M Pivot Sort – Replacing Quick Sort

James Edmondson
Computer Science Department
Middle Tennessee State University
Murfreesboro, TN 37132

1.0 Abstract

M Pivot Sort (or Pivot Sort) is a general purpose comparison based sorting algorithm based on Quick Sort. This new internal sorting algorithm borrows ideas from the Central Limit Theorem from Probability and Statistics (namely the selection of an adequate sample size to form pivot points in the population) and Heap structures. Pivot Sort was developed to be a better alternative to Quick Sort and Heap Sort as the general purpose sorting algorithm used by professionals by insuring equal or faster speed than Quick Sort while being as reliable (as in, you can trust it not to go $O(n^2)$ on your data) as Heap Sort and Merge Sort.

2.0 Keywords

sorting partitioning efficient algorithm contiguous array

3.0 Introduction

The field of general purpose comparison based sorting algorithms has seen very few new algorithm submittals in the past forty plus years. Quick Sort is commonly referred to as the fastest known average case comparison based sorting algorithm, but the algorithm has a tendency towards $O(n^2)$ on a variety of data inputs, which will be highlighted later. Merge Sort has excellent lower and upper bound performance, but uses additional memory for a copy of the array, and because of its recursion, can use large quantities of memory. Finally, Heap Sort is used as an insurance policy if a guarantee is needed against $O(n^2)$ behavior.

No general purpose comparison based sorting algorithm exists that doesn't have some kind of drawback. Quick Sort has been shown to be the fastest average case internal sorting algorithm, but Heap Sort has a better worst case. Quick Sort has been shown to have many $O(n^2)$ inputs as has been noted by several industry professionals like Jon Bentley in "The trouble with qsort" in 1992. The real problem with Quick Sort is not that these inputs could happen remotely, but that these inputs are extremely common in databases. In Bentley's publication, he notes that inputs in a form similar to a Bell Curve will produce $O(n^2)$ behavior in the most common versions of Quick Sort (Bentley 2). Anyone familiar with the field of Probability and Statistics should perk up to that kind of information, since normal distributions occur frequently and naturally in data sets. Also, Quick Sort does poorly on duplicate lists (Sedgewick.) A general purpose sorting algorithm should handle general cases such as Bell Curves and large numbers of duplicates.

M Pivot Sort was developed to address the shortcomings of Quick Sort while maintaining competitive performance on random lists. The end result is an algorithm that performs sorting in place and can be trusted as an insurance policy while performing less total operations, like data comparisons and movements, than Quick Sort in both average and worst case performances.

4.0 Methodology

The algorithm for M Pivot Sort and its local functions are as follows (where M is the number of pivots and the algorithm is expressed in pseudocode derived from Cormen et al's process, (Cormen 15-21.))

```
PIVOTSORT(A,first,last)
1.  create array P [0 .. M-1]
2.  if first < last and first >= 0
3.      then if first < last – 13
4.          then CHOOSEPIVOTS(A,first,last,P)
5.              INSERTIONSORT(A,P[0]-1,last)
6.              nextStart ← first
7.              for I ← 0 to M-1
8.                  do curPivot ← P[i]
9.                      nextGreater ← nextStart
10.                     nextGreater ← PARTITION(A,nextStart,nextGreater,curPivot)
11.                     exchange A[nextGreater] ↔ A[curPivot]
12.                     exchange A[nextGreater+1] ↔ A[curPivot+1]
13.                     if nextStart == first and P[i] > nextStart+1
14.                         then PIVOTSORT(A,nextStart,P[i]-1)
15.                     if nextStart != first and P[i] > P[i-1]+2
16.                         then PIVOTSORT(A,P[i-1]+1,P[i]+1)
17.                     nextStart ← nextGreater + 2
18.             if last > P[M-1]+1
19.                 then PIVOTSORT(A, P[M-1]+1,last)
20.         else INSERTIONSORT(A,first,last)
```

```
CHOOSEPIVOTS(A,first,last,P)
1.  size ← last–first+1
2.  segments ← M+1
3.  candidate ← size / segments – 1
4.  if candidate >= 2
5.      then next ← candidate + 1
6.      else next ← 2
7.  candidate ← candidate + first
8.  for i  ← 0 to M-1
9.      do P[i] ← candidate
10.         candidate ←candidate + next
11. for i ← M-1 to 0
12.     do exchange A[P[i]+1] ↔ A[last]
13.         last ← last-1
14.         exchange A[P[i]] ↔ A[last]
15.         last ← last-1
```

```
PARTITION(A,nextStart,nextGreater,curPivot)
1.  for curUnknown ← nextStart to curPivot-1
2.      do if A[curUnknown] < A[curPivot]
3.          exchange A[curUnknown] ↔ A[nextGreater]
4.          nextGreater ← nextGreater + 1
5.  return nextGreater
```

In a simpler bulleted format:

1.) Choose a number of pivot candidates.
2.) Sort this small list of pivot candidates with an algorithm that performs well on small lists.
3.) Select pivots from the small sorted list.

4.) Partition the rest of the list around selected pivots from the pivot candidates.
5.) Repeat steps 1-4 for each segment of the list between the pivots.

Empirical tests have shown that M Pivot Sort works most effectively on 3 to 6 pivots (5 pivots performing best) without the optimizations outlined later. So, during the first step, M Pivot Sort (hereafter also referred to simply as "the algorithm" or "Pivot Sort"), chooses 6 to 12 pivot candidates from strategic locations in the list (refer to the next diagram for a visual.) In a known random list, the algorithm should not have to strategically select candidates. However, to make a general purpose sorting algorithm that must deal with anything from ascending and descending lists to Bell Curves, the algorithm must intelligently select pivot candidates from the list. The algorithm for pivot selection showed here simply selects from strategic locations. However, choosing pivot candidates could be handled with random selection as well.

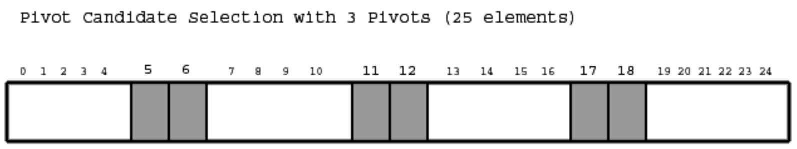


Figure 1

M Pivot Sort then needs to sort the small list of pivot candidates to intelligently select pivots. In an optimal pivot selection, there should be a value known to be less than the pivot and greater than the pivot. By moving the pivot candidates to either the front or the back of the list, and sorting these candidates plus one element from the part of the list we moved the candidates to, the algorithm can select a pivot every two elements starting with the second element to guarantee the pivot relationship mentioned earlier. Selecting pivots from the pivot pool is a passive step and does not require any real work to accomplish.

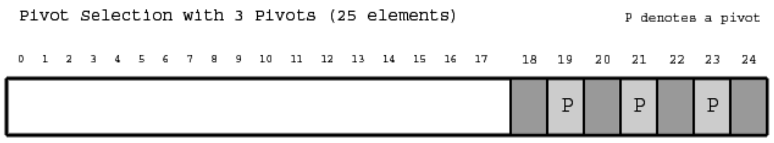


Figure 2

After the pivots have been selected, the time comes to partition the list around the M number of pivots dividing the list into M + 1 segments. For each of the M + 1 segments, the algorithm must be called on that segment, unless the segment is less than 14 elements in number, in which an algorithm like Insertion Sort is guaranteed to have better performance on such small lists (Kruse 307.)

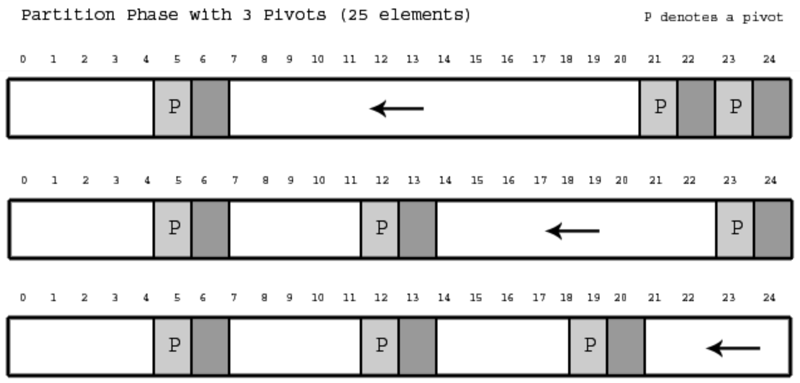
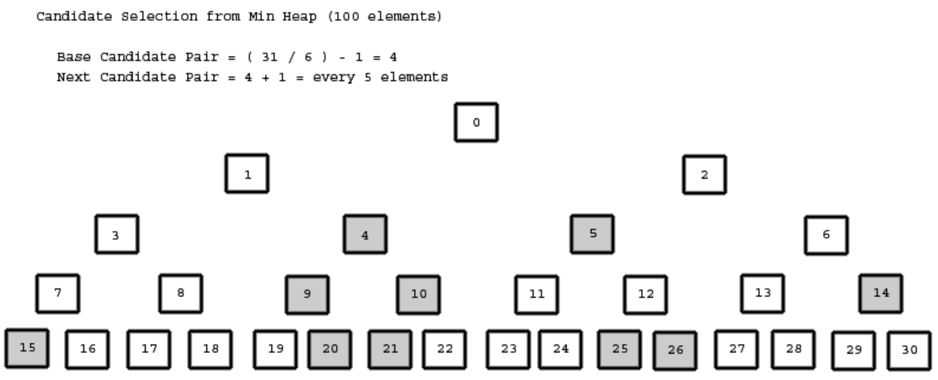


Figure 3

Three additional steps may be included in the M Pivot Sort outline to virtually eliminate any tendency toward n^2 behavior and speed up the algorithm on most list inputs. These steps can be summarized in the following:

1.) Prep the list against $O(n^2)$ behavior.
2.) If duplicate pivots were selected, partition all equal entries to the left of the duplicate pivot.
3.) If necessary, select a different number of pivots for the next level of M Pivot Sort.

M Pivot Sort works optimally when the pivots' final locations have significant distances between each other. Hypothetically, this situation exists when the array is arranged in a Minheap before the work of M Pivot Sort is begun. This step alone should all but guarantee O(n log n) performance from M Pivot Sort and is recommended in any implementation. A single call to a Build MinHeap function adds only linear time and consequently adds no real overhead, and doing so will actually improve the performance of the sort that follows. M Pivot Sort handles random lists extremely well, but also works well on patterns (especially ascending ones) and this step takes some of the randomness out of the list. The following diagram shows pivot candidate selection from a Minheap of 31 elements. As the list gets larger, the candidates become even farther apart.


Candidate Selection from Min Heap (100 elements)
Base Candidate Pair = (31 / 6) - 1 = 4
Next Candidate Pair = 4 + 1 = every 5 elements
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
Figure 4

The second optimization step allows the algorithm to deal with duplicate values, and to minimize the number of required passes through the list, the algorithm only checks for duplicates if pivots are found to be duplicates (i.e. equal), which means that a pivot, the candidate after it, and another pivot are equal. Implementing this optimization costs only M - 1 comparisons per segment per level, which is negligible and when implemented produces a function that can be trusted on any database.

The third optimization requires changing the number of pivots during runtime if a situation occurs where all of the pivots appear at the front or end of the list. This final optimization adds randomization to the arsenal of tools available to M Pivot Sort to counteract O(n^2) situations.

5.0 Preliminary Analysis

This analysis of the algorithm will concentrate on the 5 pivot version of M Pivot Sort to establish a reference point, and the lists are assumed to be greater than 14 elements since otherwise Insert Sort or a smaller version of M Pivot Sort would be used. An in-depth analysis including all of the optimizations for this algorithm will be completed in a future document. The only optimization included in this analysis is the one for duplicates.

The best case for the M Pivot Sort algorithm is O(n), and this happens when the list is completely full of duplicates (ie in the case of a list of integers where every element equals 1.) The derived function for this case is rather easy to see. Swapping the pivot candidates to the end of the list would cost 30 moves (10 swaps * 3 moves per swap.) The internal Insertion Sort on the small list will use 11 – 1 = 10 comparisons and no moves. The algorithm will search through the entire array for smaller values, adding n – 10 comparisons, and 6 moves to place the pivot and its successor at the front of the array. After finding the next pivot to be a duplicate of the first, the algorithm will search the array for duplicates adding an additional n – 10 comparisons and using 6 moves to replace the pivot's location with itself (this can be stopped by not allowing swaps of the same address in the swap function.) Afterwards, the other 3 pivots and their successors will add 3 comparisons and try to swap positions with themselves, adding 18 moves. So, the derived function for the best case of M Pivot Sort with duplicates is 2(n-10) + 10 + 4, or 2n -6 comparisons and 60 moves.

In the case that there are no duplicates, the best case for M Pivot Sort might appear to be dividing the list into 6 nearly equal pieces, but hypothetically another situation also exists. M Pivot Sort should also perform optimally in the unusual situation where the first pivot value's final location is at the ½ mark, the second pivot value is the value that should end up at the ¾ mark, the third pivot at the 7/8 mark, the fourth at the 15/16 mark, and the fifth at the 31/32 mark. This kind of pattern in the pivots will result in M Pivot Sort performing O(n log n) comparisons and O(n log n) moves.

For a derivation of the worst case for M Pivot Sort with no optimizations (other than the optimization for duplicates,) the pivot candidates would be selected from the list in a descending order (which Insertion Sort wouldn't handle well) and add 55 comparisons and 55 moves during sort, for a total so far of 55 comparisons and 30 + 55 (85) moves, then n -10 comparisons would be made 5 times (once for each pivot) and the pivots would be moved to the front of the array, adding another 30 moves, plus 4 comparisons for the duplicate optimization. So for sorting only ten elements, we have made 55 + 5(n-10) + 4 or (5n + 9) comparisons and 55 + 30 + 30 or (115) moves. Compare this kind of situation to Quick Sort's worst case on a list of 100 values. Quick Sort (already derived at n * (n – 1) / 2 comparisons in worst case) would perform 99 + 98 + 97 + 96 + 95 … + 1 compares until completion (without counting the two additional comparisons needed for a median-of-3 version of Quick Sort). Pivot Sort would perform 90 + 90 + 90 + 90 + 90 + 80 + 80 + 80 + 80 +80 … + 10 + 10 + 10 + 10 + 10 + 55 * (n/10 – 1) + 45 comparisons. Basically, the latter conservatively derives to n * (n-1) / 4 + 55 * (n/10 – 1) + 45 comparisons and 55 * (n/10 – 1) + 60 * (n/10 – 1) + 45 moves (the latter simplifying to 115(n/10 – 1) + 45 or 11.5n – 70 moves) for this comparison-based worst case. The worst case for moves happens anytime a list pivots and partitions optimally, resulting in O(n log n) moves.

The most compelling argument for this algorithm is not just these preliminary findings, but the application of Probability and Statistics to this algorithm. To quote from McClave et al, "The Central Limit Theorem assures us that, for large samples, the test statistic will be approximately normally distributed regardless of the shape of the underlying probability distribution of the population" (McClave 287.) In short, even if we don't get the sample we're looking for the first time (i.e. the sample of the population that would have given us optimal pivot points,) we will get the desired sample eventually, regardless of the distribution of the data. M Pivot Sort is even more cautious than this statement because it goes through the trouble of trying to avoid outliers in initial input by selecting pivot candidates from strategic locations in the contiguous array list. Also, the algorithm can even change the sample size and the relative locations of pivot candidates at runtime. Added together, this culminates in an algorithm that is theoretically and statistically sound.

6.0 Testing

Tests were done on Heap Sort, Merge Sort, Pivot Sort (M Pivot Sort), and Quick Sort on random integer arrays from 10 to 100,000,000 and three different kinds of random lists were used: lists generated with rand (a range of 0-32,000 possible values), brand (a range of 0-200,000 possible values) and a unique random number generator written by Andy Thomas, the generator being based on ideals found in the book "Random Number Generators: good ones are hard to find" by Stephen Park and Keith Miller (refer to Thomas' reference.) The former list types were used to show the inadequacies of Quick Sort on lists of duplicates. The Quick Sort algorithm uses the conservative median-of-3 method.

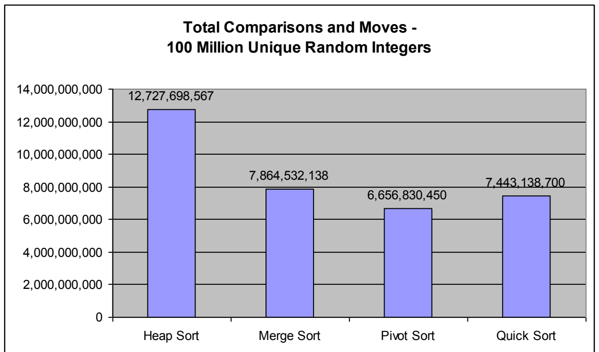


Figure 5

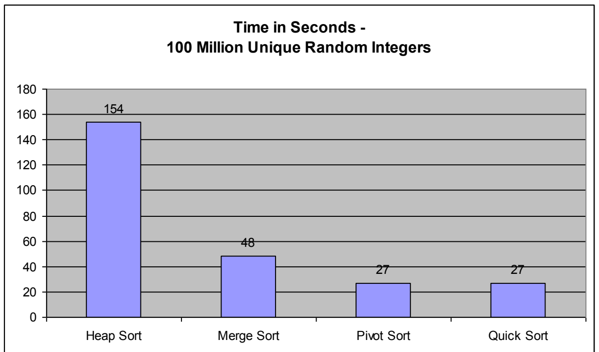


Figure 6

From these initial results on unique random lists, Pivot Sort proves itself as a contender for comparison based sorting speed king. The results of using a random function producing a range of possible values of 0-200,000 yielded almost identical results for Heap Sort and Merge Sort, but Pivot Sort and Quick Sort went in quite opposite directions.

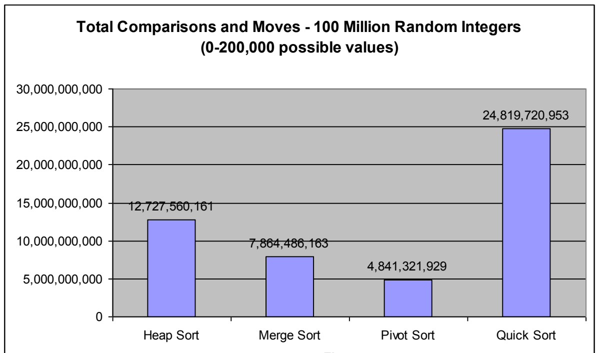


Figure 7

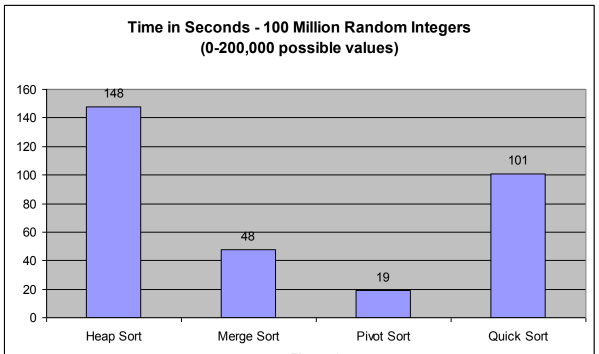


Figure 8

When the random list is generated from 200,000 possible integer values, Quick Sort begins to show O(n^2) behavior. As can be seen in the two charts above, Quick Sort's total comparisons and moves triples and requires nearly 5 times the speed needed for a list of unique random integers.

With a random list of values in the range of 0-32,000, Quick Sort quickly ballooned to over 154,319,219,511 total comparisons and moves and required 641 seconds to complete on a Quad Xeon PC with 2 GBs of RAM. Pivot Sort required only 3,984,499,415 total comparisons and moves and took only 15 seconds to finish.

Here is a list of total operations on a typical run for each array type and size. The columns are the array sizes, and the rows are the sorting algorithms tested.

Table 1: Sorting Algorithm Test - Total Operations (using unique randoms)

	10	100	1,000	10,000	100,000	1,000,000	10,000,000	100,000,000
Heap Sort	121	2,824	44,319	608,234	7,745,140	93,941,520	1,106,164,906	12,727,698,567
Pivot Sort	72	1,703	26,266	325,509	4,198,842	51,923,875	586,920,732	6,656,830,450
Merge Sort	92	1,886	28,666	387,705	4,874,228	58,577,672	686,550,089	7,864,532,138
Quick Sort	84	1,648	26,379	350,423	4,612,376	54,946,781	654,065,963	7,443,138,700

Table 2: Sorting Algorithm Test - Total Operations (using brand : 0 - 200,000 possible values)

	10	100	1,000	10,000	100,000	1,000,000	10,000,000	100,000,000
Heap Sort	130	2,782	43,931	607,454	7,744,038	93,938,697	1,106,157,427	12,727,560,161
Pivot Sort	61	1,649	25,069	340,394	4,046,825	48,444,763	496,400,893	4,841,321,929
Merge Sort	91	1,883	28,695	387,705	4,873,995	58,576,722	686,544,241	7,864,486,163
Quick Sort	71	1,741	25,074	385,445	4,380,130	56,625,530	758,649,951	24,819,720,953

Table 3: Sorting Algorithm Test - Total Operations (using rand : 0 - 32,000 possible values)

	10	100	1,000	10,000	100,000	1,000,000	10,000,000	100,000,000
Heap Sort	128	2,746	44,118	607,779	7,744,189	93,938,898	1,106,153,447	12,727,656,478
Pivot Sort	66	1,856	24,771	331,823	4,029,825	43,056,145	403,798,279	3,984,499,415
Merge Sort	91	1,885	28,655	387,644	4,873,961	58,576,788	686,544,987	7,864,470,474
Quick Sort	79	1,687	28,120	356,654	4,552,228	63,370,617	1,999,897,844	157,630,949,592

After running over 200,000 tests, several observations can be made from the compiled data. First, Pivot Sort will always perform more comparisons than Merge Sort, but for contiguous lists, Merge Sort will perform many more move operations than Pivot Sort. Second, Pivot Sort should perform more comparisons than Quick Sort on unique lists (like social security numbers or logins), but Pivot Sort performs less moves than Quick Sort. Third, although trustworthy and reliable, Heap Sort's performance doesn't match any other algorithm's performance once the array size extends beyond cache (most sorting tests focus on no more than 10,000 integers, which at 40Kbs fits easily in any modern CPU's fast cache memory.)

7.0 Conclusion

M Pivot Sort performed extremely well during empirical tests, meeting or beating the best times and total operations (data comparisons and moves) of the major established general purpose comparison based sorting algorithms. Although further testing and analysis will need to be done, the algorithm shows amazing promise in a field of Computer Science long thought dry of innovation.

One aspect of M Pivot Sort ought to receive especial notice: number of moves. As the size of an array increases, M Pivot Sort pulls away from the competition and should be an ideal sorting solution for arrays of classes or strings, cases that generally feature more work being done on a move than a comparison. However, even with integers, the tests show that such an advantage in number of moves can have a tremendous impact. Proof of this can be seen in the time taken by M Pivot Sort vs. Merge Sort. Pivot Sort offers programmers an internal sort that maintains Quick Sort speed with Heap Sort reliability in a new era of sorting that sorely needs those features.

8.0 Acknowledgements

I would like to thank Dr. Cen Li and Dr. Judy Hankins of the MTSU Computer Science Department for their guidance and expertise in bringing this algorithm and research forward. I would also like to thank Dr. Untch for presenting me with invaluable research materials and guidance. Finally, thanks to my family, friends, and loved ones for sticking with me and tolerating my banter about algorithms, logarithms, and databases over the holidays and beyond, and special thanks to Gwen for keeping me sane and well fed through my frequent late night research sessions.

9.0 References

[1] Bentley, Jon L. "The trouble with qsort." *UNIX Review.* 10.2 (1992): 85-90.

[2] Bentley, Jon L. "Software Exploratorium: history of a Heapsort." *UNIX Review.* 10.8 (1992): 71-77.

[3] Cormen, Thomas, et al. *Introduction to Algorithms.* 2nd ed. Massachusetts: MIT Press, 2001.

[4] Kruse, Robert, et al. *Data Structures & Program Design in C –.* 2nd ed. New Jersey: Prentice Hall, 1997. 335.

[5] McClave, James T. and Terry Sincich. *A First Course in Statistics – Annotated Instructor's Edition.* 8th ed. New Jersey: Prentice Hall, 2003. 224, 277-308.

[6] Sedgewick, Robert and Jon Bentley. "Quicksort is Optimal." Knuthfest, Stanford University, Stanford. January 2002.

[7] Thomas, Andy. "Randomal64 Pseudo Random Number Generator." Generation5.org. 20 April 2001. 6 Feb. 2005 <http://www.generation5.org/content/2001/ga01.asp>.