PROPORTION EXTEND SORT

JING-CHAO CHEN

Abstract. PROPORTION EXTEND SORT is a new sorting algorithm, the basic principle of which is similar to PROPORTION SPLIT SORT. This algorithm sorts a sequence by constructing three subproblems, using a QuickSort-like pivot technique and solving recursively each subproblem. The original problem and these subproblems are of such a structure: a sorted subsequence followed by an unsorted subsequence. The size of the original problem always equals the size of the third subproblem, but in general, the sorted subsequence of the third subproblem is \( p + 1 \) times as much as the sorted subsequence of the original, where \( p \) is a fixed positive constant. The worst case number of comparisons required by this algorithm is less than \( \frac{1}{2} \log(1 + 1/(2p^2 + 2p - 1))n \log n \) for \( p > 1 \). Empirical results show that the average number of comparisons is close to \( n \log n - O(n) \) for some \( p \). From our experiments for sorting integers, when \( p = 10 \), this algorithm is yet faster, on average, than PROPORTION SPLIT SORT which is faster than CLEVER QUICKSORT.

Key words: algorithm, partition, sort, quick sort, insertion sort

AMS subject classifications: 68P10, 68W30

PHI. S000753978342903

1. Introduction. It has been studied for a long time whether there exists a simple, practical, and efficient sorting algorithm that sorts \( n \) elements using constant extra space and making \( O(n \log n) \) comparisons in the worst case, and has an expected number of comparisons approaching \( \log(n!) = n \log n - 1.442695n \). (All logarithms throughout the paper are base two.) Notice that \( \log(n!) \) is the lower bound on the worst and average case number of comparisons for comparison based sequential sorting algorithms.

MERGESORT, BINARY INSERTION SORT, and WEAK-HEAP SORT [6], etc. are sorting algorithms approaching this lower bound in terms of the number of comparisons. MERGESORT requires \( n \log n - 1.2645n \) comparisons [1], [2] on average, but uses extra storage of length \( n \). BINARY INSERTION SORT is not efficient in practice, primarily because of its \( O(n^2) \) data movements. WEAK-HEAP SORT introduced by Dutton (1933) is the fastest variant of HEAPSORT [3], [5], and the average number is conjectured to be approximately \((n - 0.5) \log n - 0.413n\). Unfortunately, WEAK-HEAP SORT uses \( n \) extra bits, and by the empirical results of [8], the comparison plus exchange total required exceeds that required by the best-of-three version of QUICKSORT called CLEVER QUICKSORT.

Most versions of QUICKSORT [4] require \( O(n^2) \) comparisons in the worst case but \( O(n \log n) \) comparisons in the average case. CLEVER QUICKSORT is a practical version of QUICKSORT and runs in approximately \( 1.188(n + 1) \log(n - 1) - 2.255n + 2.507 \) (sec [7]) comparisons on average.

PROPORTION SPLIT SORT introduced by Chen [8] splits a sequence into two blocks in the ratio of \( 1 : p - 1 \), then divides them into four blocks in a QuickSort-
like pivot step, finally, sorts recursively the left two blocks and the right two blocks separately. The worst case number of comparisons of \textsc{Proportion Split Sort} is bounded by $1/\log(2p/(2p-1))n \log n$ for $p > 1$ [8]. The simulation results of [8] revealed that for some $p$ (e.g., $p = 16$), the average number of comparisons and exchanges required by \textsc{Proportion Split Sort} is fewer than that required by \textsc{Clever Quicksort}.

This paper introduces a new algorithm called \textsc{Proportion Extend Sort}. The basic principle of this algorithm is similar to \textsc{Proportion Split Sort}. \textsc{Proportion Extend Sort} assumes that a sequence to be sorted has such a structure: a sorted subsequence followed by an unsorted subsequence, then constructs three subproblems which have the same structure as the original, and solves recursively each subproblems. In general, the sorted subsequence of the third subproblem is $p + 1$ times as much as the sorted subsequence of the original, where $p$ is a positive constant given. The worst case number of comparisons of this algorithm is bounded by $1/\log(1 + 1/(2p^2 + 2p - 1))n \log n$ for $p \geq 1$. By experimental results, the performance of \textsc{Proportion Extend Sort} is better than that of \textsc{Proportion Split Sort}.

![Diagram](image)

**Fig. 2.1.** The basic principle of the sorting algorithm.

2. **The algorithm.** The algorithm assumes that an initial array of arbitrary values $A[1..n]$ is given; it then sorts the array $A$ in ascending order. We will design and analyze the algorithm by the following subarrays: $S_i = A[i..i+2]$, $S_{E_i} = A[i+1..i+2]$, $U_i = A[i+1..i+2]$, $U_{A_i} = A[i+1..i+2]$, $U_{L_i} = A[i+1..i+2]$, $U_{R_i} = A[i+1..i+2]$, $S_{L_i} = A[1..i+1]$, $S_{R_i} = A[1..i+1]$, and $U_{E_i} = A[i+1..i+2]$. $|S_i|$ will denote the length of $S_i$. Swap($i,j$), which appears below, is a procedure that interchanges the values in $A[i]$ and $A[j]$. The basic idea of \textsc{Proportion Extend Sort} is to regard the array $A$ as a sequence which has such a structure: a sorted subsequence $S_i$ followed by an unsorted subsequence $U_{A_i}$, create three subproblems: ($S_{L_i}, U_{L_i}$), ($S_{R_i}, U_{R_i}$), and ($S_{E_i}, U_{E_i}$), using a QuickSort-like pivot technique, and solve recursively each subproblem. The basic idea of the algorithm is shown in Figure 2.1.
PROPORTION EXTEND SORT

Step 1. Suppose that the first $|S_i|$ elements are sorted, and we want to extend that to $|SE_i|=(1+p)|S_i|$ elements, where $p$ is a positive constant. That is, $sr1 := s1$, $sr2 := s1+(1+p)|S_i|-1$. Let the median of $S_i$ be $s$ and the unsorted subsequence in the $SE_i$ be $U_i$. Use $s$ to partition $U_i$ into two subsequences $UL_i$ and $UR_i$ in a QuickSort-like pivot step, such that $\max(UL_i) \leq s \leq \min(UR_i)$.


```plaintext
for i := 0 to sr2 - sr1 do Swap (sr2 - i, ul2 - i).
```

Step 3. After Step 2 is done, the two subproblems $(SL_i, UL_i)$ and $(SR_i, UR_i)$ have the same structure as the original (a sorted subsequence followed by an unsorted subsequence). Let the right subsequence contiguous to $SE_i$ be $UE_i$. After the two subproblems are solved, the subproblem $(SE_i, UE_i)$ has also the same structure as the original. Hence, the algorithm is called recursively to sort $(SL_i, UL_i)$, $(SR_i, UR_i)$, and $(SE_i, UE_i)$.

We implement the algorithm by way of a recursive routine. It is easy to replace this recursive routine by way of an iterative routine. Next we describe the main algorithm ProportionExtendSort with Pascal-like procedures.

ProportionExtendSort($S_i, U_A_i$) denotes that it sorts the two contiguous subsequences $S_i$ and $U_A_i$ by using the parameters $s1, s2$, and $ur2$ in $S_i$ and $U_A_i$, where $S_i$ is a sorted subsequence and $U_A_i$ is an unsorted subsequence. Notice, in this procedure, the parameter $uw$ in $U_A_i$ is not used, since $uw = s2 + 1$. Initially $S_i = A[1..1]$, $U_A_i = A[2..n]$.

```plaintext
ProportionExtendSort($S_i, U_A_i$) if $s2 < s1$ then $s2 := s1$
if $U_A_i$ = empty then return $s2 := s1 + (1+p)\cdot|S_i| - 1$
(2.1) if $(1+p)\cdot|S_i| > |S_i| + |U_A_i|$ then $sr2 := s2$
$mid := ((s1 + s2)/2)$
$ul := s2 + 1$, $ur2 := sr2$
$ur1 := Partition(mid, U_i)$
(2.2) for $i = 0$ to $s2 - mid$ do
  Swap (s2 - i, ur1 - 1 - i)
$sr1 := ur1 - (s2 - mid)$, $sr2 := ur1 - 1$
$uw2 := ur2$
$sl1 := s1$, $sl2 := mid - 1$
$uw2 := sr1 - 2$
(2.3) ProportionExtendSort($SL_i, UL_i$)
(2.4) ProportionExtendSort($SR_i, UR_i$)
$sr1 := s1$
$uw2 := ur2$
ProportionExtendSort($SE_i, UE_i$)
end ProportionExtendSort
```

The routine Partition divides $U_i$ into two subsequences and returns their boundary. The first subsequence consists of all $A[i] \leq A[mid]$, and the second subsequence
consists of all \( A[j] \geq A[mid] \). Partition is described as follows.

\[
\text{Partition}(mid, U_i)
\begin{align*}
&\{ \text{Suppose } U_i = A[u1..u2] \} \\
&i := u1, j := u2 \\
&\text{while } i \leq j \text{ do begin} \\
&\quad \text{while } i \leq j \text{ and } A[i] \leq A[mid] \text{ do } i := i + 1 \\
&\quad \text{while } i < j \text{ and } A[j] \geq A[mid] \text{ do } j := j - 1 \\
&\quad \text{if } i \geq j \text{ then return } i \\
&\quad \text{Swap}(i, j) \\
&\quad i := i + 1, j := j - 1 \\
&\text{end while} \\
&\text{return } i \\
\end{align*}
\]

The routine Partition first searches from left to right (by increasing \( i \)) until the element \( > A[mid] \) is found, then searches from right to left (by decreasing \( j \)) until the element \( < A[mid] \) is found. When both searches have paused, if \( i < j \), then we exchange \( A[i] \) and \( A[j] \) and resume the process, otherwise, the algorithm terminates and returns \( i \).

\( p \) in the algorithm is a fixed positive constant. Depending on \( p \), we obtain various algorithms with different complexities.

Different algorithms can be obtained from this algorithm by replacing line (2.1). QUICKSORT is obtained by \( s2 := uo2 \), and BINARY INSERTIONSORT by \( s2 := s2 + 1 \). That is, QUICKSORT and BINARY INSERTIONSORT are two extreme cases of this algorithm.

ProportionExtendSort requires space for a pushdown stack which stores the arguments of pending recursive calls. We infer easily that maximal stack depth can be kept to \( 2\log n \) if we make a slight modification of ProportionExtendSort: always solve the smaller subproblem first, i.e., replace lines (2.3) and (2.4) with

\[
\begin{align*}
\text{if } |UL_i| < |UR_i| \\
&\text{then ProportionExtendSort}(SL_i, UL_i) \\
&\text{ProportionExtendSort}(SR_i, UR_i) \\
\text{else ProportionExtendSort}(SR_i, UR_i) \\
&\text{ProportionExtendSort}(SL_i, UL_i)
\end{align*}
\]

3. Analysis and simulation results.

**Theorem 3.1.** Let \( W(n) \) denote the worst case number of comparisons required by the algorithm for sorting \( n \) elements, then

\[
W(n) \leq 1/(1 + 1/(2p^2 + 2p - 1))n \log n, \text{ for } p \geq 1.
\]

**Proof.** In PROPORTION EXTEND SORT, comparisons occur only in the routine Partition. Partition(median of \( S_i, U_i \)) takes exactly \( |U_i| \) comparisons to split \( U_i \), where \( S_i \) and \( U_i \) denote the sorted set and the unsorted set upon entering the Partition for the \( i \)th time, respectively.
Let \( q \) denote the total number of the call to Partition, then we have

\[
W(n) = \sum_{i=1}^{q} |U_i| = \sum_{i=1}^{n} \sum_{j=1}^{n} b(A[j] \in U_i) = \sum_{i=1}^{n} \sum_{j=1}^{q} b(A[j] \in U_i) \\
\leq n \times \max_{j=1}^{n} \left( \sum_{i=1}^{q} b(A[j] \in U_i) \right),
\]

where \( b(A[j] \in U_i) \) is one when \( A[j] \) is in \( U_i \), and zero otherwise.

Suppose \( \max_{j=1}^{n-1}(\sum_{i=1}^{n} b(A[j] \in U_i)) = \sum_{i=1}^{n} b(y \in U_i) = k \). For the sake of simplicity, it will be assumed that \( y \) is in \( U_i \) for \( 1 \leq i \leq k \), and is not in \( U_i \) for \( i > k \).

By lines (2.1) in the algorithm, we have

\[
|U_i| = \begin{cases} |UA_i| & \text{when } (p+1)p|S_i| > |S_i| + |UA_i|, \\ p|S_i| & \text{otherwise}. \end{cases}
\]

\((p+1)p|S_i| > |S_i| + |UA_i|\) implies

\[
|UA_i| < (p^2 + p - 1)|S_i| \quad \text{for } p \geq 1.
\]

By (3.1) and (3.2), we can imply

\[
|U_i| \leq (p^2 + p - 1)|S_i| \quad \text{for } p \geq 1.
\]

Thus, we have

\[
|U_i| + (p^2 + p - 1)|U_i| \leq (p^2 + p - 1)(|S_i| + |U_i|) \quad \text{for } p \geq 1.
\]

That is,

\[
|U_i| \leq \frac{(p^2 + p - 1)(|S_i| + |U_i|)}{p^2 + p} \quad \text{for } p \geq 1.
\]

By the relation of \( UA_{i+1} \) and \( U_{i+1} \), we have

\[
|U_{i+1}| \leq |UA_{i+1}|.
\]

From lines (2.3) and (2.4) in the algorithm, we have

\[
S_{i+1} = SL_i \quad \text{and} \quad UA_{i+1} = UL_i, \quad \text{or} \\
S_{i+1} = SR_i \quad \text{and} \quad UA_{i+1} = UR_i.
\]

By the relation of \( UL_i, UR_i, \) and \( U_i \), clearly

\[
|UL_i| \leq |U_i| \quad \text{and} \quad |UR_i| \leq |U_i|.
\]

Since \( SL_i \) and \( SR_i \) are the left and right half of \( S_i \), we infer easily

\[
|SL_i| \leq \frac{|S_i|}{2} \quad \text{and} \quad |SR_i| \leq \frac{|S_i|}{2}.
\]
By (3.4)–(3.7) and (3.3), we obtain
\[
|S_{i+1}| + |U_{i+1}| \leq |S_{i+1}| + |UA_{i+1}|
\leq \max(|SL_i| + |UL_i|, |SR_i| + |UR_i|)
\leq \frac{|S_i|}{2} + |U_i|
\leq \frac{|S_i|}{2} + \frac{|U_i|}{2}
\leq \frac{|S_k|}{2} + \frac{|U_k|}{2} + \frac{(p^2 + p - 1)}{2(p^2 + p)}(|S_k| + |U_k|)
\leq \frac{(2p^2 + 2p - 1)}{2(p^2 + p)}(|S_k| + |U_k|).
\]
(3.8)

Clearly, we have
(3.9) \quad |S_i| + |U_i| \leq n \quad \text{and} \quad 2 \leq |S_k| + |U_k|.

by (3.8) and (3.9), we have
\[
2 \leq |S_k| + |U_k|
\leq \frac{(2p^2 + 2p - 1)}{2(p^2 + p)}(|S_{k-1}| + |U_{k-1}|)
\leq \left(\frac{(2p^2 + 2p - 1)}{2(p^2 + p)}\right)^{k-1}(|S_1| + |U_1|)
\leq \left(\frac{(2p^2 + 2p - 1)}{2(p^2 + p)}\right)^{k-1} \times n \quad \text{for} \quad p \geq 1.
\]

We solve this inequality to obtain
\[
k \leq \frac{\log n}{\log(1 + 1/(2p^2 + 2p - 1))}.
\]
Hence,
\[
W(n) \leq k \times n \leq \frac{n \log n}{\log(1 + 1/(2p^2 + 2p - 1))} \quad \text{for} \quad p \geq 1.
\]

The worst case behavior of PROPORTION EXTEND SORT seems to occur on the completely sorted sequence as QUICKSORT. Even so, it takes only \(O(n \log n)\) comparisons, and by empirical results, the actual number of comparisons required is far fewer than the formula given in the above theorem.

Since the “ripple swap”[8] technique can reduce the overall number of moves, the experimental results given below for PROPORTION EXTEND SORT are obtained by replacing line (2.2) with “ripple swap.”

Table 3.1 summarizes experimental results on the average case performance we observed. For each input size \(n\), we provided 20 distinct sets, each of which consists of randomly generated distinct values. The columns Compar and Moves reflect the average number of compares and exchanges for each instance, respectively. Columns EXSORT\((p = 2)\) and EXSORT\((p = 6)\) are the results of PROPORTION EXTEND SORT with \(p = 2\) and \(p = 6\), respectively. Columns SPSORT and QSORT are
the results reported for PROPORTION SPLIT SORT with \( p = 16 \) and CLEVER QUICKSORT, respectively, from [8].

Among the four algorithms given in Table 3.1, the average number of compares observed for EXSORT\((p = 2)\) is the smallest, and is less than the values of \( n \log n - n \) consistently. SPSORT\((p = 16)\) is one of the best results of PROPORTION SPLIT SORT. Nevertheless, the average number of compares observed for EXSORT\((p = 6)\) is lower than that observed for SPSORT\((p = 16)\): furthermore, when \( n \geq 1000 \), EXSORT\((p = 6)\) used fewer moves than SPSORT\((p = 16)\). For Table 3.1, exclusive of the number of compares with \( n = 50 \) and \( n = 100 \), data observed for EXSORT\((p = 6)\) are all lower than those observed for QSORT.

To compare PROPORTION EXTEND SORT with PROPORTION SPLIT SORT in further details, we present empirical results of these two sorting algorithms for some \( p \) in Tables 3.2 and 3.3, respectively. For each \( p \), we employed 20 distinct sets, each of which consists of randomly generated distinct values as the simulation above. In Tables 3.2 and 3.3, Columns COMPARES and MOVES are the average number of comparisons and exchanges required to sort 50000 elements, respectively. Table 3.2 shows the results reported from [8].

As can be seen from Tables 3.2 and 3.3, the performance of PROPORTION EXTEND SORT is better than the performance of PROPORTION SPLIT SORT, since in the case of the same number of comparisons, in general, PROPORTION EXTEND SORT uses fewer moves than PROPORTION SPLIT SORT. Like PROPORTION SPLIT SORT, one can choose also the best version of PROPORTION EXTEND SORT by comparing the practical overall cost of comparisons and movements for each \( p \) based on the property of the input sequence given.

Table 3.4 shows the average execution time required by the algorithms running on Pentium II\(350\). All the algorithms are written in C and run under the MS-DOS operating system. Both QsortN and QsortR are the best-of-three versions of Quicksort, but when selecting three values to decide a pivot, QsortR adopts a random strategy while QsortN does not. SPSORT\((p = 24)\) and EXSORT\((p = 16)\) are the fastest.

---

**Table 3.1**
The average number of compares and moves.

<table>
<thead>
<tr>
<th>Size</th>
<th>EXSORT((p = 2))</th>
<th>EXSORT((p = 6))</th>
<th>SPSORT</th>
<th>QSORT</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Compares</td>
<td>Moves</td>
<td>Compares</td>
<td>Moves</td>
</tr>
<tr>
<td>10</td>
<td>23</td>
<td>9</td>
<td>22</td>
<td>9</td>
</tr>
<tr>
<td>50</td>
<td>227</td>
<td>89</td>
<td>241</td>
<td>68</td>
</tr>
<tr>
<td>100</td>
<td>551</td>
<td>316</td>
<td>592</td>
<td>157</td>
</tr>
<tr>
<td>500</td>
<td>3895</td>
<td>2536</td>
<td>4154</td>
<td>1102</td>
</tr>
<tr>
<td>1000</td>
<td>8815</td>
<td>5925</td>
<td>9416</td>
<td>2152</td>
</tr>
<tr>
<td>5000</td>
<td>55585</td>
<td>25065</td>
<td>58835</td>
<td>15432</td>
</tr>
<tr>
<td>10000</td>
<td>121845</td>
<td>45971</td>
<td>127064</td>
<td>33536</td>
</tr>
<tr>
<td>50000</td>
<td>727599</td>
<td>314952</td>
<td>756937</td>
<td>185026</td>
</tr>
</tbody>
</table>

---

**Table 3.2**
The average-case results of PROPORTION SPLIT SORT when \( n = 50000 \).

<table>
<thead>
<tr>
<th>( p )</th>
<th>COMPARES</th>
<th>MOVES</th>
</tr>
</thead>
<tbody>
<tr>
<td>3/2</td>
<td>717527</td>
<td>598808</td>
</tr>
<tr>
<td>7/4</td>
<td>719662</td>
<td>744052</td>
</tr>
<tr>
<td>2</td>
<td>720121</td>
<td>619890</td>
</tr>
<tr>
<td>3</td>
<td>725147</td>
<td>413725</td>
</tr>
<tr>
<td>4</td>
<td>730372</td>
<td>346538</td>
</tr>
</tbody>
</table>

---

**Table 3.3**
The average number of compares and moves.

<table>
<thead>
<tr>
<th>( p )</th>
<th>COMPARES</th>
<th>MOVES</th>
</tr>
</thead>
<tbody>
<tr>
<td>3/2</td>
<td>717527</td>
<td>598808</td>
</tr>
<tr>
<td>7/4</td>
<td>719662</td>
<td>744052</td>
</tr>
<tr>
<td>2</td>
<td>720121</td>
<td>619890</td>
</tr>
<tr>
<td>3</td>
<td>725147</td>
<td>413725</td>
</tr>
<tr>
<td>4</td>
<td>730372</td>
<td>346538</td>
</tr>
</tbody>
</table>
versions of PROPORTION SPLIT SORT and PROPORTION EXTEND SORT, respectively, for sorting integers. From the empirical results given in Table 3.4, our algorithm is the fastest.

4. Comments. PROPORTION EXTEND SORT is a simple, competitive, and efficient sorting algorithm that sorts \( n \) elements, using \( O(\log n) \) extra space and making \( O(n \log n) \) comparisons in the worst case. Since an average case theoretic analysis seems to be quite sophisticated, it is left as an open problem. However, in our simulation, this sorting algorithm is better than PROPORTION SPLIT SORT, and used fewer data moves and fewer comparisons, and its number of comparisons is close to \( \log(n!) \). Therefore, we believe that it is possible to replace CLEVER QUICKSORT by PROPORTION EXTEND SORT.

REFERENCES