Modified Quick Sort: Worst Case Made Best Case

Omar Khan Durrani¹, Dr. Khalid Nazim S. A²

¹²Department of Computer Science & Engineering Vidya Vikas Institute of Engineering & Technology, Mysore

Abstract—Sufficient work has been carried out on analysis of quick sort by pioneers of computer Science and applied mathematics, which, no doubt is out of reach to describe and discuss. Also we cannot overlook any contribution made in the field of research and development, but still, facts come out with necessities. Having this in mind some improvements have been made with respect to the worst case performance of quick sort which is found in the academic reference material and contributions at a saturated state. In this paper quick sort is modified to perform Best when it is supposed to be worst. The results of modifications have yielded sufficient improvements over the existence one.

Keywords—sorting, quicksort, randomized, worst case, quicksort_wmb, partition, global variables, ordered input.

I. INTRODUCTION

Mathematicians have contributed algorithmic analysis from info-theoretic view point on the other side we algorithm engineers contribute from the angle of software and the computer architecture. A responsibility that showers is to cope for the upcoming challenges is to improve and provide a compatible code design keeping time efficiency as our objective. Sorting is one of the most important, well-studied and commonly applied problem in the field of computer technology.

Many sorting algorithms are known which offer various trade-offs in efficiency, simplicity, memory use, and other factors. However, these algorithms do not take much into account the features of compilers and computer architectures that significantly influence performance. Hence Quick sort algorithm analyzed and improved it in the worst case, though practitioners may have not shown much interest as they manage it with other sorting algorithms like the merge sort which perform much better for such cases, for which quick sort is proven to be slow. As an academian I feel this type of outcomes have to be shared among our community. It is also well known that Quick sort has been proven to be fastest on average case when compared to the other n log n class of algorithms like Merge sort and heap sort. Other sorting algorithms like bubble sort, selection, insertion sort, shell sort fall under n² class of sorting algorithm which has shown slow performance. Analysis and performance measurement of both the classes is experimented [7, 8].

II. QUICK SORT

QuickSort is an algorithm based on the Divide-and-Conquer paradigm that selects a pivot element and reordered the given list in such a way that all elements smaller to it are on one side and those bigger than it are on the other. Then the sub lists are recursively sorted until the whole list gets completely sorted. The selection of pivot could be the first element when the input is random numbers. In case of ordered input elements choosing pivot as first element will lead slow performance of algorithm resulting complexity as θ(n²) (i.e. worst case) instead of O (n log n) which is the complexity in best and average cases. To make quick sort perform O (n log n) with respect to time complexity when ordered input is considered practitioners have preferred random selection of the pivot leading to Randomized Quicksort. A complete theoretical analysis of Quick Sort is given in the following subsection. Also Quick sort is the default sorting scheme in some operating systems, such as UNIX.

III. ANALYSIS OF QUICKSORT

The total time taken to re-arrange the array as just described in the above section always takes O (n) or α n where α is some constant needed to execute in every partition. Let us suppose that the pivot we just chose has divided the array into two parts: one of size k and the other of size n − k. Notice that both these parts still need to be sorted. This gives us the following relation:

\[ T(n) = T(k) + T(n - k) + \alpha n \]  

Where \( T(n) \) refers to the time taken by the algorithm to sort \( n \) elements.
A. Worst case analysis

Consider the case, when pivot is the least element of the array (input array is in ascending order), so that we have \( k = 1 \) and \( n - k = n - 1 \) in equation (1). In such a case, we have:

\[
T(n) = T(1) + T(n-1) + \alpha n
\]

by solving the recurrence as follows:

\[
= T(n-i) + iT(1) + \alpha (\sum_{j=0}^{i-1}(n-j)) \quad \text{-------- (2)}
\]

Now clearly such a recurrence can only go on until \( i = n-1 \) (because otherwise \( n-1 \) would be less than 1). So, substitute \( i = n-1 \) in the equation (2), which gives us:

\[
T(n) = T(1) + (n-1)T(1) + \alpha \sum_{j=0}^{n-2}(n-j)
\]

\[
= nT(1) + \alpha (n(n-2) - (n-2)(n-1)/2) \quad \text{(Notice that } \sum_{j=0}^{n-2}j = \sum_{j=0}^{n-2}j = (n-2)(n-1)/2 \text{)}
\]

which is \( O(n^2) \).

This is the worst case of quick-sort, which happens when the pivot we picked turns out to be the least element of the array to be sorted, in every step (i.e. in every recursive call). A similar situation will also occur if the pivot happens to be the largest element of the array to be sorted.

B. Best case analysis

The best case of quick sort occurs when the pivot we pick happens to divide the array into two exactly equal parts, in every step. Thus we have \( k = n/2 \) and \( n-k = n/2 \) in equation (1) for the original array of size \( n \).

Consider, therefore, the recurrence:

\[
T(n) = 2T(n/2) + \alpha n \quad \text{------------------------- (3)}
\]

\[
= 2(n/4) + \alpha n \quad \text{(By simplifying and grouping terms together)}
\]

\[
= 22(2T(n/8) + \alpha n/4) + 2\alpha n
\]

\[
= 23T(n/8) + 3\alpha n
\]

\[
= 2kT(n/2k) + k\alpha n \text{{(Continuing likewise till the kth step)}}
\]

Notice that this recurrence will continue only until \( n = 2k \) (otherwise we have \( n/2k < 1 \), i.e. until \( k = \log n \). Thus, by putting \( k = \log n \), we have the following equation:

\[
T(n) = nT(1) + \alpha n \log n, \text{ which is } O(n \log n). \text{ This is the best case for quick sort.}
\]

It also turns out that in the average case (over all possible pivot configurations), quick sort has a time complexity of \( O(n \log n) \), the proof of which is beyond the scope.

C. Avoiding the worst case

Practical implementations of quick sort often pick a pivot randomly from the list each time [1, 2]. This greatly reduces the chance that the worst-case ever occurs. This method is seen to work excellently in practice but still much time is exploited by the randomizer [1]. The other technique, which deterministically prevents the worst case from ever occurring, is to find the median of the array to be sorted each time, and use that as the pivot. The median can be found in linear time but that is saddled with a huge constant factor overhead, rendering it suboptimal for practical implementations [3].

IV. LITERATURE SURVEY

Thomas H. Cormen et. al [3] have quoted that Quick sort, deteriorates and takes Quadratic time in the worst case, spends a lot of time even on the sorted or almost sorted data. It performs a lot of comparisons even on sorted data, but swap count is low for sorted or almost sorted input. Mark Allan Weiss in [11] has also stated that quick sort has \( O(n^2) \) worst case performance. Howrowitz et at in [1] have said that, a possible input on which quicksort displays worst case behavior is one in which the elements are already in order.

Almost all the authors of algorithm books and research papers on quick sort analysis have agreed that quicksort perform no better than \( O(n^3) \) in case of ordered input (considering randomized and medians as exceptional cases). With all the above survey and many others references made the theoreticians and practitioners have considered quick sort worst case classified under asymptotic class \( O(n^3) \).
void quick::quicksort_wmb(int low, int high)
{
    int j;
    if(low<high)
    {
        j=partition_wmb(low,high);
        if (no_part==1)
            if(Aorder)
                {cout<<"A order"<<endl; return;}
            else
                if(Dorder)
                    {cout<<"Dorder"<<endl; return;}
        quicksort_wmb(low,j-1);
        quicksort_wmb(j+1,high); } // end of if compound statement
    } // end of quicksort
}

int quick::partition_wmb(int low, int high)
{
    int key,i,j,temp;
    no_part++;
    key=a[low];
    i=low;
    j=high+1;
    while(i<=j)
    {
        do{i++;}while(key>=a[i]);
        do{j--;}while(key<a[j]);
        temp=a[i];a[i]=a[j];a[j]=temp;
    }
    temp=a[low];a[low]=a[j];a[j]=temp;
    if (no_part==1) if ((j==n-1) && (i==n)) Dorder =1;
    else if ((i==1) && (j==0)) Aorder =1;
    return j;
} //end of partition_wmb

Note:The array element at the position high+1 is assigned with a value maximum+1(1000 in our case) for the algorithm as it will stops index I at high+1.

Figure 1.1: Quicksort and Partition functions modified to perform as best case for ordered inputs.

V. QUICKSORT WORST CASE MADE BEST CASE

When bubble sort and selection sort can be modified to perform only n-1 comparison on sorted input data as is stated in most algorithm books and also realized by experiments conducted at laboratory, why not Quick sort can do this? This made me to sit aside for a short time and modify the existing code of quicksort and partition as quicksort_wmb and partition_wmb as shown in the following C++ code in figure 1.1. The bold statements in the code reflects the changes made over the existing one.

In the above code, three global variables int no_part, Aorder, Dorder are initialized to zero, further the decision to return from the quick sort is made when the respective values of global variables are set to 1 and recursion is avoided later on. The global variable no_part counts the number of partitions made in each execution of quicksort, Aorder is an indicator when the array is encountered to be in ascending order, similarly Dorder indicates if the array is in descending order.

Dorder is set to value 1 in the partition function when i=n and j=n-1, that is i is out of upper bound and j is at the upper bound of given array.

That i gets incremented to n searching for key >a[i] which is the highest in the array and j stands still at n-1 as a[n-1] i.e a[j] is < key as shown in figure 1.2.

Aorder is set to value 1 in the partition function when i=1 and j=0, that is i does not get incremented further as key>a[i] becomes false in the statement do{i++;}while(key>=a[i]);(do-while is executed only once), where as key<a[j] is true until j become 0 after n executions of the statement do{j--;}while(key<a[j]); (until key=a[j]). The above two cases are shown in figure 1.2(in case of ascending order input)and figure1.3(when descending order input is considered).

In case of descending order input, a small modification can be made in the algorithm which will avoid the last or the highest element getting swapped with the first (i.e the lowest) element and printing the array in reverse order which may take a time complexity of O(2n), i.e Θ(n) time for one execution of partition function and Θ(n) for reversing the input array.

Also the quicksort_wmb fails to sort the random input arrays which has the first element as the highest or the lowest among the array elements. In such cases one can always use int findmin() or int findmax() which will return respective minimum and maximum array elements using it in comparison with the first element of given input array one can overcome the such problem.This modification will take extra time of about O(n) along with the time of quicksort_wmb which is Θ(n)resulting the time complexity as Θ(n^2+n)=Θ(n^2) using the theorem1 given in chapter 2 of [2].

![Figure 1.1: Quicksort and Partition functions modified to perform as best case for ordered inputs.](image1)

![Figure 1.2: Ascending order input.](image2)

![Figure 1.3: Descending order input.](image3)
VI. EXPERIMENTAL RESULTS

The configuration of the test bed used for experiments is described as: Intel ®, core (™) 2E7200, 2.53 GHz, 0.99 GB of RAM, System: Micro Windows XP Windows 7 viena, Service Pack 3.version 2008-2009. First the code Quicksort_wmb was tested for its correctness on different samples of under following 3 categories (two test cases under each category)

1. Ascending order Input (Table 1):
2. Descending Order Input (Table 2)
3. Randomly generated input numbers (Table 3)

The results obtained for the above three categories (for simplicity only two are shown here) of data tests the correctness of the algorithm. In the above tables the first line indicates the input size of array, the second line requests for entering the input list of elements, no. of partitions displays the number of partitions happened when quicksort_wmb was executed. Finally we see the sorted array of elements displayed. The flag Aorder and Dorder is displayed when quicksort_wmb identifies the input array in ascending order or in descending order respectively and it had returned from quicksort_wmb after the first partition.

Further after confirmation of test cases for quicksort_wmb the time of execution for various samples of n ranging from 0…1000 for the following set of algorithms:
1. Quick sort for randomly generated array elements considering pivot as first element(column 3)
2. Quick sort for randomly generated array elements with randomized pivot (column 4)
3. Quick sort for ordered input array elements considering pivot as first element(column 5)
4. Randomized Quick sort for ordered input array elements with randomized pivot (column 6)
5. WMB Quick sort for any given array of elements (column 7)

A method specified by [1] under Bird’s Eye View is used to clock the time in my C++ code for all the above listed algorithms. We can notice from the Table 4 that the quick sort worst case time for n=1000 (column 6) is 0.467081 which is slowest in comparison with other experiments results (i.e. in the last row). For randomly generated input array with first element as pivot the time clocked is 0.093458 which shows that it is quite fast from that of worst case algorithm (column 3). The randomized pivot selection for the same has resulted the time as 0.09873 which is because of calling randomized function to select the pivot element (column 4). The column 6 which showcases the results for randomized Quick sort in which ordered input array of elements are considered but pivot is randomly generated has improved the speed by resulting the value of time clocked (for n=1000) as 0.064995 which is the speediest among the experiments considered so far. Finally we can see the worst made best which has resulted the time for ordered input as 0.1319. The respective graph plotted in figure 1.4 for the data in Table 4 gives a clearer statistics of speed and time complexity.

<table>
<thead>
<tr>
<th>enter the size of array :20</th>
<th>enter the elements</th>
<th>0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19</th>
<th>Aorder no. of partition:1</th>
</tr>
</thead>
<tbody>
<tr>
<td>sorted array is</td>
<td>0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19</td>
<td>Aorder no. of partition:1</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>enter the size of array : 50</th>
<th>enter the elements</th>
<th>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 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49</th>
<th>Aorder no. of partition:1</th>
</tr>
</thead>
<tbody>
<tr>
<td>sorted array is</td>
<td>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 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49</td>
<td>Aorder no. of partition:1</td>
<td></td>
</tr>
</tbody>
</table>

Table 1: shows the result obtained when ascending order input was considered for Quicksort_wmb
In this paper, I have shown how Quicksort can perform well from an angle which has not been discussed in the theories and analysis of quick sort. With fine tuning the quicksort_wmb one can use this to sort all possible cases of input data. Finally considering work done in this paper, we can claim that this modification will classify quicksort’s efficiency as Θ (n log n) when we have almost ordered or randomly ordered as worst and average case respectively, and Θ(n) when we have strictly ordered as best case. Further this paper also gives grounds to prove for its correctness formally. The paper also encourages one to consider design of program from various angles like compiler design, system architecture being used to execute the programs and similar other aspects. This paper also encourages a program designer to review the designed algorithms especially for sorting and searching.

Acknowledgement

First of all I would like to thank almighty for making this work possible. Heartfelt gratitude to Sartaj Sahni for supporting during the course of analysis. Special thanks to my colleagues at Vidyavikas Institute of Engineering and Technology, Mysore especially Aditya C. R Assistant Professor who has reviewed and pointed places where I had to improve, finally to my Head of the department Dr. Khalid Nazim S. A for his encouragement and motivation.

REFERENCES


