Fast Exchange Sort

Hon Wai Leong
National University of Singapore
150 PUBLICATIONS 2,092 CITATIONS
SEE PROFILE

FAST EXCHANGE SORTS*

Nachum Dershowitz
Department of Computer Science
University of Illinois
Urbana, IL 61801, USA

Hon–Wai Leong
Department of Computer Science
National University of Singapore
Singapore 0511

ABSTRACT

We present three variations of the following new sorting theme: Throughout the sort, the array is maintained in piles of sorted elements. At each step, the piles are split into two parts, so that the elements of the left piles are smaller than (or equal to) the elements of the right piles. Then, the two parts are each sorted, recursively. The theme, then, is a combination of Hoare's Quicksort idea, and the Pick algorithm, by Blum, et al., for linear selection. The variations arise from the possible choices of splitting method.

Two variations attempt to minimize the average number of comparisons. The better of these has an average performance of $1.075n\lg n$ *comparisons. The third variation sacrifices the average case for a worst-case performance of* $1.756n\lg n$, *which is better than Heapsort. They all require minimal extra space and about as many data moves as comparisons.*

1. Introduction

The sorting problem is: Given an array $a_1, a_2, \ldots, a_n$ of elements, rearrange them so that $a_1 \le a_2 \le \cdots \le a_n$, where $\le$ is a given linear ordering of elements. An *exchange sort* [Knut73] is one that goes about this task by repeatedly looking for a pair of elements a_i and a_j $(1 \le i < j \le n)$ that are inverted $(a_i > a_j)$ and exchanging them. By combining ideas from Hoare's Quicksort algorithm [Hoar61, Hoar62] and Blum, Floyd, Pratt, Rivest, and Tarjan's Pick algorithm [BlFP73] for linear selection, we have come up with a new scheme for exchange sorts.

One the one hand, we were interested in the possibility of improving the average case of Quicksort, and, on the other hand, in improving the worst case of Heapsort. The model under consideration is a comparison-based model; the *cost* of a method is measured by the number of

*This research was supported in part by the National Science Foundation under Grant DCR 85–13417.

comparisons required for sorting n elements. It is well-known [Knut73] that under this comparison-based model, sorting has cost $\Omega(n \lg n)$. Furthermore, there are well-known methods that are optimal to within a multiplicative constant. Heapsort [Will64] has both average- and worst-case performances of $O(n \lg n)$; asymptotically, its worst case is $2n \lg n$. Quicksort [Hoar61] has average-case performance of $1.39 n \lg n$ [Knut73]. This was improved by Singleton [Sing69] to $1.19 n \lg n$ using the median-of-three method, and can be made arbitrarily close to $n \lg n$ by increasing the sample size. However, Quicksort suffers from quadratic worst-case performance. In both Heapsort and Quicksort, the number of data-moves is the same order of magnitude as the number of comparisons. Binary-Insertion Sort [Knut73] is also optimal for this model, since it has worst-case performance of $O(n \lg n)$ comparisons; however, it requires $O(n^2)$ data-moves.

Briefly, our scheme is as follows: The array is maintained in small piles of sorted elements throughout the sorting process. Thus, the first step is to preprocess the input array into piles of sorted elements. This step is done only once; from then on, the piled structure is preserved. At each subsequent step, a particular pivot element is chosen and the piles are split into two parts so that all the elements in the left piles are smaller than all the elements in the right piles. Then, the two parts are each sorted, recursively. The theme, then, which we will call **Pilesort**, combines splitting around a pivot, as in Quicksort, with piling to find a pivot, as in Pick.

The variations we consider arise from different possibilities for splitting. Two variations attempt to minimize the average number of comparisons (over all possible input array permutations); another sacrifices average-case performance for an $O(n \lg n)$ worst-case that out-performs Heapsort [Will64, Floy64]. They all require some extra space and about as many data moves as comparisons; their practical value is limited to cases where comparisons are relatively expensive.

In the next section, we elaborate on the general scheme. Section 3 analyzes its time complexity. Three variations of the scheme are considered in Sections 4 and 5; experimental results are given in Section 6. We conclude with a brief discussion.

2. The Theme

The first step is to preprocess the input array, creating piles of three elements, each pile sorted. The result is shown in Figure 1, assuming (for simplicity) that n is a multiple of 3. This step costs at most n comparisons; on the average, it costs $8n/9$. From this point on, the piled structure will be maintained by the algorithm.

To sort a piled array, a *pivot-pile* is first selected, and its middle is used as the *pivot element*. Then the piles themselves are rearranged so that the middle elements of the left-piles are less than (or equal to) the pivot element, while the middle elements of the right-piles are greater than or equal to the pivot element. This is similar to Hoare's Partition except that key comparisons are based on the pile middles and that entire piles are being moved around. Different ways of choosing the pivot lead to different algorithms, as will be described in subsequent sections. We shall refer to this rearrangement as *Pile_Partition*. After *Pile_Partition*, the situation is as depicted in Figure 2.

At this point, note that elements in $W \cup Z$ are on the proper side of the partition, while elements in $X \cup Y$ may be greater or smaller than the pivot element. To complete the partitioning process, two things need to be done:

(a) The elements in $X \cup Y$ must be compared with the pivot element to determine if they are in the proper partition. This step takes $n/3$ comparisons.

(b) Restoring the pile structure.

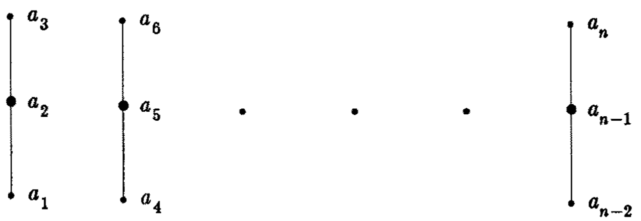

Figure 1. After INITIAL_PILING

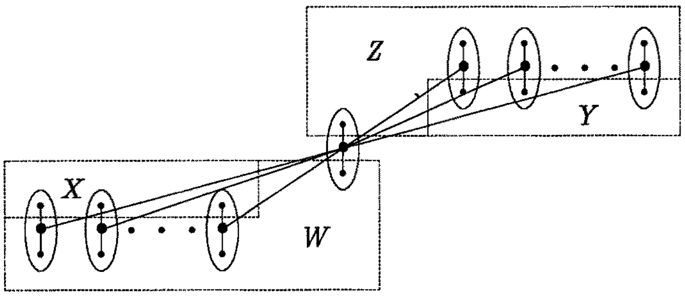

Figure 2. Configuration after PILE_PARTITION

In order to efficiently restore the pile structure, we note the following:

(i) A *loose* element can be merged with a pile of two *sorted* elements in 2 comparisons at most, and 5/3 comparisons on the average.

(ii) Three loose elements can be repiled in 3 comparisons at most, and 8/3 comparisons on the average.

(iii) Three piles of two elements can be merged into two piles of three elements by first comparing two maximal elements to form one pile of three, and then comparing the remaining loose element with the smaller, and, if necessary, the larger element of the remaining pile of two. This takes 3 comparisons at most, and 232/90 comparisons on the average (the average is over all possible arrangements of three piles of two elements).

The following procedures sketch the whole sorting scheme:

```
procedure Sort;
    begin
        Initial_Piling (1,N);
        Pilesort (1,N)
    end;

procedure Pilesort (L,R);
    begin
        if (R-L) ≤ Threshold
            then Smallsort (L,R)
            else begin
                     Pile_Partition (L,R,T);
                     Partition_Cleanup (L,R,T);
                     Pilesort (L,T-1);
                     Pilesort (T+1,R)
                 end
    end;
```

As in Quicksort [Knut73], a *Smallsort* routine is used whenever the array segment is so small that the overhead does not justify recursive calls. The *Partition_Cleanup* procedure does tasks (a) and (b) simultaneously. It is similar to the Partition phase of Quicksort and is applied to the X and Y elements. The elements in X are scanned until an element that is greater than the pivot (i.e. in the wrong partition) is found; then the Y–elements are scanned until one smaller than the pivot is encountered. These elements are broken off from their respective piles, interchanged, and inserted into the remaining piles of two. By (i) above, this takes at most 2 comparison per element moved and 5/3 comparisons on the average. This process is continued until all the elements of *either* X or Y are exhausted. Without loss of generality, assume that the elements in Y are exhausted. At this point, the remaining elements in X are scanned until three elements are found out of place. These form a pile that is moved to the right partition and the remaining remaining three piles of two are repiled into two piles of three. This will take, in the worst-case, 2 comparisons per element moved, and $(8/3+232/90)/3 = 472/270$ comparisons per element moved, on the average.

To summarize, *Partition_Cleanup* takes n comparisons in the worst case, $n/3$ to decide whether the element is in the proper partition and $2n/3$ for the actual moves. The average case, however, depends on how many elements get moved and on how they are moved.

Finally, once the piles are completely partitioned, the algorithm recursively sorts each of the two parts.

3. Analysis

Let cm be the cost to partition m piles in the *Pile_Partition* procedure. We assume that the pivot pile falls in the interval $[xm:(1-x)m]$ where $0 \le x \le 0.5$. Note that x is the split of the piles, not of the elements themselves. Let α be the fraction of the elements in $X \cup Y$ that are interchanged in the *Partition_Cleanup* phase, and let A be the cost per element for such a move. Let β be the fraction of $X \cup Y$ that are moved from the larger partition to the smaller partition, and B, the cost per element for such a move. Thus, α is in the range $0 \le \alpha \le 2x$ and β is in the range $-(x-\alpha/2) \le \beta \le (1-x)-\alpha/2$ (β is negative if the elements are moved in the opposite direction). Then the cost for one pass is the sum of the following:

(i) $cn/3$ to select the pivot-pile and in the process, partition the *piles* themselves;

(ii) $n/3$ comparisons to compare the elements in $X \cup Y$ without repiling;

(iii) $A\alpha n/3$ for interchanging elements in X and Y; and

(iv) $B|\beta|n/3$ for moving the elements from one side to the other.

Since the resultant split of the elements (not just the piles) is $x+\beta/3$, the cost $T(n)$ for sorting n elements satisfies the following recurrence:

$$T(n) = \frac{(c+1)n}{3} + \frac{(A\alpha+B|\beta|)n}{3} + T\left(\frac{(3x+\beta)n}{3}\right) + T\left(\frac{(3-3x-\beta)n}{3}\right) \tag{3.1}$$

A recurrence of the form $T(n) = kn + T(yn) + T((1-y)n)$ has a solution $T(n) = \gamma n \lg n$ where

$$\gamma = \frac{-k}{y\lg y + (1-y)\lg(1-y)}.$$

Therefore, the solution to recurrence (3.1) is given by

$$\gamma = \frac{-\dfrac{(c+1+A\alpha+B|\beta|)}{3}}{\left(\dfrac{3x+\beta}{3}\right)\lg\left(\dfrac{3x+\beta}{3}\right)+\left(\dfrac{3-3x-\beta}{3}\right)\lg\left(\dfrac{3-3x-\beta}{3}\right)} \tag{3.2}$$

It follows that the performance of the algorithm depends on two factors: the repiling cost $(A\alpha+B|\beta|)$ which depends on α and β, and the final split $(3x+\beta)/3$ which depends only on β. We note that the worst (maximum) repiling cost occurs when all the undetermined elements change sides in which case $\alpha=2x$, $\beta=1-2x$, but in this case the final split is $(x+1)/3$. On the other hand, the worst final split occurs when all the undetermined elements in the smaller part move to the larger part, namely $\alpha=0$, $\beta=-x$ and the split is $2x/3$. Thus, we have

$$T(n) = \begin{cases} \left(\dfrac{c+1+2Ax+B(1-2x)}{3}\right)n + T\left(\dfrac{(x+1)n}{3}\right) + T\left(\dfrac{(2-x)n}{3}\right) \\[2ex] \left(\dfrac{c+1+Bx}{3}\right)n + T\left(\dfrac{2xn}{3}\right) + T\left(\dfrac{(3-2x)n}{3}\right) \end{cases} \tag{3.3}$$

More generally, with piles of height $2h+1$, the worst case recurrences are

$$T(n) = \begin{cases} \left(\dfrac{c+h+2hAx+Bh(1-2x)}{2h+1}\right)n + T\left(\dfrac{(x+h)n}{2h+1}\right) + T\left(\dfrac{(h+1-x)n}{2h+1}\right) \\[2ex] \left(\dfrac{c+h+Bhx}{2h+1}\right)n + T\left(\dfrac{(h+1)xn}{2h+1}\right) + T\left(\dfrac{(2h+1-(h+1)x)n}{2h+1}\right) \end{cases} \tag{3.4}$$

But it turns out that $h = 1$ (giving piles of height 3) is usually optimal. Though taller piles reduce the cost of pivot selection, they cost more to build and maintain, and make $X \cup Y$ proportionally larger.

4. Linear Selection

In this section, we present the first variation of our basic sorting scheme. We describe an (unimplemented) algorithm with worst–case performance of $1.756n\lg n$ comparisons, which is better Heapsort.

We first note that if, in the *Pile_Partition* procedure, the median of the pile middles (i.e. $x=0.5$) can be found in linear time in the worst–case, then the procedure to divide the piles would take linear time in the worst–case. Furthermore, at least one third of the elements would be in each of the two parts. This guarantees that the resultant sorting algorithm is $O(n\lg n)$ in the worst–case. In fact, any linear time selection procedure that partitions the m piles so that the worst–case split is proportional to m will guarantee an $O(n\lg n)$ worst–case sorting method.

The fastest linear median selection algorithm is that of Schonhage, Paterson, and Pippenger [ScPP76] (improved in [Eber79]; see also [Knut73]). It has a $3k$ asymptotic upper bound on the number of comparisons to find the median of k elements. Thus, for worst–case performance, we have $x=0.5$, $c=3.0$, $A=B=2$, and by (3.3) the worst case of this sorting algorithm is then given by $\gamma = \max(2,\ 1.815)$. In other words, $T(n)=2n\lg n$ in the worst–case, as for Heapsort. The average case should be considerably lower.

To improve on the worst case, we need an approximation of the median that costs less to compute that still guarantees $O(n\lg n)$ worst–case running time. This may be accomplished by piling the middle elements themselves. First, the middle elements are grouped into piles of five elements (five turns out to be bettr than 3, as we will see below) and, then, the *median of their medians* is used as the pivot element. To be more precise, we expand *Pile_Partition* as follows:

P1: Pile the middle elements into piles of fives, so that the center element is their median.

P2: Use the median find algorithm to partition the piles of center elements.

P3: Move undetermined piles into the correct partition with respect to the median of medians. At the end of Step P2, we have the Hasse diagram shown in Figure 3 where each node in the diagram is a pile of three elements. Step P3, then moves the piles in $Q \cup R$ into the proper partition.

Step P1 costs $6(n/15)=2n/5$, since it takes at most 6 comparisons to find the middle of five elements [Knut73]. Step P2 costs $3n/15=n/5$, using the $3m$ worst–case median find algorithm. Lastly, Step P3 costs just $2n/15$ comparisons. Therefore, *Pile_Partition* costs $11n/15$ comparisons. Also, this method guarantees a worst–case split of the piles at $x=3/10$ which occurs when all the undetermined piles in Q moves to the other side. Thus, the corresponding recurrence is given by

$$T(n) = \begin{cases} \dfrac{11n}{15} + \dfrac{n}{3} + \dfrac{2n}{3} + T\left(\dfrac{13n}{30}\right) + T\left(\dfrac{17n}{30}\right) \\ \\ \dfrac{11n}{15} + \dfrac{n}{3} + \dfrac{n}{5} + T\left(\dfrac{n}{5}\right) + T\left(\dfrac{4n}{5}\right) \end{cases}$$

giving $\gamma=\max(1.756,1.755)$. Accordingly, the worst case is $1.756n\lg n$, which is better than Heapsort's worst–case performance of $2n\lg n$.

If we replace the $3m$ median find algorithm of [ScPP76] by the slower $5.4305m$ Pick algorithm [BlFP73], the worst case of the sort will be $\max(1.920,1.979)n\lg n = 1.979n\lg n$ which is just under $2n\lg n$.

We can also consider grouping the *middle* elements into piles of three, instead of five. Then it costs $3n/9$ to pile the middles into groups of three, $3n/9$ to find median of middles, and $n/9$ to move undetermined piles, and the worst–case split of the piles is $x=1/3$ and the recurrence is

$$T(n) = \begin{cases} \frac{7n}{9} + \frac{n}{3} + \frac{2n}{3} + T\left(\frac{4n}{9}\right) + T\left(\frac{5n}{9}\right) \\ \\ \frac{7n}{9} + \frac{n}{3} + \frac{2n}{9} + T\left(\frac{2n}{9}\right) + T\left(\frac{7n}{9}\right) \end{cases}$$

and γ=max(1.794,1.745). Thus, the resultant sort has worst–case performance of $1.794n\lg n$, which is still better than Heapsort.

5. Fast Selection

In order to improve the average case of Pilesort, we look for a median finding algorithm that has faster average–case performance. The best of these methods has an average case of $1.075n\lg n$, which is better than $1.19n\lg n$ of Singleton's median–of–three Quicksort.

The idea here is to always split the piles evenly into two parts using a fast median selection algorithm for *Pile_Partition*. We use the fastest known method, Select, by Floyd and Rivest [FlRi75a, FlRi75b] as modified by Brown [Brow76] which requires $1.5m$ comparisons on the average, to find the median of m elements. This variation of the sorting method is called Select–Pilesort.

The average case can be partially analyzed as follows: The worst–case split of the piles is $x = 0.5$. Substituting the average values for each of c, A and B, (viz. $c = 1.5$, $A = 5/3$, and $B = 472/270$) into (3.3), we get γ=max(1.389,1.223). This analysis, however, assumes the worst distribution of undetermined elements. In order to get a better estimate, we have run extensive experiments to measure the empirical values of α and β. The results suggest that α is approximately 0.75 and β is small enough to be negligible. Substituting these into (3.2), we get the value $\gamma = 1.25$.

This variation of Pilesort can be significantly improved by eliminating the top–level recursive calls to Select from *Pile_Partition*. This is based on the observation that Select first uses the median of a *random sample* to partition the piles. If the sample median is indeed the exact median, Select terminates; otherwise, it calls itself recursively to find the exact median within the appropriate part. In most cases, however, there is already a fairly good partitioning of the piles and so there is little point in partitioning the piles further to get the exact median.

To further reduce the cost of *Pile_Partition*, we consider another variation, called *Random–Pilesort*. In our implementation, the middle pile is always chosen. To complete the procedure, the piles are partitioned about this pivot element. Thus, the cost of partitioning the piles is only $n/3$. However, we have no guarantee as to the worst–case split of the piles and so the analysis in Section 3 cannot be applied. Since the entire array is always partitioned around the median of three elements, it is natural to compare this sort with Singleton's variation of Quicksort, which requires an average cost of $1.19n\lg n$ comparisons. Empirical results (see the next section) show that, on the average, γ is approximately 1.105 for Random–Pilesort.

To push this method further, one can use the median of three pile–middles as the pivot element. In our experiments, we uses the first, middle and last piles. This version of Random–Pilesort was found to perform slightly better.

6. Experimental Results

We implemented and tested the algorithms described in Section 5 in order to evaluate their average–case performances. Select–Pilesort uses the Select algorithm for choosing the pivot; Fast–Pilesort is the improved version using Select; Random–Pilesort uses Partition; Random–Pilesort–3 is the variation using three piles. For comparison, we also implemented Quicksort [Hoar61] and Singleton's median–of–three variation [Sing69], which we will call Quicksort–3. The implementations are all in Pascal and were tested on a CDC Cyber 175. A summary of the implemented algorithms is given in Figure 4.

Quicksort 1.338 ± 0.014	Hoare's Quicksort (partition about the middle element)
Quicksort–3 1.144 ± 0.008	Singleton's Quicksort (partition about median of first, middle, and last elements)
Select–Pilesort 1.415 ± 0.006	Fast median–finding variation of new scheme (uses Select to partition piles about median of pile middles)
Fast–Pilesort 1.093 ± 0.003	Improvement of Select–Pilesort (partition about the median of a sample)
Random–Pilesort 1.105 ± 0.006	Random partition variation of new sorting scheme (uses middle element of center pile as pivot)
Random–Pilesort–3 1.075 ± 0.003	Refinement of Random–Pilesort (uses median of middles of first, last, and center piles as pivot element)

Figure 4. Summary of the various sorts.

The data used to test the programs were randomly generated real numbers uniformly distributed over [10000.0 , 100000.0). This large range was chosen to minimize any chance of duplication of data values. For each sort, we ran the program on various data–sets of sizes $n =$ 1024, 2048, 4096, 8192. For each size, we ran the program on sixteen different sets of random data. The mean values for γ obtained in these experiments are shown in the table below, along with their standard deviations. We use the formula $\bar{\gamma} \pm \frac{t_{s-1,0.025} \cdot \sigma}{\sqrt{s}}$ to obtain a *95%*

confidence interval for the values of γ where $\bar{\gamma}$ is the mean of γ, s is the sample size, σ is the standard deviation, and t is Student's distribution. These intervals are also given in Figure 4.

As is to be expected, the values of γ for Quicksort (1.338 ± 0.014) and Quicksort–3 (1.144 ± 0.008) are quite close to their theoretical values of 1.386 and 1.19, respectively. For Select–Pilesort, the overall average value of γ is 1.415 ± 0.006. The discrepancy between this and the value 1.25, obtained in the previous section, is due to the fact that the actual cost of Select over the range of values $m \leq 8192/3$ was approximately $2m$, which is higher than the asymptotic average cost of $1.5m$ used in the previous section. With the more realistic value, the value of γ should be 1.417, in close agreement with the experimental results.

For Fast–Pilesort, where top–level recursive calls to Select are omitted, the value of γ drops dramatically to 1.093 ± 0.003, which is better than the performance of Quicksort–3. The reason for the dramatic improvement is that after Select has partitioned the array about the median of a random sample, the split of the piles is reasonably close to 0.5. Insisting on finding the exact median, as in Select–Pilesort, requires an additional ck, where k is the size of the larger partition, which—with high probability— is about $n/2$. The effect of this variation is to choose as pivot element the median of a random sample of the elements, which explains why this algorithm performs better than Quicksort–3.

In Random–Pilesort the pivot element is just the middle of a random pile. Thus, Random–Pilesort effectively partitions the elements about the median of three elements. We found that Random–Pilesort has $\gamma = 1.105 \pm 0.006$, which is slightly better than Quicksort–3.

	1024	2048	4096	8192	Overall
Quicksort	1.321 .050	1.354 .058	1.360 .051	1.317 .048	1.338 .054
Quicksort–3	1.138 .040	1.151 .030	1.150 .031	1.138 .020	1.144 .031
Select–Pilesort	1.406 .031	1.416 .018	1.417 .020	1.422 .015	1.415 .022
Fast–Pilesort	1.087 .017	1.092 .012	1.098 .007	1.096 .005	1.093 .012
Random–Pilesort	1.087 .017	1.112 .032	1.104 .020	1.117 .014	1.105 .024
Random–Pilesort–3	1.067 .015	1.074 .010	1.080 .009	1.080 .006	1.075 .011

Table 1. Means and standard deviations of γ.

In Random–Pilesort–3, the median of three pile middles is chosen as the pivot element, and so the pivot element is, in effect, the median of between 7 and 9 elements. As expected, the corresponding value of γ for Random–Pilesort–3 is 1.075 ± 0.003, better than that of all the other sorting methods we tried.

Note that Quicksort and Quicksort–3 have average cost $\gamma n \lg n$, while all the new algorithms require an additional $8n/9$ preprocessing cost, giving a total average cost of $8n/9 + \gamma n \lg n$. Finally, we note that the new sorting methods showed consistent performance: the value of γ did not change very much either for different data sets, nor for different values of n. The standard deviations for all the new methods were consistently less than 0.04 and are smaller than those for Quicksort and Quicksort–3. As is to be expected, the variances are smallest for Random–Pilesort and Select–Pilesort, which partition about the median of a small subset of the elements.

7. Conclusion

We have studied the performance of a new sorting scheme, called Pilesort, under the comparison–based model where the cost function is the number of comparisons needed to sort. The scheme combines ideas from Hoare's Partition algorithm and the linear time selection algorithm by Blum *et al.* It is robust in the sense that it can be implemented to obtain good worst–case or average–case performances.

Several variations of the scheme were studied. The first variation, using linear selection, was aimed at improving on the worst case of Heapsort, and is of theoretical interest. A simple implementation has a worst case of $2n \lg n$, the same as Heapsort. An improved version gave a worst–case performance of $1.756 n \lg n$ which is better than Heapsort.

The other variations sacrifice $O(n \lg n)$ worst case for a faster average case. A simple implementation, using a fast median–find algorithm, was found (empirically) to have $1.417 n \lg n$ average–case performance, which is comparable to that of Hoare's Quicksort. With a small change, this was improved to $1.093 n \lg n$. Using a random partition, the average case was found to be $1.105 n \lg n$, which is slightly better than that of Singleton's Quicksort. By selecting the median of three piles, rather than a pile at random, the performance improved to $1.075 n \lg n$.

The number of data–moves required by the new methods tends to be between two to three times that of Quicksort because of the need to moves piles of elements rather than one element at a time. Therefore, the new algorithms could be practical only in situations where key comparison is more expensive than data movement.

References

[BlFP73] Blum, M., R. W. Floyd, V. Pratt, R. L. Rivest, and R. E. Tarjan, "Time Bounds for Selection," *Journal of Computer and System Sciences,* (1973), Vol. 7, pp. 448–461.

[Brow76] Brown, T., "Remarks on Algorithm 489," *Trans. on Mathematical Software,* (1976), Vol. 3, No. 2, pp. 301–304.

[Floy64] Floyd, R. W., "Algorithm 245 (Treesort3)," *Comm. of ACM,* (1964), Vol. 7, No. 12, p. 701.

[FlRi75a] Floyd, R. W., and R. L. Rivest, "Expected Time Bounds for Selection," *Comm. of ACM,* (1975), Vol. 18, No. 3, pp. 165–172.

[FlRi75b] Floyd, R. W., and R. L. Rivest, "Algorithm 489 (The Algorithm SELECT — for Finding the i–th Smallest of n Elements)," *Comm. of ACM,* (1975), Vol. 18, No. 3, p. 173.

[FrMc70] Frazer, W. D. and A. C. McKellar, "Samplesort: A Sampling Approach to Minimal Tree Sorting," *Journal of ACM,* (1970), Vol. 17, No. 3, pp. 496–507.

[Hoar61] Hoare, C. A. R., "Algorithm 63 (Partition); 64 (Quicksort); 65 (Find)," *Comm. of ACM,* (1961), Vol. 4, No. 7, pp. 321–322.

[Hoar62] Hoare, C. A. R., "Quicksort," *Computer Journal,* (1962), Vol. 5, No. 1, pp. 10–15.

[Knut73] Knuth, D. E., *The Art of Computer Programming, Vol. 3 (Sorting and Searching),* (1973), Addison–Wesley, Reading, MA.

[ScPP76] Schonhage, A., M. S. Paterson, N. Pippenger, "Finding the Median," *Journal of Computer and System Sciences,* (1976), Vol. 13, pp. 184–199.

[Sing69] Singleton, R. C., "Algorithm 347 (An Efficient Algorithm for Sorting with Minimal Storage)," *Comm. of ACM,* (1969), Vol. 12, No. 3, pp. 185–187.

[Will64] Williams, J. W. J., "Algorithm 232 (Heapsort)," *Comm. of ACM,* (1964), Vol. 7, No. 6, pp. 347–348.