American Journal of Engineering Research (AJER)
e-ISSN: 2320-0847 p-ISSN : 2320-0936
Volume-6, Issue-1, pp-32-36
www.ajer.org
Research Paper Open Access

Quick Sort with Optimal Worst Case Running Time

Dr. Mirza Abdulla
College of Computer Science, AMA International University

ABSTRACT: *Quick sort is more than 50 years old and yet is one of the most practical sorting techniques used in the computing industry. Its average running time matches the best asymptotic running time for a sorting algorithm and with a constant factor that outperforms most of the known sorting techniques. However, it suffers from the theoretical worst case time bound of $O(n^2)$ on a list of size n. Ironically, this worst case occurs when the list is already sorted in ascending or descending order! We present ways to improve the worst case time performance to asymptotically match the optimal worst case running time for any comparison based sorting techniques. Our technique, however, tries not to affect the average running time but the slightest.*

Keywords: *Quick sort, Sorting, order statistics, analysis of algorithms.*

I. INTRODUCTION

Sorting and searching are probably the most important problems in computing. Indeed most if not all of the problems in computing require the solution to some kind of search problem. The importance of sorting, in particular stems from the fact that it aids in the permutation or the rearrangement of data to facilitate faster search.

If we follow the history of sorting we see that some techniques for the rearrangement of data existed even before the first contemporary computer was built in the mid of the twentieth century. For example the Hollerith sort was used in the in the early years of the twentieth century during the population census in the USA. However, the construction of computers made the number of sorting techniques increase from few techniques to hundreds and thousands different techniques. These various sorting techniques can be divided into mainly two types: comparison based and non-comparison based.

Non-comparison based sorting techniques rely mainly on the random access ability of memory. Some of the most well known non-comparison based sorting techniques are Count sort, Radix and Bucket sort. On the other hand comparison based techniques rely for the most part on the comparison between elements to obtain the desired rearrangement of data. Most well known comparison based sorting techniques can be divided into incremental techniques and divide and conquer techniques. Examples of the incremental sorting techniques are bubble sort, Selection sort and Shell sort. These are known to be grossly inefficient techniques as they require $O(n^2)$ steps to sort a list of n elements in the worst and average case. However, variation of these techniques exists that attain sort times that are far better than the $O(n^2)$ steps of the original techniques. For example, Shell sort [1] which is a variant of the Insertion sort can attain times of the order of $O(n\ \log^2 n)$. Similarly, Selection sort [2] can be made to attain times of the order of $O(n\ \log^2 n)$, or even better as in the case of heap sort [3] which is a variant of selection sort and has a worst case time of $O(n\ \log n)$ which is optimal.

Examples of divide and conquer sorting techniques include ones such as merge sort, quick sort and heap sort. All these techniques attain the optimal time of $O(n\ \log n)$ in the average case. Similarly, in the worst case, both the merge sort, and heap sort attain the optimal bound of $O(n\ \log n)$ comparison steps to sort a list of n elments. However, quicksort may require $O(n^2)$ steps in the worst case, and strangely enough this case occurs when the data is already sorted! Yet quick sort was and is still the practical sorting technique in the average. Indeed many computer application vendors used quicksort as the main sorting technique, since the average running time of quick sort on a list of n elements may require about $1.9nlgn$ comparisons which is superior to those attained by other famous sorting techniques. The low constant factor in the average time complexity of quicksort contributed to making it one of the most important sorting algorithms. The algorithm is centered on a partitioning scheme that splits an input list into two: those with values not more than the pivot and those with values more than the pivot. It follows, that the choice of the pivot is of crucial importance to the running time of the algorithm. Indeed, if we (say) always choose the first item in a list as the pivot, then the running time of the quick sort technique degrades to $O(n^2)$ steps when fed with a list that is already sorted, since we would always get two lists after the partition: one that contains the pivot only if all the number are different, and the other contains the rest of the list.

Quicksort was originally proposed by Hoare [4] in the early sixties of the last century. The choice of the pivot was the first element in the given list. However, later the algorithm received wide spread attention and several ideas for the pivot were introduced. Ragab [5-6] discusses some of the choices for the pivot.

In this paper we tackle exactly this point of making quicksort run in $\theta(n \log n)$ time without losing its attractive feature of being highly practical. We do so by allowing quick sort perform the normal operations of partitioning and swapping without any modification, but only intervening in the choice of the pivot when the degenerate case of partitioning continues for some time without reverting to a more "acceptable" pivot for the partitioning of the list. Moreover, the intervention is thinned out to make sure that in the average case the partitioning step doesn't affect the overall time when it is not regularly called, yet making sure that we don't run into the $O(n^2)$ running time.

In section 2 we explain the working of the intervention method, the median of medians, and in section 3 we explain the quicksort technique and analyze its worst case complexity. Section 4 provides comparison of the enhanced quicksort to the classical one on various data sets.

II. THE MEDIAN OF MEDIANS

The famous median of median technique works by following the steps:

1. Partition the list of n elements into $\frac{n}{5}$ blocks of 5 elements each.
2. Find the median of each group.
3. Find the median of medians by recursion on the groups medians.
4. If the found median of medians is indeed the median of the whole list we can stop with answer
5. Otherwise partition the original list on this median and the larger partition would contain the median of the whole list and all we need to do is to find the item in that list whose rank would make it the median of the original list.

This technique was shown to run in $\theta(n)$ time in the worst case. However, the constant factor of the time complexity can be quite high to be practical in the average case during the pivot selection in quicksort. A faster implementation can be obtained if we relax the condition of finding the median to that of finding an acceptable pivot that splits the list into two lists, the smaller of which is not less than say quarter the size of the original list. For example if we accept the answer we get in step 3 of the above algorithm steps and ignore steps 4 and 5, we get a pivot which is guaranteed to split the list into two lists the smaller of which is at least one quarter the size of the original list. Such a relaxation would still guarantee the asymptotically optimal worst case running time of quicksort to be $O(n \log n)$.

If we relax the choice of the pivot to be one that when used to partition the list the size of the smaller of the two resulting lists is some constant fraction of the original list size we can reduce the constant factor in the asymptotic running time of the median of medians algorithm even further. Indeed, to guarantee the $O(n \log n)$ worst case time bound on quicksort, we may choose to call the medianOfMedians to work only on a small fraction of the given list. This work can guarantee that we get a pivot that would divide the list into two lists each not less than a particular fraction of the original list size, yet the extra work of calling this function would not substantially affect the running time, yet maintain the $O(n \log n)$ bound.

III. PIVOT SELECTION BY MEDIANOFMEDIANS

```
function acceptable Pivot(a[], l, r, k, j)
  i = l;
  while (i+jk ≤ r){
        m1 = i;
        t = i + jk;
        for(s= i+j; s<t; s+=j)      if (a[s] < a[m1]) m1 = s;      //takes O(n/j) time
        swap(i, m1);                                               //every item greater than k items
        i+=jk;
        }
  i = l;
  while (i+jk^2 ≤ r){
m1 = i;
        t = i + jk^2;
        for(s= i+jk; s<t; s+= jk)   if (a[s] > a[m1]) m1 = s;      //takes O(n/jk) time
        swap(i, m1);                                               //every item less than k items
        i+=k^2;
        }
  i = jk^2;
  return mom(a[l]; a[l+i]; a[l+2i]; . . . ; a[t])
}                                                                  // takes O(n/k^2) time.
```

Description of the algorithm

The first while loop iterates in jumps of *jk* where in each iteration the maximum of *k* items in this range each at distance of its neighbors, and the maximum element is placed in the first location in the range. At the end of the first while loop we are guaranteed that the first element in each block or range is not less than at least k other items. Similarly the second while loop iterates in jumps of jk^2 items and the minimum element of the maxima items found during the first while loop is placed in the first location of the block. This way we guarantee that the first item in each block of jk^2 items is the maximum of *k* elements in the block and the minimum of *k* elements in the block of jk^2 items.

IV. ANALYSIS OF THE ALGORITHM

The median of medians algorithm finds then the median of these first elements in each block. Thus we are guaranteed to get the median of n/jk^2 elements in the original list. The median obtained this way is thus $\geq \frac{n}{2jk^2}$ items and $\leq \frac{n}{2jk^2}$ items in the given list. It follows, therefore, that the returned median is greater than or equal at least $\frac{n}{2jk}$ item in the original list, and similarly it is less than or equal at least $\frac{n}{2jk}$ item in the original list. since both *j* and *k* are constants we are guaranteed to split the original list into two lists, smallest of which is a constant fraction of the original list.

The overall time of the algorithm is the sum of the time for each of the two while loops and the median of medians running time on the n/jk^2 items.

The first while loop inspects every j^{th} item in the list and the maximum of *k* items in a block of size *jk* is placed in the first item of the group. Thus there are at most $\frac{n}{jk}$ such blocks and the overall number of iterations is: $k\left(\frac{n}{jk}\right) = \frac{n}{j}$. It follows, therefore, that the overall time for the first loop is $\frac{c_1 n}{j}$ time units, for some constant $c_1 > 0$.

Similarly the second loop loop inspects every *jk*th item in the list and the minimum of *k* items in a block of size jk^2 is placed in the first item of the group. Thus there are at most $\frac{n}{jk^2}$ such blocks and the overall number of iterations is: $k\left(\frac{n}{jk^2}\right) = \frac{n}{jk}$. It follows, therefore, that the overall time for the first loop is $\frac{c_2 n}{jk}$ time units, for some constant $c_2 > 0$.

Finally the median of medians requires $c_3 n$ steps on a list of *n* items. Thus for a list of $\frac{n}{jk^2}$ items it would run in $\frac{c_3 n}{jk^2}$.

The overall time for our version of the median of medians requires $\frac{c_1 n}{j} + \frac{c_2 n}{jk} + \frac{c_3 n}{jk^2}$ time steps in the worst case to return a pivot that can split the original list into two lists, the smallest of which contains no less than $\frac{n}{2jk}$ items, where n is the number of elements in the original list. Thus we have:

Lemma 1. The acceptablePivot function makes $\frac{c_1 n}{j} + \frac{c_2 n}{jk} + \frac{c_3 n}{jk^2}$ steps to find a pivot which is guaranteed to produce partitions of size not less than $\frac{n}{2jk}$ items. ■

Corollary 2. The constants *j*, and *k* can be chosen so that the acceptablePivot function runs in time less than *n* in the worst case and yet produce partitions of acceptable size each. ■

V. THE QUICKSORT ALGORITHM

We use the conventional quicksort algorithm with the conventional pivot selection. However, we interfere when we see that the algorithm is going more and more into the degenerate case. The intervention forces the selection of a pivot that guarantees that the smallest of the partitioned lists is at least a constant fraction of the original list size. This way we can guarantee that we can get a worst case of $O(n \log n)$ time steps to sort a list of n elements. To make sure that the running time of the sorting algorithm is for the most part not affected by the "intervention", we only allow the intervention if we see signs indicative that the partitioning is doesn't yield for the most part acceptably size "balanced" partitions. In this case we use the median of medians algorithm on a small subset of the list to guarantee that we get a division of the list into more acceptable size partitions.

However, in order to know when to perform such an activity we need to have some kind memory to remember past history. This is accomplished through the use of a variable which is initially zero, but is incremented every time we see that the partitioning is not acceptable. When the count reaches a threshold we apply the median of medians or else the variable is reset to zero again. This way we don't intervene with the working of the sorting algorithm, unless there are strong indications that it is going to the degenerate case, and the intervention is applied in that case only to a small subset of the given list so as to reduce the time cost of the intervention.

```
HoareQS(a, l, r, d){
   if (r-l) < 25 return InsertionSort(a, l, r);
   else {
         if (d > maxDegenerate)  {v =acceptablePivot(a[], l, r, 5, 20);          swap(l, v); d =0;}
        x = a[l];
        i = l - 1
        j = r + 1
        while (i < i){
          repeat
            j = j - 1
          until a[j] ≤ x
          repeat
            i = i + 1
          until a[i] ≥ x
          if i < j
            swap( a[i], a[j] )
          else
            return j
        }
        swap(l, j);
         if ((r-l) > 10*(j-l)){
           HoareQS(a, l, j, 0);
   d++;
           HoareQS(a, i, r, d);
      }
      else if ((r-l) > 10*(r-i)){
           HoareQS(a, i, r, 0);
   d++;
           HoareQS(a, l, j, d);
      }
      Else {
           HoareQS(a, l, j, 0);
           HoareQS(a, i, r, 0);
  }
}
```

Explanation of algorithm steps

The quicksort algorithm checks to see of the size of the list is less than 20 in which case insertion sort is used. If the number of consecutive recursive calls in which one of the lists is unacceptably more than the size of the other we call the median Of Medians algorithm explained earlier to return the location of the pivot, which is then swapped with the first element in the list. The rest of the code is exactly what is found in classical quicksort where a conventional pivot finding technique such as the first element of the list. However, after the list is partitioned, we check to see if one of the partitions is at least 10 times the size of the other, in which case we increment the variable *d* before calling quicksort on the larger of the two. The variable *d* is used to count the number of successive recursive calls in which the size of one partition is "unacceptably" greater than the size of the other. However, it is reset to 0 when the disparity of the partition sizes is not "unacceptable". In the algorithm above the constant *max Degenerate* is used as the acceptability measure.

VI. ANALYSIS OF THE QUICKSORT ALGORITHM.

For a list of size *n* where *n* is a constant less than 25 the algorithm calls insertion sort which takes constant time on the given list. For lists of size more than 25 assume that the value of *d* has reached a threshold where we call the median of medians. In such a case the algorithm takes $\frac{c_1 n}{j} + \frac{c_2 n}{jk} + \frac{c_3 n}{j k^2}$ more time as was explained earlier. In order to find the worst case behavior of the algorithm we consider a completely ordered list, which would force the call to the acceptable Pivot function the maximum number of times. In this case the classical quicksort will keep calling itself recursively on a resulting partition list whose size can be as high as the original input list but one less item. However, with the new quicksort technique we continue only for an acceptable number of times before we revert to the median of medians which would give us a more acceptable partitioning of the list. Thus in the worst case, after *max Degenerate* consecutive recursive calls, where each call results in the reduction of the maximum partition size by one

we may have to call the acceptable Pivot function. In this case the size of the list cannot be more than $n - maxDegenerate$ items, where n is the size of the list during the previous call to the acceptable Pivot function in the recursion tree. Thus the acceptable Pivot function will divide the list into two lists the smallest of which is at least $i = \frac{(n - maxDegenerate\ )}{2jk}$ items in size.

Moreover the time for finding the pivot and then partitioning the list is $c_5 n = \frac{c_1 n}{j} + \frac{c_2 n}{jk} + \frac{c_3 n}{jk^2} + c_4 n$, where $c_4 n$ is the time to partition the list, for some constant $c_4 > 0$.

Thus we have the following recurrence for the worst case time complexity of the quicksort algorithm.

$T_n = T_i + T_{n-i} + c_5 n$ and $T_n = constant$ for $n < 25$.

Since i and n-i are both constant fractions of n, we have.

Theorem 3. The worst case running time of quick sort is $O(n \log n)$. ■

VII. CONCLUSIONS

In this paper we introduced and analyzed a quicksort variant that unlike the classical quicksort, attains optimal asymptotic time bound of $O(n \log n)$. The algorithm provides minimal intrusion to the working of the classical quicksort and thus the average should not deviate much from that of the classical quicksort algorithm. We carried our analysis on the single pivot partitioning technique, but the bounds still apply for the dual pivot partitioning technique.

REFERENCES.

[1]. Shell, D. L. (1959). "A High-Speed Sorting Procedure" (PDF). Communications of the ACM. 2 (7): 30–32.
[2]. Abdulla, Mirza (2016). "An efficient enhancement to selection sort". Submitted for publication.
[3]. Williams, J. W. J. (1964), "Algorithm 232 - Heapsort", Communications of the ACM, 7 (6): 347–348.
[4]. Hoare, C.A.R. (1962) Quicksort. The Computer Journal, 5, 10-15
[5]. Ragab, M. (2011) Partial Quicksort and Weighted Branching Processes. PhD Thesis, 28-35.
[6]. Ragab, M. and Rosler, U. (2014) The Quicksort Process. Stochastic Processes and their Applications, 124, 1036-1054.
[7]. Fill, J.A. and Janson, S. (2001) Approximating the Limiting Quicksort Distribution. Random Structures Algorithms, 19, 376-406
[8]. Fill, J.A. and Janson, S. (2004) The Number of Bit Comparisons Used by Quicksort: An Average-Case Analysis. ACM-SIAM Symposium on Discrete Algorithms., New York, 300-307.
[9]. R. Sedgewick, "Quicksort," PhD dissertation, Stanford University, Stanford, CA, May 1975. Stanford Computer Science Report STAN-CS-75-492.
[10]. R. Chaudhuri and A. C. Dempster, "A note on slowing Quicksort", SIGCSE Vol . 25, No . 2, June 1993.
[11]. Fuchs, M. (2013) A Note on the Quicksort Asymptotics. Random Structures and Algorithms.
[12]. Iliopoulos, V. (2013) The Quicksort Algorithm and Related Topics. PhD Thesis. Department of Mathematical Sciences, University of Essex.
[13]. Joseph JaJa, "A Perspective on Quicksort", Computing in Science & Engineering, vol. 2, no. , pp. 43-49, January/February 2000.
[14]. Knuth, The Art of Computer Programming, vol. 3, Addison-Wesley, 1975.
[15]. R. Loeser, "Some performance tests of :quicksort: and descendants," Comm. ACM 17, 3 , pp 143 – 152, Mar. 1974.
[16]. Wild, S., Nebel, M.E. and Mahmoud, M. (2014) Analysis of Quickselect Under Yaroslavskiy's Dual-Pivoting Algorithm. Algorithmica, 78, 485-506.