Quick Sort Algorithm

Song Qin
Dept. of Computer Sciences
Florida Institute of Technology
Melbourne, FL 32901

ABSTRACT

Given an array with n elements, we want to rearrange them in ascending order. In this paper, we introduce Quick Sort, a divide-and-conquer algorithm to sort an N element array. We evaluate the O(NlogN*)* time complexity in best case and $O(N^2)$ in worst case theoretically. We also introduce a way to approach the best case.

1. INTRODUCTION

Search engine relies on sorting algorithm very much. When you search some key word online, the feedback information is brought to you sorted by the importance of the web page.

Bubble, Selection and Insertion Sort, they all have an $O(N^2)$ time complexity that limits its usefulness to small number of element no more than a few thousand data points.

The quadratic time complexity of existing algorithms such as Bubble, Selection and Insertion Sort limits their performance when array size increases. Merge Sort needs more memory to create new arrays.

In this paper we introduce Quick Sort. It is accomplished by dividing the array into two partitions and then sorting each partition recursively.

Our main contribution is the introduction of a O(NlogN) sorting algorithm.

The next section describes some existing sorting algorithms: Bubble Sort, Insertion Sort, Selection Sort and Merge Sort. Section 3 provides a details explanation of our Quick Sort algorithm. Section 4 and 5 discusses theoretical and empirical evaluation based on efficiency. Section 6 summarizes our study and gives a conclusion.

Note: Arrays we mentioned in this article have the size of N.

2. RELATED WORK

Selection sort [1] works as follows: At each iteration, we identify two regions, sorted region (no element from start) and unsorted region. We “select” one smallest element from the unsorted region and put it in the sorted region. The number of elements in sorted region will increase by 1 each iteration. Repeat this on the rest of the unsorted region until it is exhausted. This method is called selection sort because it works by repeatedly “selecting” the smallest remaining element.

We often use Insertion Sort [2] to sort bridge hands: At each iteration, we identify two regions, sorted region (one element from start which is the smallest) and unsorted region. We take one element from the unsorted region and “insert” it in the sorted region. The elements in sorted region will increase by 1 each iteration. Repeat this on the rest of the unsorted region without the first element.

Bubble sort works as follows: keep passing through the list, exchanging adjacent element, if the list is out of order; when no exchanges are required on some pass, the list is sorted.

Merge sort [4] has a O(NlogN) time complexity. It divides the array into two subarrays each with N/2 items. Conquer each subarray by sorting it. Unless the array is sufficiently small(one element left), use recursion to do this. Combine the solutions to the subarrays by merging them into single sorted array.

In Bubble sort, Selection sort and Insertion sort, the $O(N^2)$ time complexity limits the performance when N gets very big. Although Merge Sort has the O(NlogN) time complexity, it needs more memory to create new arrays. Our algorithm will solve this problem and also approach a O(NlogN) time complexity.

3. APPROACH

Quick sort uses a divide-and-conquer approach. It is accomplished by dividing the array into two partitions and then sorting each partition recursively. In Quicksort, the array is partitioned by placing all items smaller than some pivot item before that item and all items larger than the pivot item after it. See figure 1. After each partition, the pivot item is in the right place. Also, each partition is decreasing. We can guarantee that until there is one element in partition, we will have a sorted array.

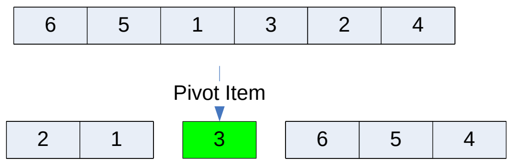

Figure 1.Partition

Problem: Sort Array a of N elements in ascending order

Input: Array a, index of first array element lo and last array element hi

Output: Array a in ascending order

```
 1|void quicksort (int[] a, int lo, int hi)
 2|{
 3|    int i=lo, j=hi, h;
 4|    int x=a[k%*N];

          //  partition
 5|    do
 6|   {
 7|        while (a[i]<x) i++;
 8|       while (a[j]>x) j--;
 9|        if (i<=j)
10|        {
11|            h=a[i]; a[i]=a[j]; a[j]=h;
12|            i++; j--;
13|        }
14|   } while (i<=j);

        //  recursion
15|    if (lo<j) quicksort(a, lo, j);
16|    if (i<hi) quicksort(a, i, hi);
17|}
```

Line1, function quicksort sort array a with N element.   lo is the lower index, hi is the upper index of the region of array a that is to be sorted.

Line2-4 is the partition part. Line 3, i is pointed to the first element of the array, j points to the last. Line 4, x is the pivot item which can be any item. k% is the percentage of the best performance of quick sort. We will use 50%=k% in this paper.

Line 5-16. After partitioning the sequence, quicksort treats the two parts recursively by the same procedure. The recursion ends whenever a part consists of one element only. Line 14 repeats line 6-13 until i >j. Line 7 searches for first element a[i] greater or equal to x.   Line 8 searches for last element a[j] less than or equal to x. Line 9-13, If i<=j, exchange a[i] and a[j] and move i and j.
Figure 2 illustrate the implementation of Quicksort. Given an array a[]={6,5,1,2}, 1 is chosen as the pivot item in line 1. i is pointed to the first element a[1], j points to the last a[4]. Compare a[1] with pivot, a[1] > pivot. Because a[4]<pivot, so j=j-1. Line 2, a[3]=pivot and a[1]>pivot, so exchange a[3] with a[1]. Line 3, i=j=2, so partition stops here. After this, 1 is in the right place. Line 4, 6 is chosen as pivot item. Swap 6 and 2. When i>j, stop partitioning.

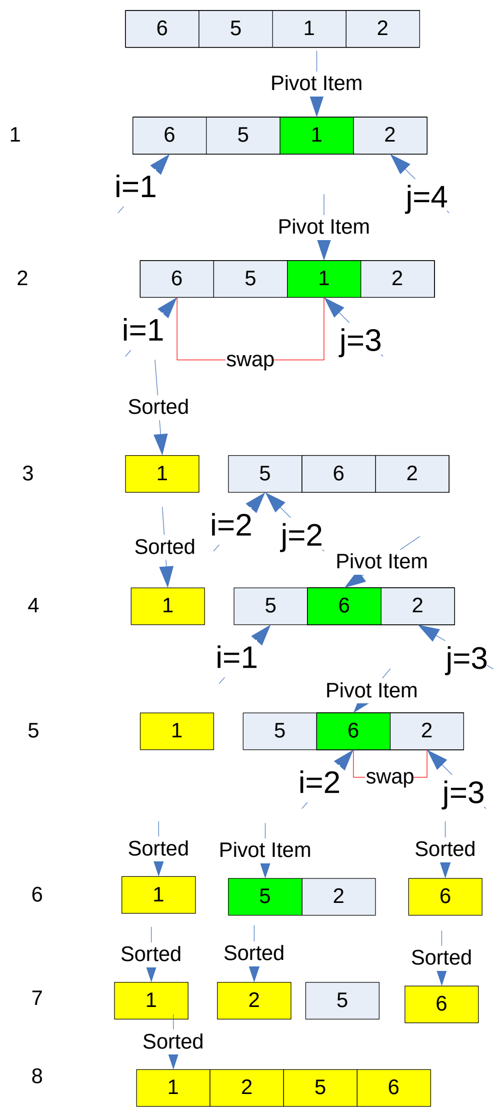

Figure 2. QuickSort Example

4. Theoretical Evaluation

Comparison between two array elements is the key operation of quick sort. Because before we sort the array we need to compare between array elements.

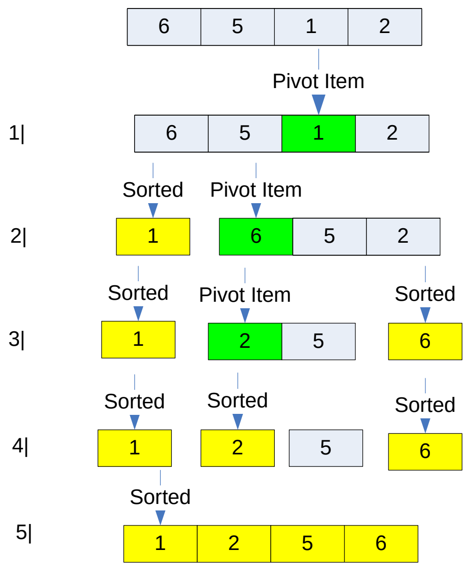

Figure 3.Worst Case of Quicksort

The worst case of quick sort occur when we choose the smallest or the largest element as the pivot. It needs the most comparisons in each iteration. See figure 3, we want to sort a=[6,5,1,2] . From line1 to 2, the smallest item *1* is chosen as the pivot. All elements need to compare with the pivot item once. From line 2 to 3, the largest item *6* is chosen as the pivot. All elements need to compare with the pivot item once. From line 3 to 4, *1* comparison is needed.

Let T(N) be the number of comparison needed to sort an array of N elements.
In the worst case, we need N-1 comparisons in the first round. Then we need T(N-1) comparisons to sort N-1 elements.

$T(N)=N-1+T(N-1)$ [1]

We need 0 time to sort *1* element:

$T(1)=0$ [2]

$T(N-1)=(N-2)+T(N-2)$ [3]

$T(N-2)=N-3+T(N-3)$ [4]

...

$T[N-(N-1)]=[N-(N-1)]+T[N-(N-1)]=1+T(0)=1$ [5]

$$\begin{aligned}T(N)&=(N-1)+T(N-1)\\&=(N-1)+(N-2)+T(N-2)\\&=(N-1)+(N-2)+(N-3)+T(N-3)\\&=(N-1)+(N-2)+(N-3)+\ldots+T(N-(N-1))\\&=(N-1)+(N-2)+(N-3)+\ldots+1\\&=(N^2+N)/2\\&=O(N^2)\end{aligned}$$

Thus, we have a $O(N^2)$ time complexity in the worst case.

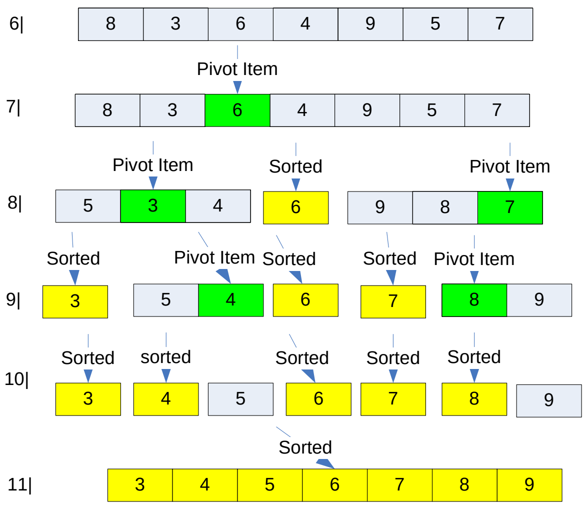

Figure 4.Best Case of Quicksort

The best case of quicksort occurs when we choose median as the pivot item each iteration. Figure 4 shows a example of the best case of Quicksort. Each partition decreases by almost half the array size each iteration. From line 7 to line8, after 6 times comparison, 6 is in the right place and we get partition {5,3,4} and {9,8,7}. It is the same with line 8-9. Line 10, when there are two elements left, the array is sorted.

Let T(N) be the number of comparison needed to sort an array of N elements. In the best case, we need N-1 comparisons to sort the pivot in the first iteration. We need T((N-1)/2) comparisons to sort each two partition.

$T(N)=N-1+2T((N-1)/2)$ [6]

We need *0* comparisons to sort *1* element.

$T(1)=0$ [7]

$T((N-1)/2)=2T((N-3)/4)+(N-1)/2-1$ [8]

Equation [8] replacing T((N-1)/2)) in equation [6] yields

$$\begin{aligned}T(N)&=2[(2T((N-3)/4)+(N-1)/2-1)]+N-1\\&=2^2T((N-3)/2^2)+N-3+N-1\end{aligned}$$ [9]

$T((N-3)/2^2)=2T((N-7)/2^3)+(N-3)/4-1$ [10]

Equation [10] replacing $T((N-3)/2^2)$ in equation [9] yields

$T(N)=2^3T((N-7)/2^3)+N-7+N-3+N-1$ [11]

$T(N)=2^kT(N-1+N-3+N-7+\ldots+N-(2^k-1))$

$$T(N) = 2^k T\left(\frac{(N-1+N-3+\cdots+N-(2k-1))2^k}{2^k}\right) + N-1+N-3+\cdots+N-(2^k-1)$$

$$=2^kT\left(\frac{N-2^k+1}{2^k}\right)+kN-(-k+2^{k+1}-2)$$ [12]

When $\frac{N-2^k+1}{2^k}=1$,array is sorted.
We have $T(N)=(N+1)\log(N+1)-2N$
Thus, $T(N)=O(N\log N)$

5.Empirical Evaluation

The efficiency of the Quicksort algorithm will be measured in CPU time which is measured using the system clock on a machine with minimal background processes running, with respect to the size of the input array, and compared to the Mergesort algorithm. Quick sort algorithm will be run with the *array size* parameter set to: 1k, 2k,…10k. To ensure reproducibility, all datasets and algorithms used in this evaluation can be found at "http://cs.fit.edu/~pkc/pub/classes/writing/httpdJan24.log.zip".
The datasets used are synthetic data sets of varying-length arrays with random numbers. The tests were run on PC running Windows XP and the following specifications: Intel Core 2 Duo CPU E8400 at 1.66 GHz with 2 GB of RAM. Algorithms are run in lisp.

5.1 Procedures

The procedure is as follows:

1 Store 10,000 records in an array
2 Choose 10,000 records
3 Sort records using merge sort and quick sort algorithm 10 times
4 Record average CPU time
5 Increment Array size by 10,000 each time until reach 10,000, repeat 3-5

5.2 Results and Analysis

Figure 5 shows Quick Sort algorithm is significantly faster (about 3 times faster) than Merge Sort algorithm for array size from 4k to 10k and less faster(about 1 time) when array size is less than 3k.

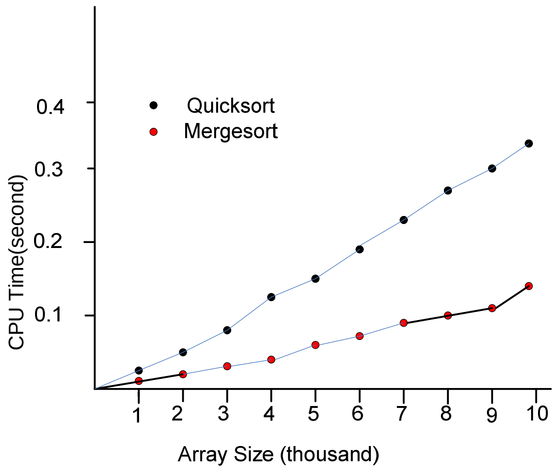

Figure 5.Runtime of Mergesort and Quicksort

By passing the regular significant test, we found that difference between merge and insertion sort is statistically significant with array size from 4k to 10k with 95% confident. (t=2.26, d.f.=9, p<0.05). Also, when array size is less than 3k, the difference is insignificant.

6. Conclusions

In this paper we introduced Quick Sort algorithm, a *O(NlongN)* time complexity sorting algorithm. Quick sort uses a divide-and-conquer method recursively sorts the elements of a list while Bubble, Insertion and Selection have a quadratic time complexity that limit its use to small number of elements. Merge sort uses extra memory to create new arrays.
Our theoretical analysis showed that Merge sort has a *O(NlogN)* time complexity in the best case and O(N^2) in the worst case.
In order to approach the best case, our algorithm tries to choose the median as the pivot. Quick Sort's efficiency was compared with Merge sort which is better than Bubble, Insertion and Selection Sort. In empirical evaluation, we found that difference between merge and insertion sort is statistically significant with array size from 4k to 10k with 95% confident. (t=2.26, d.f.=9, p<0.05). Also, when array size is less than 3k, the difference is insignificant.

One of the limitations is that in order to approach the best case the algorithm consumes much time on finding the median. How to find exactly the median in less time may needs future work.

REFERENCES.

[1] Sedgewick, Algorithms in C++, pp.96-98, 102, ISBN 0-201-51059-6 ,Addison-Wesley , 1992

[2] Sedgewick, Algorithms in C++, pp.98-100, ISBN 0-201-51059-6 ,Addison-Wesley , 1992

[3] Sedgewick, Algorithms in C++, pp.100-104, ISBN 0-201-51059-6 ,Addison-Wesley , 1992

[4] Richard Neapolitan and Kumarss Naimipour, pp.52-58. 1992
[5] C.A.R. Hoare: Quicksort. Computer Journal, Vol. 5, 1, 10-15 (1962)