An Efficient Sorting Algorithm by Computing Randomized Sorted Sub-Sequences Based on Dynamic Programming

Toqeer Ehsan[1], M. Usman Ali[2], Meer Qaisar Javed[3]

Faculty of Computing and Information Technology, University of Gujrat, Pakistan
[1] toqeer.ehsan@uog.edu.pk, [2] m.usmanali@uog.edu.pk, [3] qaisar.javed@uog.edu.pk

SUMMARY
A lot of sorting algorithms exist today which are based on different problem solving techniques and with different performance behaviors. Algorithms are judged by the running time and space complexity which they take to solve any specific problem. In this papers an efficient sorting algorithm has been introduced, this algorithm is based on dynamic programming technique which is used to solve the optimization problems and here to sort arrays with optimal merges. Algorithm uses a bottom-up approach to compute the pre-sorted sub-sequences of random lengths in a given array of numbers and then sorts the whole array after efficiently combining the identified sub-sequences by using dynamic programming technique. Overlapping sub-problems are identified while sorting the given array and dynamic programming keeps the track of all the overlapping sub-problems by memorizing the data in a tabular form which is the main theme of the mentioned technique. Running time of the algorithm is compared with the standard merge sort and the results are satisfying.

Keywords:
Asymptotic, Complexity, Dynamic programming, Sorted sub-sequence

1. INTRODUCTION

To sort an array of numbers is a very fundamental and most discussed problem in computer science under the study of design and analysis of algorithms. Almost all the books written on algorithms contain a portion of sorting arrays and lists. It is not only to understand the design and analysis of sorting algorithms, sorting is also used in a wide range of applications. It is used to sort numbers and records in database management systems, spreadsheets, text editors, priority queues and network protocols etc. So there are dozens of sorting algorithms that exist today to sort arrays. If a sorting algorithm can sort a given array of numbers then why we discuss other sorting algorithm? The answer is that, all sorting algorithms, with respect to performance, based on different techniques are different from each other in the context of efficiency and memory. All the algorithms do not perform well in all the situations. Some algorithms are efficient on smaller data and some are efficient on larger data set, some of them perform well on random data. Random data or input is very important in our case as we normally are encountered with the random data in real world.

Existing algorithms belong to different problem solving techniques, their sorting mechanisms are different from each other and they have different complexities. Some famous algorithms are discussed in this section just to make better understanding. For example bubble sort and selection sort, both sort the given array of size 'n' in $O(n^2)$ in worst case. They both belong to the brute force technique [3]. Merge sort and Quick sort belong to divide and conquer technique and they sort the given array in $\Theta(nlgn)$ time in average case. Best, worst and average case of merge sort is the same and it is considered the most stable and efficient comparison based sorting algorithm. Average case performance of quick sort is O(nlgn) and sometimes it performs better on random input so the study of average case of any algorithm is important [1][4]. Heap sort also sorts the given array of element in $\Theta(nlgn)$ and belongs to transform and conquer techniques. Some other sorting techniques are count sort, average sort, radix sort, rack sort, bucket sort etc [1][4].

Sorting algorithm presented in this paper is designed for the random inputs and sorts the random data on the bases of dynamic programming technique. If we have an array of numbers which is populated with the random numbers then some of the numbers may be sorted already, why should we sort the pre-sorted data again and again? This paper gives the answer to that question by designing an algorithm which is based on dynamic programming to save the number of comparisons by solving the overlapping sub-problems only once.

2. BOTTOM-UP METHOD BY SORTED PAIRS

Let us consider an array A below with length 10 and indexes start from 0 and end at 7.

0	*1*	*2*	*3*	*4*	*5*	*6*	*7*
21	35	90	64	75	30	12	89

Manuscript received September 5, 2013
Manuscript revised September 20, 2013

After computing the sorted pairs, array A would look like:

0	1	2	3	4	5	6	7
21	35	64	90	30	75	12	89

By examining the array critically, we would be able to know that some of the elements are already sorted, let's call them sorted pairs. For example first two numbers 21 and 35 are in sorted form. Next two elements 90 and 64 are also sorted but in reverse order we can just swap them to compute the next sorted pair so that would be (64,90). Similarly (30,75) and (12,89). If the length of the array is 'n' then there are at most n/2 total number of pairs. If the length of the array is odd then there are n/2+1 total number of sorted pairs. We can combine these sorted pairs by using an iterative algorithm. We can use the merge procedure from merge sort to combine the sorted pairs all together.

1.1 ALGORITHM STEPS

Step1: Compute all the sorted pairs in the array.
Step2: Combine the first two sorted pairs / sub-arrays then next two pairs up to end.
Step3: Move forward in each iteration by the multiple of 2.
Step4: Jump to Step2 (lgn) times.
To combine the sorted pairs we combine first two pairs then next two pairs and so on up to the last pair. After the first iteration the number of sorted sub-arrays would become half i.e. n/4. In our example after first iteration two sorted sub-arrays would left: (21,35,64,90) and (12,30,75,89). In the next iteration of the combination we will get a sorted array i.e. (12,21,30,35,64,75,89,90).

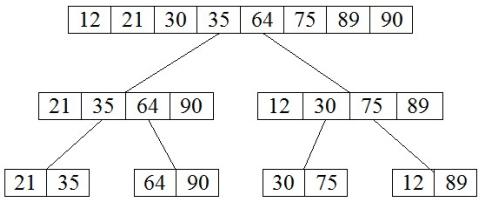

Figure.1: A bottom-up approach to combine the sorted pairs.

1.2 ANALYSIS

Computing the sorted pairs in an array of length 'n' is done in O(n) comparisons. Now combining the array of length 8 would call the merge procedure three times, similarly for the length 16 there would be four merge calls, 5 for 32 and so on. So there would be at most Θ(lgn) calls to merge procedure and height of the tree in the Fig.1 would also be Θ(lgn) . As we already know that the merge procedure of merge sort combines the two sub-arrays of length n/2 each in Θ(n). So the running time of this iterative bottom up sorting would be O(nlgn) + Θ(n) which is asymptotically equal to O(nlgn) and is same as merge sort.

3. RANDOMIZED SORTED SUB-SEQUENCES

The algorithm discussed in the section.2.1 performs well on random data as well as already sorted array either in ascending or descending order if we ignore the time to swap the element at the start. As we discussed at the start to analyze an algorithm, the real time data is very important and in our case the real time input is random. Now let's move beyond the computation of the sorted pairs and try to find the ordered sub-arrays also called sorted sub-sequences that can be further combined to sort the whole array.

3.1 COMPUTING SORTED SUB-SEQUENCE:

Let's consider the following array with random data the name of the array is B and the length of the array is 10.

0	1	2	3	4	5	6	7	8	9
2	4	9	7	6	3	8	1	5	6

Starting from '0' index to last index we compare the numbers, if the next element is greater or equal to the previous then move forward and if the next element is less than previous then stop and start computing the next sorted sub-sequence. So first sorted sub-sequence is [2,4,9] and for second sub-sequence we compare 7 and 6. 6 is less than 7 so we have to move forward but after doing that we would have a sequence of one element which makes no sense. We set the procedure of computing the sorted sub-sequences such that it keeps the track whether the comparison is first or not. If this happens in the first comparison then swap the numbers and further make the comparisons with the larger element so 7 is further compared with 3 and now 3 is less than 7 we start computing the next sequence.

3.2 COMBINING THE SUB-SEQUENCES:

After identifying the sorted sub-sequences in an array with random data we need to combine the sorted sub-arrays to sort the whole array. We can use the merge procedure of merge sort algorithm. Merge sort is considered the most stable and efficient sorting algorithm so using the merge procedure will allow us to compare the results with merge sort. We can combine these sub-arrays by two methods given below; both methods are bottom-up.

3.2.1 A STRAIGHT FORWARD APPROACH

Using this method we find the sub-sequences and then combine them, after merging we have one sequence so find the next sub-array and combine with the previous sorted sub-array and so on up to the end. After combine all the sub-sequences we will have the whole array sorted. The process of combining the array B is:

1- [2,4,9] will be combine with [6,7]
2- [2,4,6,7,9] will be combined with [3,8]
3- [2,3,4,6,7,8,9] will be combined with [1,5,6]

Now we get [1,2,3,4,5,6,6,7,8,9].
As we now already that the merge procedure performs Θ(n) comparisons to combine two sub-arrays of size n/2 each. So we can just sum the lengths of two sub-sequences to find the number of comparisons. Now case I performs 5 comparisons to merge them, case II performs 7 comparisons and case III performs 10 comparisons. Combining the sorted sub-arrays in the described fashion would take 22 comparisons and to find the sorted sub-sequences in the array of size 'n' takes 'n' comparisons which is same as computing the sorted pairs.

3.2.2 AN ALTERNATIVE APPROACH

In this method we first compute all the sorted sub-sequences then combine those sequences as we done for sorted pairs. In our example we have four sub-arrays as under.

1- [2,4,9]
2- [6,7]
3- [3,8]
4- [1,5,6]

After combining sequence 1 with 2 and sequence 3 with 4 we have only two sub-sequences. These two combinations would take 10 comparisons.

1- [2,4,6,7,9]
2- [1,3,5,6,8]

Combining these two sub-arrays we get [1,2,3,4,5,6,6,7,8,9] with 10 comparisons. Total 20 number of comparisons which are 2 less than the first method. It means the second approach is faster than first when we talk about the merge process only. But it may take more time while computing the sorted sub-sequences.

3.3 OVERLAPPING SUB-PROBLEMS

After computing first two sorted sub-sequences, our array B would look like this:

2	4	9	6	7	3	8	1	5	6

First sub-array is [2,4,9] and second sub-array is [6,7], now we combine them by merge call to the merge procedure. After combining first pair of sub-arrays we move further and compute next pair of sorted sub-arrays which is [3,8] and [1,5,6]. First iteration would be finished after combining these two sub-arrays because there are no further elements left in the array. After these two merges our array would be:

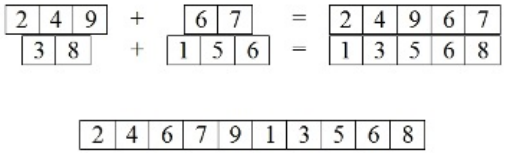

Figure.2: Overlapping sub-sequences.

Now in the next iteration we need to compute the sorted sub-sequences before combining them. So first sub-array is [2,4,6,7,9], which is computed by performing five comparisons. The point is that in the last iteration we already computed the sorted sub-sequences [2,4,9] and [6,7] which further merged together. But now we are computing them again. Similarly with the next sub-array [1,3,5,6,8] which is the combination of [3,8] and [1,5,6]. Solving the overlapping sub-problems again and again would result a very bad performance of the algorithm.

4. DYNAMIC PROGRAMMING (DP) SOLUTION

Dynamic programming is a famous problem solving technique that solves the optimization problems by solving the overlapping sub-problems once and save the results in tabular form so that this result could be used when the same sub-problem occurs rather solving the sub-problem again. In the section.3.3 we discussed the sub-problems which occur again and again in each iteration and we need to re-compute them. Dynamic programming finds the structure of an optimal solution then computes the solutions based on the pre-computed values [1]. Optimal solution is computed recursively in most cases but it is not necessary to design recursive solution always. In our algorithm, we are going to design an iterative bottom-up solution to our problem.

4.1 STRUCTURE OF THE SOLUTION:

Let's consider a new array C of length 14.
After computing the sorted sub-sequences we save the starting and ending index of each sub-sequence in a tabular form typically an array so that we can pick the indexes of

sorted sub-sequence from the table rather re-computing the sub-arrays again and again. After combining the sub-arrays we update the table as the number of sub-arrays will decrease after each iteration. Minimum length of any sorted sub-sequence is 2 so there would be at most n/2 sub-sequences for an array of length 'n'. We create a new array R with the size n/2+2 the last index of the array R will have value -1 so that we can find the end of the values and first index contains '0' which is the starting index of first sub-sequence.

0	1	2	3	4	5	6	7	8	9	10	11	12	13
2	14	19	7	6	3	18	1	5	6	3	25	12	9

4.2 MEMORIZATIONS

R[0] contains '0' as the start of the first sorted sub-sequence in C is C[0].
R[1] contains the ending index of first sorted sub-sequence.
R[2] contains the ending index of second sorted sub-sequence as the starting index of second sub-sequence is R[1]+1.
Similarly R[3] contains the ending index of third sorted sub-sequence and so on. After computing all sub-sequences, place -1 at the current index of R. After populating array R with the index the array would look like:

indexes	0	1	2	3	4	5	6	7	8	9
values	0	2	4	6	9	11	13	-1		

Array C will look like:

0	1	2	3	4	5	6	7	8	9	10	11	12	13
2	14	19	6	7	3	18	1	5	6	3	25	9	12

Values of array R after each iteration of the algorithm are shown in the Table.1. V_0 denotes the values of the array R before the algorithm starts merging the sub-sequences. Similarly the values of the array C after each iteration are shown in the Table.2.

Table.1: Array R Memorization of sorted sub-sequences.

indexes	0	1	2	3	4	5	6	7	8	9
V_0	0	2	4	6	9	11	13	-1		
V_1	0	4	9	13	-1					
V_2	0	9	13	-1						
V_3	0	13	-1							

Table.2: Array C elements after each iteration.

indexes	0	1	2	3	4	5	6	7	8	9	10	11	12	13
I_0	2	14	19	6	7	3	18	1	5	6	3	25	9	12
I_1	2	6	7	14	19	1	3	5	6	18	3	9	12	25
I_2	1	2	3	5	6	6	7	14	18	19	3	9	12	25
I_3	1	2	3	3	5	6	6	7	9	12	14	18	19	25

4.3 COMPUTING THE VALUES OF THE SUB-PROBLEMS

We use the merge procedure here which combines two sub-arrays. It takes array, starting index of first sorted sub-sequence, ending index of first sorted sub-sequence and ending index of next sub-sequence which is adjacent to the first one. Let's call this procedure as Combine.

k = 0, j =1 as the first element of R is always '0'.
Combine (B, R[k], R[k+1], R[k+2])
j = R[k+2]
k = k+2

After combining these two sorted sub-sequences we update array R such that the ending index of the second sub-sequence is assigned to the ending index of first sub-sequence i.e. R[k+1] = R[k+2]. Index 'j' is used to compute the indexes of array R so this assignment can be done by the statement R[j] = R[k+2]. After updating array R we move forward to the next pair of sorted sub-sequence by increasing the value of k by 2 units i.e. k = k + 2. Again call to combine procedure as: Combine (B, R[k], R[k+1], R[k+2]). Algorithm performs these steps until any of R[k], R[k+1] and R[k+2] gets -1. When -1 is encountered, it means that current iteration has been completed. Algorithm updates array R and places -1 at the end of the array and moves to the next iteration from start.

5. NON-RECURSIVE ALGORITHM

```
MEMORIZED-SORT(array, start, end)
Rsize = (end + 1)/2  + 1
Create a new array R of size Rsize
i=start
R[0]=start
flag = 0
j = 1
Seq_count=0
While(I < end)
        while( array[i]<=array[i+1] )
                flag = 1
                i=i+1
        if(!flag)
                swap array[i] and array[i+1]
                i=i+1
                flag = 1
        else
                R[j] = i
                i=i+1, j=j+1
                Seq_count=seq_count+1
                flag = 0
endwhile
```

```
if (i ==end)
        R[j]=i
        j=j+1
        Seq_count=seq_count+1

R[j]=-1
k=0
left=0
mid=0
right=0

While(seq_count >1)
        j=1
        k=0
        left=0
        mid=R[1]
        right=R[2]
        while(left!= -1AND mid!= -1ANDright!= -1)
                COMBINE(array,left,mid,right)
                //Merge procedure

                R[j] = right
                j=j+1
                Seq_count=Seq_count -1
                If (right == end)
                     break;  // If at the end then break

                k = k + 2 // To next sub-sequences
                left=R[k]+1
                mid=R[k+1]
                right=R[k+2]
        endwhile

        if( right==-1 OR mid == -1)
                R[j]=end
                R[j+1]=-1
        else
                R[j]=-1 // Place -1 at the end of R

endwhile
```

5.1 TIME COMPLEXITY

If there are 'm' sub-sequences then the algorithm will call the merge procedure m-1 times. If our array is of length 'n' then the maximum n/2 sub-sequences are computed. In first iteration algorithm combines n/2 sub-sequences by calling the merge procedure n/4 times. Second iteration combines remaining n/4 sub-sequences in n/8 merge calls. This process continues until there is only one sorted sub-sequence left which is equal to the original array and now in the sorted form. The elements of array R and C after each iteration are shown in the Table.1 and Table.2 respectively.

5.1.1 WORST CASE

When the input array of length 'n' is sorted in reverse order, we will have maximum number of sub-sequences and all the sub-sequences will be of length 2. First portion of the algorithm will compute the sorted sub-sequences in O(n) time. Total number of sub-sequences would be equal to n/2 and by combining those in one iteration will decrease the number of sorted sub-sequence to half i.e. n/4. Similarly n/8, n/16 up to 1 sorted sub-sequence whose length is equal to the original array. So there would be at most lg(n) number of iterations of the outer loop. If we draw a tree of these merges then the maximum height of the tree would be lg(n). Performing the merging of all the sub-sequences in one iteration would perform maximum $\Theta(n)$ number of comparisons as our algorithm merges the sub-arrays in an alternative way. So the running time of the algorithm in worst case would be $O(n) + \Theta(nlgn)$ which is equal to O(nlgn) asymptotically [1]. In short we can say that the worst case running time of Dynamic Programming sort is equivalent to bottom-up pair sort which is again practically efficient as compared to merge sort. We are using the upper bound to denote the complexity of the function because our algorithm starts combining the pairs not from single elements so it would save some merge calls.

5.1.2 BEST CASE

Best case running time of the algorithm is linear as it identifies the sorted array. In this case there would be only one sorted sub-sequence so the second portion of the algorithm will not be executed that's why the best case running time of the algorithm is O(n).

5.1.3 AVERAGE CASE

Average case analysis is performed by computing the running time of the algorithms on random inputs [4]. In any case the first portion of the algorithm will run by O(n) times for sure. Our algorithm based on dynamic programming is very efficient as compared with the merge sort on real time random inputs but we do not have any notations to compute the complexity function other than O(nlgn). The running time of algorithm depends on the number of sorted sub-sequences, more the number of sub-sequences more running time less the sub-sequences less running time. Running time of the algorithm would be somewhere between best and worst cases. Asymptotically we can denote the running time of our algorithm as O(nlgn) because we cannot change the class of efficiency for our algorithm. But we can run this algorithm on random inputs with different lengths of the array and show the difference by comparing the results with the recursive merge sort algorithm.

5.2 SPACE COMPLEXITY

Memorized-Sort is also memory efficient even it creates a new array R for memorization. If we consider the input size as 'n' means the length of the array then a new array is also created of size n/2+2 so by analyzing the size of both arrays we get n+n/2+2 memory spaces which is asymptotically equal to Θ(n). It is concluded that space complexity of memorized-sort is linear.

6. RESULTS

The algorithm is implemented using C++ and has been tested on Intel(R) Core (TM) i3 CPU with 4.00 GB memory space. Results are drawn by executing the program from small to large inputs. When the input array is already sorted, Mem-Sort is very fast as compared to merge sort. Performance on reverse and random inputs is astonishing and is much faster than merge sort in both cases. Fig.3, Fig.4 and Fig.5 below show the running time of Mem-Sort on sorted, reverse and random inputs respectively. Red line shows the behavior of merge sort and blue line shows the behavior of Mem-Sort which is very efficient as the graphs are self illustrative.

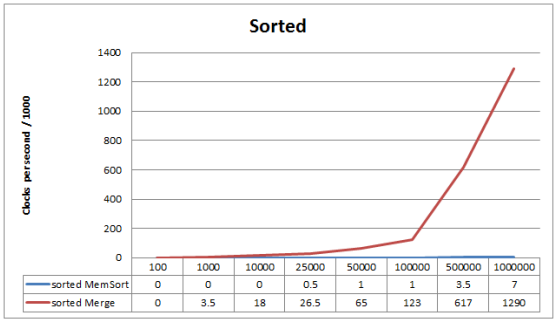

Figure.3: Running time on sorted inputs.

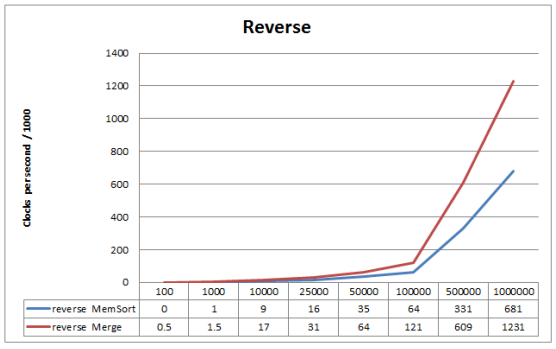

Figure.4: Running time on reverse inputs.

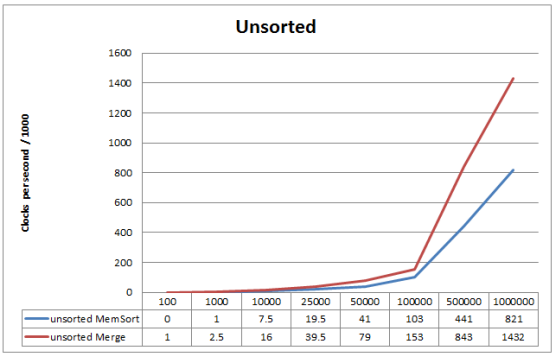

Figure.5: Running time on random inputs.

7. CONCLUSION

Dynamic programming is considered as a classical technique to solve the optimization problems but it could also be used to solve other computer science problems with some modifications. In this paper we have used the dynamic programming approach to develop an efficient sorting algorithm by computing random sorted sub-sequence with memorizations. Our algorithm further sorts the sorted sub-sequences by using the merge procedure from merge sort algorithm. Results of the designed algorithm are encouraging on all type of inputs. It can also identify if the array is already sorted so no need to sort the elements again. Mem-Sort's main target was to sort an array of random numbers efficiently and it achieved its target. As the basic operation of any comparison based sorting algorithm is the comparison operation of two elements, our algorithm performs optimal number of comparisons to sort a random array. Dynamic programming can also be used to find the most optimal number of comparisons before sorting any array so that we can perform sorting efficiently.

REFERENCES

[1] T. Cormen, C. Leiserson, R. Rivest, C. Stein, *Intoduction to Algorithms*, 3rd ed, MIT Press, 2011.

[2] Deepak Abhyankar, Maya Ingle, Elements of Dynamic Programming in Sorting, *International Journal of Engineering Research and Applications (IJERA), 1(3)*, 446-448.

[3] S. Baase and A. Gelder, *Computer Algorithms:Introduction to Design and Analysis*, Addison-Wesley, 2000.

[4] Anany Levitin, *Introduction to the Design and Analysis of Algorithms*, 2nd ed, Pearson Education, 2007.

[5] D. E. Knuth, *The Art of Computer Programming*, Vol. 3, Pearson Education, 1998.

[6] D. Abhyankar, M. Ingle, A Performance Study of Some Sophisticated Partitioning Algorithms, *International Journal of Advanced Computer Science and Applications* (IJACSA), *2(5)*, 2011, 135-137.

[7] Frederic H. Murphy, Edward A. Stohr, A Dynamic Programming Algorithm for Check Sorting, *Management Science, 24(1)*, September 1977, 59-70.
[8] Dr. Anupam Shukla and Rahul Kala, Predictive Sort, *International Journal of Computer Science and Network Security* (IJCSNS), *8(6)*, June 2008, 314-320.

Toqeer Ehsan is a Lecturer in the Department of Computer Science at University of Gujrat, Pakistan. He completed his Master of Science in Computer Science from University of Management and Technology in 2010 and Bachelor of Science (Hons) in Computer Science from Punjab University College of Information Technology in 2007. He is serving University of Gujrat since 2010. He also has a three years work experience of Web Development and Database Management. His research interest includes Algorithm Design and Analysis, Formal Language Theory, Language Engineering, Information Theory and Communication Networks.

Muhammad Usman Ali is an Assistant Professor in the Dept. of Information Technology at University of Gujrat, Pakistan. He completed his Master of Science in Computer Engineering from University of Engineering and Technology Texila in 2010. He is serving University of Gujrat since 2011. He also has a seven years work experience as a Principal Software Engineer. His research interest includes Computer Vision, Machine Learning, Language Engineering and Algorithm Design & Analysis.

Meer Qaisar Javed is an Associate Lecturer in the department of Information Technology at University of Gujrat, Pakistan. He completed his Master of Science in Information Systems from Linnæus University, Sweden, in 2013 and BS (Hons) in Computer Science from Hajvery University in 2006. He is teaching in the department of IT since 2011. He also has a two years work experience as a Business Analyst. His research interest includes Information Systems, Human Computer Interaction, and Algorithm analysis & design.