von p verhält.

Bemerkung:

Aus der Theorie wissen wir bereits, dass der Proportion Extend Sort im Worstcase weniger als $(1/(log(1+(1/(2p^2+2p-1))) \cdot n\ log(n)$ Vergleiche benötigt.

Auch hier wurden alle Messungen 10mal wiederholt, die dargestellten Werte sind wieder die Mittelwert aus den Ergebnissen. Zu sortieren waren jeweils 300.000, 500.000 und 700.000 zufällig erzeugte Objekte.

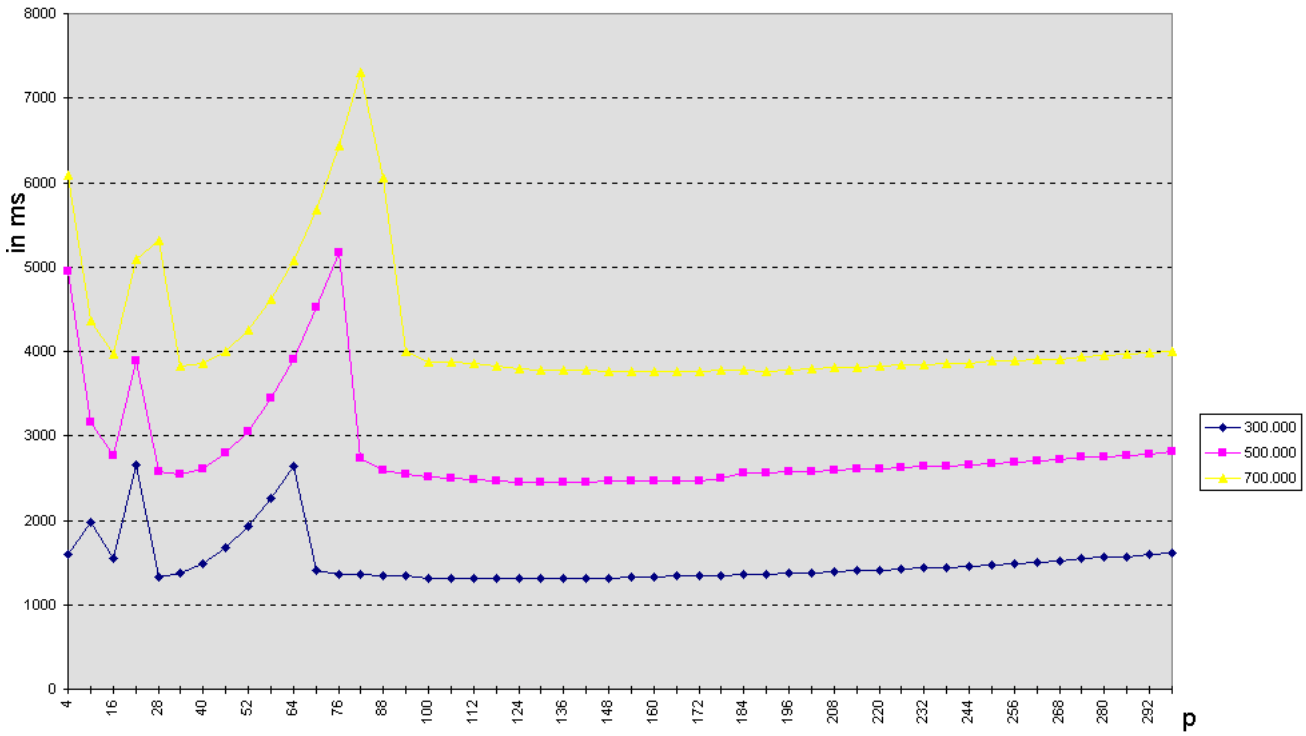


Diagramm 5.9) Laufzeiten bei zufällig erzeugten Objekten

Für den doch etwas seltsamen Verlauf von Diagramm 5.9, gerade im Bereich von $p = 4$ bis $p = 100$ habe ich keine Erklärung. Das sich die Kurve für großes p langsam der Laufzeit des Quicksort annähern wird, ist noch einzusehen (für $p >= n$ entspricht der Proportion Extend Sort dem Quicksort). Interessant ist auch, daß das Diagramm 5.9 für „p=16“ ein lokales Minimum zu haben scheint, unabhängig von der Anzahl der zu sortierenden Objekte. $p = 16$ ist nämlich genau der Wert, den Jing-Chao Chen in seinem Artikel als schnellste Version für Proportion-Extend Sort vorgeschlagen hat.

In Tabelle 5.10 habe ich nochmal die Messwerte aufgeführt, die Diagramm 5.9 zugrunde liegen. Außerdem ist dort auch die Anzahl der jeweils benötigten Vergleiche und Rekursionen aufgeführt.

p	Laufzeit	Vergleiche	Rekursionen
4	6.094ms	15.040.904	3.684.616
10	4.360ms	14.547.101	2.571.899
16	3.971ms	14.550.027	2.383.889
22	5.090ms	14.661.885	2.842.047
28	5.316ms	15.170.175	3.465.331
34	3.825ms	14.606.563	2.167.462
40	3.852ms	14.424.324	1.911.971
46	4.001ms	1.440.7786	1.974.056
52	4.248ms	14.436.994	2.186.147
58	4.615ms	14.544.349	2.456.110
64	5.083ms	14.654.333	2.757.616
70	5.669ms	14.918.023	3.132.784
76	6.428ms	15.173.837	3.510.464
82	7.306ms	15.402.625	3.783.706
88	6.054ms	15.265.604	3.388.932
94	4.002ms	14.986.703	2.724.930
100	3.877ms	14.930.143	2.436.465
106	3.873ms	14.899.518	2.244.803
112	3.856ms	1.4862.043	2.087.129
118	3.827ms	14.803.183	1.949.205
124	3.796ms	14.745.564	1.848.127
130	3.778ms	14.727.909	1.774.227
136	3.774ms	14.731.160	1.712.925
142	3.771ms	14.721.860	1.663.309
148	3.760ms	14.706.149	1.632.823
154	3.762ms	14.677.354	1.612.522

p	Laufzeit	Vergleiche	Rekursionen
160	3.761ms	14.652.156	1.605.400
166	3.765ms	14.617.529	1.603.654
172	3.763ms	14.579.036	1.604.993
178	3.772ms	14.544.340	1.610.594
184	3.777ms	14.535.198	1.618.868
190	3.770ms	14.536.482	1.628.107
196	3.785ms	14.537.425	1.638.257
202	3.795ms	14.531.978	1.649.047
208	3.803ms	14.524.408	1.659.795
214	3.815ms	14.511.254	1.671.656
220	3.819ms	14.497.883	1.682.818
226	3.834ms	14.482.744	1.695.387
232	3.838ms	14.465.534	1.707.505
238	3.854ms	14.448.669	1.719.631
244	3.857ms	14.430.788	1.733.094
250	3.883ms	14.410.497	1.746.372
256	3.888ms	14.394.404	1.759.161
262	3.900ms	14.407.402	1.773.477
268	3.912ms	14.414.193	1.789.574
274	3.933ms	14.423.012	1.806.103
280	3.948ms	14.429.519	1.821.811
286	3.970ms	14.433.524	1.836.945
292	3.990ms	14.437.142	1.855.370
298	4.002ms	14.438.622	1.872.577

Tabelle 5.10) Anzahl Vergleiche, Rekursionen und Laufzeiten bei n zufällig erzeugten Objekten

6. Verbesserungen

Eine Mögliche Verbesserung für den Proportion Extend Sort aber auch für den Quicksort ist, dass man beim rekursiven Aufruf zuerst das kleinere Problem löst. Dadurch spart man sich etwas Speicher.
Im Proportion Extend Sort würde das bedeuteten, dass man in Schritt 5 zuerst bestimmt, welches Problem kleiner ist (SL, UL bzw. SR, UR) und dieses dann löst. In meiner Implementation müsste man die beiden Zeilen (***) durch folgende 7 ersetzt:

if $(ul2 - sl1 > ur2 - sr1)$ *then*

$sort(SL, UL);$
$sort(SR, UR);$

else

$sort(SL, UL);$
$sort(SR, UR);$

end if

6.1 Proportion Extend Sort Version2

Mir ist bei den Messungen aufgefallen, dass Proportion Extend Sort relativ viele Rekursionen benötigt um ein Problem zu lösen. Und da der Algorithmus stark von der Länge des bereits sortierten Teils abhängt, hab ich nach einer Möglichkeit gesucht, diesen zu vergrößern. Ich habe dann die einfachste Möglichkeit implementiert, nämlich dass man am Anfang des Algorithmus schaut, wieviel von dem unsortierten Teil man zu dem bereits sortierten Teil hinzufügen kann (Das geht in linearer Laufzeit).
Dazu muss man die erste Zeile des Algorithmus einfach durch folgende Zeilen ersetzen:

if $(s2 < s1)$ *then* $s2 = s1;$ *// Überprüfen der Eingabedaten*
for$(s2++;\ s2 \leq ua2;\ s2++)$ *if* $(A[s2-1].compareTo(A[s2]) > 0)$ *break;*
$s2--;$

Die Messungen zeigen, dass dieses „lineare vorsortieren“ nicht groß stört. Von der Laufzeit ist Version2 nur geringfügig langsamer als das Orginal, bei der

Anzahl der Vergleich ist Version2 sogar besser (immer gleiches p vorrausgesetzt). Die Anzahl der Rekursionen ist bei Version2 deutlich geringer als beim „normalen“ Proportion Extend Sort.

Leider ist bei der Version2 anzumerken, dass sie bei meinen Messungen sehr stark teilweise vom Mittelwert abweicht (in Extremfällen sogar bis zu 20Prozent). Das muss mit den Zufallszahlen zutun haben. Wie das jetzt aber genau kommt, habe ich nicht weiter untersucht.

6.2 Ripple Swap

Jing-Chao Chen schläg in seinem Artikel noch eine Methode names „Ripple Swap“ vor, um bei dem Algorithmus die Gesamtzahl der Datenbewegungen zu verringern. Wie das genau funktionieren soll, konnte ich nicht erfahren.

6.3 Rekursionen sparen

Ganz am Ende meiner Studienarbeit ist mir noch aufgefallen, dass sich Proportion Extend Sort zuerst selbst Rekursiv aufruft, und dann testet ob der unsortierte Teil leer ist und sich dann ggf. beendet. Ich denke es wäre deutlich besser, man würde erst diesen Test durchführen und dann ggf. die Rekursionen durchführen.
Allerdings habe ich diese Verbesserungsmöglichkeit nicht mehr Implementiert.

7. Quellenangabe

- Jing-Chao Chen, Proportion Extend Sort
  Siam J. Comput. Vol. 31, No. 1, pp. 323 - 330