Merge Sort: Awards sorting algorithm with approximate linear time complexity

Bingheng Wang
Dept. of Computer Science
Florida Institute of Technology
Melbourne FL 32904
Wangb2008@fit.edu

ABSTRACT
Given a list of elements, we rearrange the list with ascending or descending order. So far the existing approaches involve Bubble Sort, Insertion Sort, Selection Sort ($O(N^2)$ respectively in worst case) performing slow. Now we introduce a new algorithm Merge Sort to solve the problem with divide and conquer strategy. We prove the time complexity of Merge Sort is $O(N \log N)$ by theoretical evaluation and demonstrate the data of empirical evaluation for the better performance of merge sort compared with previous ones.

KEYWORDS
Merge Sort, Sorting algorithm.

1. INTRODUCTION

Motivation In the management system of banks, database is pivot component applying binary search. We have to sort the elements previously in order to apply binary search. Moreover, the existing approaches (selection sort, bubble sort and insertion sort) take time in order of $O(N^2)$, which means the comparisons jump with quadratic rate according to increment of size of elements.

Problem Given an array of elements (list(n) with n elements), containing numbers or letters, we rearrange the elements according to ascending or descending order with time complexity less than $O(N^2)$.

Approach We adopt the divide and conquer strategy as well as recursion to cut the time complexity $O(N \log N)$ to decrease the number of comparisons. In order to rearrange entire elements, we could divide the list into two subarrays until one elements left in each subarray, since one element is already sorted, which means we simply the original problem into smaller one that is already resolved. Then the resolved arrays would be merged together, consequently, the whole elements are sorted.

Contribution We introduced a new algorithm to solve the problem of rearranging number in array. Furthermore, we demonstrate the details of theoretical evaluation of merge sort that the time complexity of merge sort is $O(N \log N)$ in both best and worst case. In addition, the disparity between best and worst case is trivial. In light of experiment merge sort can shrink the execution time of sorting sharply compared with previous approaches.

Organization The next section is related work, followed by the description of Merge Sort section 3. Section 4 demonstrates the theoretical evaluation and empirical evaluation. Finally we get the conclusion.

2. RELATED WORK

Bubble Sort Bubble sort is one approach to resolve the problem by n passes. After the first pass the biggest number will be pushed to the last position identified the sorted region, furthermore the rest of part is unsorted region. After the subsequent pass the biggest number in unsorted region will float upwards to the sorted region as “bubble”. Then the sorted region will increase until the unsorted elements exhaust. Consequently, all elements have been sorted expectedly. Donald Knuth[1], in his famous The Art of Computer Programming, concluded that “the bubble sort seems to have nothing to recommend it, except a catchy name and the fact that it leads to some interesting theoretical problems”. Bubble Sort takes a time in order of $O(n^2)$ in worst case executing quadratic comparisons and $O(n)$ in best case, which means the disparity of comparison between best case and worst case is distinct.

Insertion Sort Insertion Sort demonstrated by Neapolitan & Naimipour[2] arranges the elements in the problem by sorting the n-1 passes. Before the first pass the first element is identified as the member of sorted region, since one element is already sorted, moreover the rest part is unsorted region. Then after the first iteration the first element of unsorted region is inserted into sorted region, more and more elements inserted into sorted region decrease the unsorted region until no element in unsorted region left. Consequently the elements of array have been sorted. Like the Bubble Sort Insertion Sort takes time in order of $O(n^2)$ in worst case.

Selection Sort In line with the algorithm of Selection Sort given by Neapolitan & Naimipour[3], in order to resolve the problem, we have to execute n-1 passes, first of which the smallest number is selected to switch with the first element, such that, one number is in correct position forming sorted region, and rest part is unsorted region. Subsequently the smallest number will be selected to increase sorted region by each following passes. Therefore after all the unsorted region exhausted, the list has been sorted. Selection sort also performances quadratic time complexity $O(n^2)$.

We can easily find Bubble Sort, Insertion Sort and Selection Sort demand for $O(n^2)$, that is, when n increases the comparisons jump at the speed of quadratic rate. Conversely, our algorithm requires fewer comparisons in worst case to improve the efficiency of sorting with divide and conquer strategy and recursive approach.
3. APPROACH
We adopt a distinct method, which is different from two regions way.

3.1 Merge Sort
In order to resolve the problem rearranging specific array we apply divide and conquer strategy including the following three key ideas.

Division we can divide the complex array into smaller ones recursively until the subarray could be resolved easily.

One element Since one element left in single subarray means the specific array is already sorted, we should terminate the division procedure until one element left in subarrays.

Merge After obtaining the sorted 1 element subarrays we merge the sorted subarrays recursively back to form entire array. Consequently we can sort the original array.

Input: int[] s (arbitrary numbers)
Output: int[] s (sorted numbers)

```java
1 function int[] divide(int[] s)
2    if(length of s>1)
3      int m=length of s/2;
4      int[] l;int[] r
5      l=divide(s from 0th to mth);
6      r=divide(s from m+1th to end);
7      return merge(l,r);
8  function int[] merge(int[] first,int[] second)
9  int pf;int ps; int j;
10 while(pf<length(first)&& ps<length(second)){
11   if(first[pf]<=second[ps])
12    append s[j] first[pf] pf++
13   else append s[j] second[ps];ps++; j++
14   if(length(first)>0)
15    append rest(first) to s[j];
16   if(length(second)>0)
17    append rest(second) to s[j];
18  return s[];
```

Given the array of arbitrary numbers(int[] s)(line 1), Function divide (Division) aims to divide the relative source array into two subarrays (line 1 & 2) recursively (line 5 & 6) until only One element left in relative source array, followed by the application of function merge (Merge) in line 7 which inputs two subarrays (l, r representing left subarray and right subarray) iterating and compare each element from left with the assistance of variable pf for l, ps for r as pointer (line 9) respectively, by which the smaller number perches the left position to the greater one back to source array, to which subsequently the rest of first and second (line 15&16) is appended, such that all the numbers could be arranged in ascending order. Similarly, after the recursive process from line 5 to 7 has been accomplished, the arbitrary array could be sorted in ascending order (int[] s).

Figure 1 demonstrates one instance during the pivot step(Merge) including two arrows representing two pointers for two already sorted subarrays respectively. The four groups of boxes is the process of obtaining merged array. The red and black arrows iterate each element above boxes from first element respectively. Since 1 pointed by red arrow is smaller than 3 pointed by black arrow according to (a), then 1 perches the 1st box and red arrow moves forwards, followed by (b) showing 3 locating the next box and black arrow moving to next box. Similarly all elements are in ascending order finally in figure (d).

In line with Figure 2, which illustrates the entire process of Merge Sort, we can see that from step a to d, the original array has been divided into two subarrays recursively, and finally 8 separate one element subarrays locate at the bottom of first section for division. Subsequently the second section for merge from step e to g has merged two subarrays together and at last we obtain the sorted array.

Now we will demonstrate the theoretical evaluation based on time complexity for Merge Sort.

3.2 Criteria
Since the number of comparison in merge sort determines the speed of the algorithms, that is, comparison is one of main process before sorting to determin if the the numbers need to be switched. Thus, we consider the comparison as the key operation counted for theoretical evaluation.

3.3 Worst case for Merge Sort
When we observe the pseudocode of merge sort, we can find that if the last process of copying rest of number executes once, which means all of the number except the last one in two subarrays should be compared, more specific, the greatest and second greatest number perch in difference arrays respectively, the worst case occurs.
3.4 Best case for Merge Sort

Against the worst case, best case occurs, when the largest number is smaller than any numbers in the other array, since after all the numbers in first subarray have been compared, second subarray could be copied directly, more specific, the last process of coping should be executed for all numbers of second subarray.

According to equation [7], \( T(N) \) is the number of comparison, where \( N \) is the array size. 2 represents two subarrays, \( T\left(\frac{N}{2}\right) \) is previous number of comparison, followed by \( \frac{N}{2} \) meaning \( N \) numbers in array should be compared \( \frac{N}{2} \), while only the first subarray numbers should be compared. One element in array conducts no comparison for equation [8].

\[
T(N) = 2T\left(\frac{N}{2}\right) + \frac{N}{2} \quad [7] \\
T(1) = 0 \quad [8]
\]

In order to solve the equation [7], we can use equation [9] to substitute \( T\left(\frac{N}{2}\right) \) in [7].

\[
T\left(\frac{N}{2}\right) = 2T\left(\frac{N}{4}\right) + \frac{N}{2} \quad [9]
\]

Subsequently, we obtain the equation [10], in which \( \sum_{i=1}^{k} 2^{-i} \) represents 1 + 2 + 4 + \( \ldots \) + \( 2^{k-1} \).

\[
T(N) = 2^k T\left(\frac{N}{2^k}\right) + kN - \sum_{i=1}^{k} 2^{-i-1} \quad [10]
\]

Then we use \( N = 2^k \)and \( T(1) = 0 \) to simply equation [10] in order to eliminate \( T\left(\frac{N}{2^k}\right) \), and obtain [11].

\[
T(N) = kN - \sum_{i=1}^{k} 2^{-i-1} \quad [11]
\]

We get the final expression of \( T(N) \) [6] with \( k = \log_2 N \).

\[
T(N) = N \log_2 N + 1 - N \quad (n \in Z^+), \quad [6]
\]

In line with equation [6] we can obtain the time complexity of merge sort in worst case \( O(N \log N) \).
We get the final expression of $T(N)$ [12] with $k = \log_2 N$.

$$T(N) = \frac{1}{2} N \log_2 N \quad (n \in Z^*) \quad [12]$$

In line with equation [12] we can obtain the time complexity of merge sort in best case $O(N \log N)$.

4. **EMPIRICAL EVALUATION**

The goal of evaluation for merge sort is to demonstrate the efficiency with various array size as parameter compared with alternative. The datasets used in experiment can be accessed at [http://cs.fit.edu/~pkc/pub/classes/writing/httpdJan24.log.zip](http://cs.fit.edu/~pkc/pub/classes/writing/httpdJan24.log.zip).

4.1 Criteria

Among the existing approaches, Bubble Sort is the fastest algorithms. Thus in order to evaluate the efficiency of Merge sort, we can calculate the runtime of merge sort and bubble sort, which doesn’t depend on the number of processors running in the computer.

$$g(n) = \sum_{k=0}^{n} f_b(k) / \sum_{k=0}^{n} f_m(k) \quad \{n \in Z^*\}$$

$n$ is the array size, $f_b(k)$ and $f_m(k)$ represent the processor time for bubble sort and merge sort with k array size respectively. $g(n)$ is the multiple of sum processor time from 0 to n between bubble sort and merge sort.

4.2 Procedure

We can divide the data into three datasets-- 10000 per group, in which the arrays whose size (parameter) amounts to 1000, 2000, and so on are used in the test. We can run the merge sort and bubble sort on Pentium(R) 1.60GHz, 512 MB RAM, and implemented the two algorithms by Java 6.

4.3 Result and Analysis

In line with Figure 1 demonstrating the results of bubble sort and merge sort for three datasets, we can find patently that the lines of bubble sort mounted dramatically, that is the average processor time inclined to 4200 milliseconds at the array size of 6000, whereas the trend of merge sort went upwards steadily from 0 to 6000 milliseconds. The area under curve for bubble sort ($\sum_{k=0}^{6000} f_b(k)$), that is the sum of runtime from 0 to 6000, outstripped that of merge sort ($\sum_{k=0}^{6000} f_m(k)$), which means merge sort performed more effectively than bubble sort in the experiment.

<table>
<thead>
<tr>
<th>Array size</th>
<th>1000</th>
<th>2000</th>
<th>3000</th>
<th>4000</th>
<th>5000</th>
<th>6000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dataset1</td>
<td>2.35</td>
<td>2.40</td>
<td>2.52</td>
<td>3.89</td>
<td>6.25</td>
<td>10.08</td>
</tr>
<tr>
<td>Dataset2</td>
<td>2.38</td>
<td>2.42</td>
<td>2.50</td>
<td>3.97</td>
<td>6.30</td>
<td>10.20</td>
</tr>
<tr>
<td>Dataset3</td>
<td>2.31</td>
<td>2.39</td>
<td>2.51</td>
<td>4.02</td>
<td>6.37</td>
<td>9.83</td>
</tr>
</tbody>
</table>

Table 1 indicates the multiple of criteria meaning the efficiency of merge sort compared with bubble sort by increasing the array size, according to which the value of $g(n)$ increasing obviously respectively for three distinct datasets, that is, $g(n) > 0 (0 \leq n \leq 6000, n \in Z)$. Thus we can conclude the efficiency of merge sort jumped with the increment of array size compared with bubble sort.

5. **CONCLUSION**

In this paper we have introduced a new algorithm- Merge Sort- to sort the specific array with distinct strategy compared with Insertion Sort, Selection Sort and Bubble Sort, by which we can sort the entire elements of array by decomposing into simply unit, which is already sorted. Then the decomposed result would be merged again to form the expected sorted array. Moreover the mathematical analysis shows the time complexity $O(N \log N)$ beat the existing three algorithms with $O(N^2)$, that is, merge sort take less comparisons than that of bubble sort, insertion sort and selection sort. According to the data of empirical evaluation, bubble sort finished the sorting specific array with time 6 times than that of merge sort. Merge Sort has drawback obviously, which means the space complexity with $O(n)$ demands more memory to support the execution. In order to avoid the limitation the future improvement could related to a new method to shrink the space complexity.

6. **REFERENCE**
