ISSN 2278-3091
Volume 9 No.2, March - April 2020
International Journal of Advanced Trends in Computer Science and Engineering
Available Online at http://www.warse.org/IJATCSE/static/pdf/file/ijatcse169922020.pdf
https://doi.org/10.30534/ijatcse/2020/169922020

SLSort (Smallest-Largest Swap Sort) – A Sorting Algorithm

Hirkani Padwad[1]
[1] Assistant Professor, Department of Computer Science & Engineering, Shri Ramdeobaba College of Engineering and Management, Nagpur, Maharashtra, India, hirkani.pathak@gmail.com

ABSTRACT

Sorting problem is one of the classic problems of Computer Science. Sorting is among the most basic tasks involved in computation and is very commonly used in almost every application. Design of a fast sorting algorithm has always been a challenge and there has been a lot of research since many decades. A variety of sorting algorithms have been designed amongst which Quicksort is considered as the fastest in-place sorting algorithm and Timsort is being used in practically for sorting problems in programming languages like Java and Python. Few parallelized versions of sorting algorithms have also been developed. This paper proposes a Divide and Conquer based scheme of in-place sorting which sorts the array using repeated swap operations of largest and small elements of two sub-arrays of the original array. The procedure is recursively applied on the sub-arrays until they are reduced to size 1. Analysis and results show that the algorithm performs better than quicksort for an input that is almost or completely sorted.

Key words: Sorting, divide and conquer, n log n sort

1. INTRODUCTION

For decades sorting has been one of the most basic and principle problems of computer science. Many sorting algorithms have been proposed since long time. But, very few algorithms have achieved the goal of fast sorting.

This paper presents a fast sorting algorithm which employs “Divide and Conquer” strategy similar to Quick Sort and Merge Sort algorithms. It recursively divides the given list into 2 sub-lists until finally sub-list with only 1 item is left. The algorithm uses an iterative swap operation which ensures that the all items in 1st sub-lists are lesser than all items in the 2nd sublist.

2. LITERATURE REVIEW

2.1 Quicksort[2]

Quicksort algorithm uses Divide and Conquer strategy which partitions the input list on the basis of a pivot element, such that items less (greater) than pivot element are placed in the left(right) side of pivot and then the algorithm is recursively called on the left and right sub list of pivot. In quicksort the partition operation, which loops over the elements of the list once, uses O (n) time. In the best case, each time a partition operation divides the list into two nearly equal parts. This means each recursive call processes a list of half the size of original list. Therefore, the depth of the recursion tree is log n. Each part of the list at every recursive call processes different items, thus, each level of calls needs only O(n) time in total. The result is that the algorithm uses only O (n log n) time. If the input list is already sorted or in reverse sorted order, the algorithm takes maximum time to complete. This is the worst case of Quicksort with O(n^2) time requirement. It happens because each partitioning operation results into a list of size 0 and a list of size n-1, thus causing n-1 partioning operations instead of log n. This problem can be solved by random selection of pivot. This version of quicksort is the randomized quicksort algorithm. Quicksort and mergesort have same running time of O(n log n),but quicksort is an in-place sorting algorithm unlike mergesort. This gives quicksort the advantage of space. Moreover, quicksort has a good cache locality which makes it practically faster than mergesort.

2.2 MergeSort[1][3]

Similar to quicksort, merge sort also uses Divide and Conquer strategy for sorting a list. The input list is divided into 2 halves from its middle position. This process continues recursively until a list of size 1 is reached. Then, the merge procedure starts which merges two lists in sorted order. Finally when all sub-lists are merged, we get a sorted list. In mergesort, the merge operations requires a total of O(n) operations on all partitions. There are O(log n) such partitioning operations. Thus total running time of this algorithm in all cases is O(n log n). However, during merging, a copy of the entire array is created. This copy operation copies more than a constant number of items, so merge sort is not an in place sorting algorithm. There are some applications where space is critical, and thus in-place algorithms are preferred over mergesort.

If data is present on disk, mergesort can be used in combination with quicksort as quicksort alone cannot work due to a large number of disk accesses. Mergesort is a great algorithm for sorting linked lists, because it does not rely on random access to elements as does heapsort or quicksort

2.3 TimSort[4][5][6]

TimSort is a hybrid sorting algorithm based on Insertion Sort and Merge Sort. It is a stable sorting algorithm that works in O(n log n) time. The list is divided into blocks (until the block size reaches a particular threshold) that are sorted using insertion sort one by one and then the sorted blocks are merged using merge operation used in merge sort. If the size of block is less than the threshold, then the list gets sorted just by using Insertion Sort. The size of threshold may vary from 32 to 64 depending upon the size of the input list. This algorithm is used in practical sorting implementations of Java (Array.sort()) and Python (sorted(), sort()). This algorithm has worst case time requirement of O(n log n).

2.4 Insertion Sort[1][7]

The insertion sort is an online sorting algorithm. It inserts each item at its proper position in the sorted sequence. It is an in-place and a stable sort that works by inserting the current item into the already sorted sequence of items by identifying its proper position and shifting the items that are greater than current item, one place to next position. Insertion sort has a complexity of O (n^2) in worst case and average case. The insertion sort is a almost twice as efficient as the bubble sort and almost 40% faster than the selection sort. Best case of insertion sort is O(n) and it occurs if the list is already sorted. The worst-case occurs when the list is in reverse order. The insertion sort is a good choice for sorting lists of a few thousand items or less and when all the items are not known in advance.

3. THE SLSort ALGORITHM

The SLSort algorithm recursively sorts a list by splitting it into 2 sub-lists and repeatedly applying a swap operation on these sub-lists. The basic working principle of this algorithm is based on QuickSort algorithm. Fig. 1 shows the working of SLSort algorithm.

The algorithm performs following operations to sort a list.

i) Partitioning the list
   The algorithm initially does a logical partitioning of the input list at middle position into 2 sub-lists, say, ‘A1’ (low to mid-1) and ‘A2’ (mid+1 to high)

ii) Computing the Largest item in ‘A1’
   After obtaining the middle position of the list, the algorithm computes largest item, say, ‘IL’ from the first sub-list i.e. ‘A1’

iii) Searching for an item in A2 that is less than ‘IL’ in ‘A1’
   In step 3, an item ‘IS’ is linearly searched in the 2nd sub-list i.e. ‘A2’ which is smaller than ‘IL’

iv) Swapping of IL with IS
   If, an ‘IS’ is identified in step 3, it is swapped with ‘IL’

v) Repeating Steps iii and iv till no IS is left in A2
   Step iii and iv are performed repeatedly until all items in A2 are either greater than or equal to the largest item in A1. This repetition ensures that no item in A2 is smaller than any of the items in A1. Then, the largest item in A1 is placed at ‘mid’ position. Thus, the list (problem) can finally be divided into 2 independent sub-lists (sub problems).

vi) Recursively sort modified A1 and A2

Once the two parts ‘A1’ (low to mid-1) and ‘A2’ (mid+1 to high) of the list are obtained such that {i1<i2, for all i1 in A1 and i2 in A2 }

That is, all items in ‘A1’ are less than every item in A2; these two sub-lists can be recursively sorted using same procedure explained above.

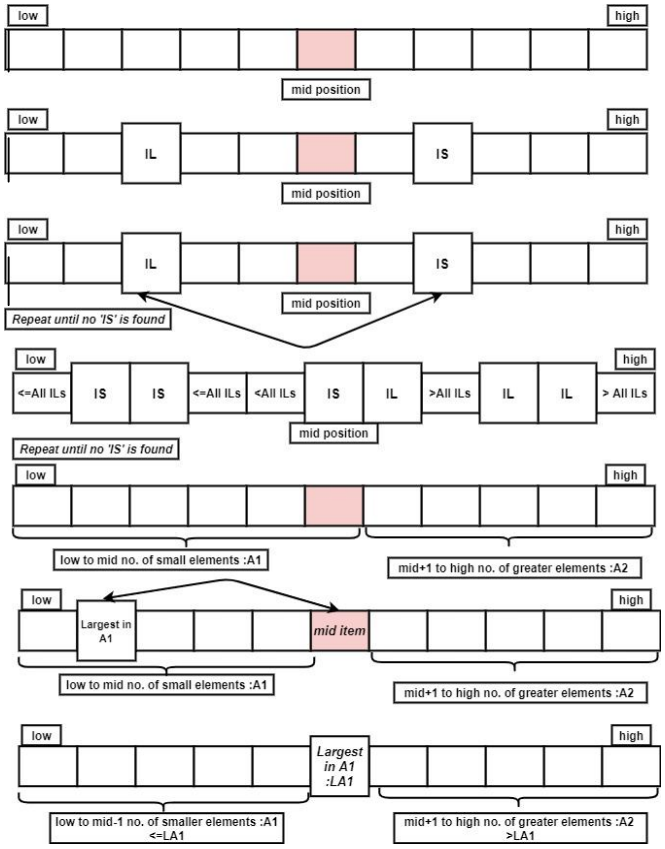

Figure 1: Working of SLSort algorithm

Algorithm

(a[] – input list, low (high) is lower (upper) index of list
IS- 1st item in A2 that is less than largest in A1

Algorithm SLSort(a[], low, high)

```
1.    set j:=0
2.  if(low<high) then
3.      mid=(low+high)/2
4.      while IS exists do
5.          largest=findLargest(a, low, mid)
6.          swap(a,largest,mid+1+j,high)
7.          j++
8.      end while
9.      swap a[largest] and a[mid]
10.         SLSort(a,low,mid-1)
```

11. SLSort(a,mid+1,high)
12. end if

Algorithm swap(a[],largest, low, high)
low is at (mid+1+j)th index
1. set i:=low
2. while(i<=high and a[i]>=a[largest]) do
3. i:=i+1
4. end while
5. if(i<=high)
6. temp:=a[largest]
7. a[largest]:=a[i]
8. a[i]:=temp

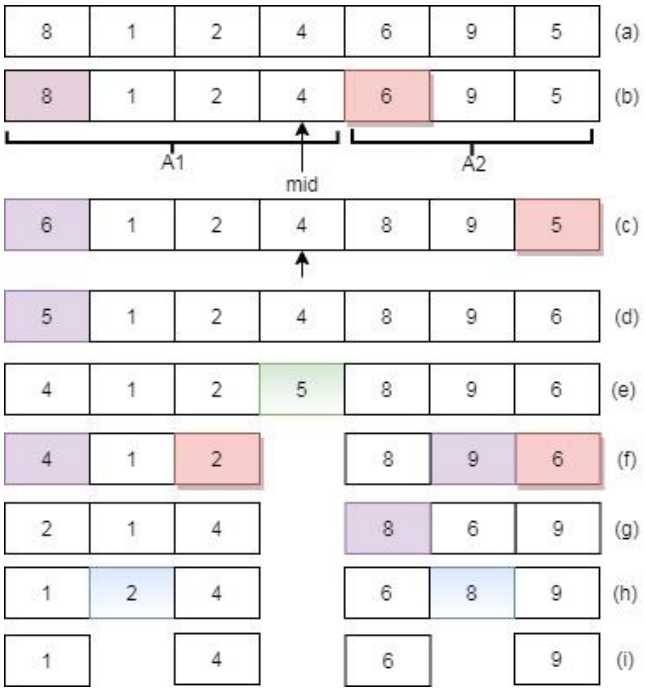

Figure 2: Example showing working of SLSort algorithm (a) input list (b) shows mid position and sub-lists A1, A2 with IL=8 and IS=6 (c) shows modified list after swapping 8 and 6 and the next IL=6 and IS=5 (d) shows modified list after swapping of 6 and 5 (e) shows list after swapping of largest item (5) in A1 with item at mid (f) shows how the list is split into 2 sub-lists, one from low to mid-1 and other from mid+1 to high (g)(h)(i) show recursive sorting of sub-lists.

The proof of correctness for SLSort involves a proof that Swap procedure works correctly and a proof that SLSort recursion works correctly.
The proof of correctness for Swap involves a loop invariant. The invariant is that all array elements from mid+1 to a[mid+j] are greater than largest

a[mid+1..mid+j]>largest in A1

and a[mid+j+i]th item is greater than largest in A1 and i<high before loop terminates. Therefore, at loop end, either no smaller item 'IL' is found or if it is found, it is swapped with largest item in A1. While loop at line no. 4 in SLSort ensures that all such ILs are shifted to sub-lists A1
Proof for Correctness of recursion algorithm SLSort is as follows.

Proof by induction on the length of the array, high - low + 1

1. When low = high SLSort does not do anything, which is the correct to do when sorting a list of length 1.
2. Assume that SLSort can correctly sort any list of length n or less
3. We show that it can correctly sort a list of length n+1.
   The line no. 3 of SLSort algorithm will partition the list of length n+1 into two sub-lists A1 & A2 and swap subroutine places smaller half from low to mid and greater (equal items in case of duplicates) half from mid+1 to high. This is followed by swapping of largest in A1 with mid item.
   That is, we end up with a[low..mid-1] ≤ a[mid] < A[mid+1..high]. These sub-lists will have length n/2, so the induction hypothesis tells us that the recursive calls to SLSort will correctly sort the two sub-lists. After the sub-lists are sorted, sorting is over.

4. RESULTS

The algorithm was implemented in C. The implementation was tested for input data of varying sizes and arrangements. Result of few test cases is shown in table 1. It is evident that for sorted and almost sorted input, SLSort algorithm performs significantly faster than quicksort algorithm. For unsorted input of input size in 100s, the SLSort algorithm performs at par with quicksort algorithm. For larger inputs, SLSort algorithm is slower as a result of findLargest() subroutine, however there is no significant increase in the execution time of SLSort as compared to Quicksort.

Table 1: Execution Time and No. of comparisons for SLSort and Quicksort

SN	SLSort	Quicksort
n= 1000 sorted	Time 0.00015 seconds Partition count =511 No of swap iterations=4049 No of comparisons in largest=3938	Time 0.00472 seconds Partitions=999 Split iterations =499500
n= 10000 Almost sorted	Time 0.00203 seconds Partition count =5904 No of swap iterations=59018 No of comparisons in largest=54613	Time 0.19267 seconds Partitions=9999, Split iterations =49995000
n= 20000 sorted	Time 0.00433 seconds Partition count =11808 No of swap iterations=128027 No of comparisons in largest=119221	Time 0.72281 seconds Partitions=19999, Split iterations =199990000
n= 100000 sorted	Time 0.01291 seconds Partition count =65535 No of swap iterations=753916 No of comparisons in largest=715030	Time 18.00247 seconds Partitions=99999, Split iterations =704982704
100 sorted	Time 0.00001 seconds Partition count =63 No of swap iterations=261 No of comparisons in largest=219	Time 0.00005 seconds Partitions=99, Split iterations=4950
300 sorted	Time 0.00004 seconds Partition count =172 No of swap iterations=1014	Time 0.00047 seconds Partitions=299, Split iterations=44850

	No of comparisons in largest=884	
100 unsorted	Time 0.00010 seconds Partition count =63 No of swap iterations=1655 No of comparisons in largest=3551	Time 0.00003 seconds Partitions=65, Split iterations=652
500 unsorted	Time 0.00153 seconds Partition count =255 No of swap iterations=6951 No of comparisons in largest=88862 Total=96068	Time 0.00016 seconds Paritions=323, Split iterations=4837
1000 unsorted	Time 0.00540 seconds Partition count =511 No of swap iterations=27063 No of comparisons in largest =370050	Time 0.00036 seconds Paritions=668, Split iterations =10931
5000 unsorted	Time 0.05471 seconds Partition count =2952 No of swap iterations=706598 No of comparisons in largest= 9073019	Time 0.00080 seconds Paritions=3326, Split iterations =78338

Number of partitioning operations is same in all cases: (log n)

5. PERFORMANCE ANALYSIS

The algorithm performs various operations viz. calculation of middle position, computing the largest item in A1, searching for an item in A2 that is smaller than the largest item in A1 followed by a swap operation, partitioning of the list into 2 sub-list and recursively sort these sub-lists. The running time of all the operations is computed separately.

Calculation of middle position and swapping of two items requires a constant time and hence are O(1) operations.

As evident from fig. 3 and fig. 4 the depth of recursion tree is $\log_2 n$, that is, the number of partitioning operations is $\log_2 n$, with n as the size of input list.

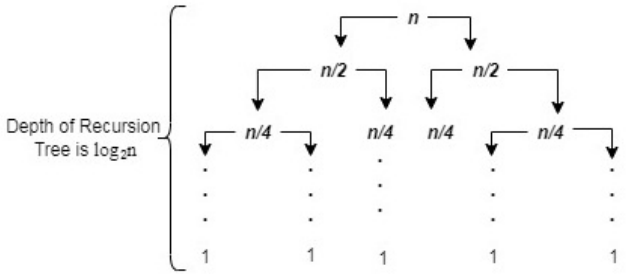

Figure 3: Recursion tree for a list of size n

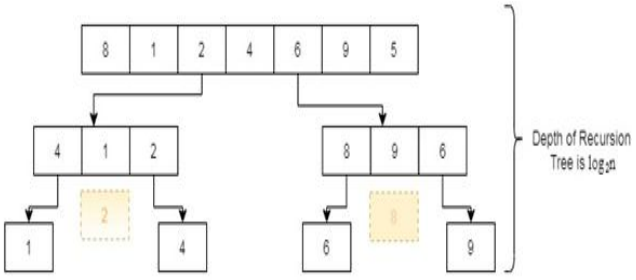

Figure 4: Recursion tree for n=7

Initially the algorithm takes full array as the input. Mid position is present at n/2th item in the list. Therefore, finding largest item 'IL' in A1 requires n/2 i.e. (n) comparisons and finding an item in A2 that is smaller than IL requires maximum n/2 comparisons which again accounts to O(n) operations. Thus, the total time requirement is O(n) for 1 swap operation. There can be upto n/2 i.e. O(n) such swap operations.

Since, there are $\log_2 n$ partitioning operations, hence the total time requirement of this algorithm in worst case is $O(n^2 \log n)$. However, the expected number of comparisons in swap() procedure are much less than n on the average, resulting into a smaller constant factor.

Best case of this algorithm occurs if the list is already in sorted order. If the input is sorted, the while loop in SLSort algorithm at line number 4 executes only once as there is no item in A2 that is smaller than the largest in A1. So, the total number of comparisons required for finding largest is n/2, which is $\theta(n)$ and that required for searching an 'IS' in A2 is again n/2, which is $\theta(n)$. Therefore, the total time requirement is O(n log n) in best case.

If the input is almost sorted, the number of calls to swap() procedure are a small constant. Thus, the algorithm performs much better than $O(n^2 \log n)$, as the factor of n^2 reduces to O(n).

The performance of this algorithm can be further improved by studying the pattern (arrangement) of items in the input list. If smallest item in A1 is also computed alongwith the largest item and if same procedure is repeated on A2, then, a simple comparison of largest item in A1 and smallest item in A2, followed by swapping of all items in A1 with all in A2, can save the number of comparisons by a factor of n in each recursive call, thus resulting in a running time of O(n log n). This optimization can work if the input list is completely unsorted.

6. CONCLUSION

The algorithm partitions the array by repeatedly swapping larger and smaller items and the same procedure is recursively applied on the sub-arrays. Thus, it uses a Divide and Conquer paradigm to sort an array. The SLSort algorithm performs better if the input array is almost or completely sorted. This is because, for an almost sorted input, the array is traversed only once in each partition. Since, the algorithm does not use any auxiliary space; hence SLSort is an in-place sorting algorithm. Similar to quicksort and mergesort [9], parallelized and concurrent version of this algorithm can be designed for faster execution on large inputs. Further, the worst case performance of the algorithm can be improved by the proposed optimization, for completely unsorted list.

REFERENCES

1. Donald E. Knuth et al. ***The Art of Computer Programming***, 2nd ed. Volume 3, Adssison-Wesley, Sorting and Searching,pp.212-224, pp.115-118
2. Cormen et al. ***Introduction to Algorithms***, 3rd ed., City : pub, 2009, ch. 2, pp. 170-190

3. Skiena, Steven S. ***4.5: Sorting by Divide-and-Conquer***. The Algorithm Design Manual, 2nd ed. Springer, pp. 120–125, 2008
4. Wikipedia: ***Timsort*** "Class: java.util.TimSort<T>".
5. Peters, Tim,"[Python-Dev] Sorting", Python Developers Mailinglist
6. Nicolas Auger, Vincent Jugé, Cyril Nicaud, and Carine Pivoteau. ***On the Worst-Case Complexity of TimSort***, 26th Annual European Symposium on Algorithms (ESA 2018), Leibniz International Proceedings in Informatics Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany, Article No. 4, pp. 4:1–4:13
7. Wikipedia: ***Sorting Algorithm***, Retrieved from http://en.wikipedia.org/wiki/Sorting_algorithm
8. John Harkins, Tarek El-Ghazawi, Esam El-Araby, Miaoqing Huang. ***Performance and Analysis of Sorting Algorithms on the SRC 6 Reconfigurable Computer***, The George Washington University, 2 Nov. 2005.
9. Dhirendra Pratap Singh, Ishan Joshi , Jaytrilok Choudhary , ***Survey of GPU Based Sorting Algorithms***, International Journal of Parallel Programming, Volume 46, 1017–1034,2018 https://doi.org/10.1007/s10766-017-0502-5