Enhanced Quicksort Algorithm

Rami Mansi
Department of Computer Science, Al al-Bayt University, Jordan

Abstract: *Sorting is considered as one of the important issues of computer science. Although there is a huge number of sorting algorithms, sorting problem has attracted a great deal of research; because efficient sorting is important to optimize the use of other algorithms. It is also often in producing human-readable output. This paper presents a new sorting algorithm called SMS-algorithm (Scan, Move, and Sort). The SMS algorithm is considered as an enhancement on the Quicksort algorithm in the best, average, and worst cases when dealing with an input array of a large size and when the maximum and the minimum values were small, especially when sorting a list of distinct elements. The SMS algorithm is compared with the Quicksort algorithm and the results were promising.*

Keywords: *SMS algorithm, Quicksort algorithm, large size array, distinct elements, time complexity, space complexity.*

Received August 25, 2008; accepted September 25, 2008

1. Introduction

Sorting has been a profound area for the algorithmic researchers. And many resources are invested to suggest a more working sorting algorithm. For this purpose many existing sorting algorithms were observed in terms of the efficiency of the algorithmic complexity [7]. Quicksort [8] was observed to be both economical and efficient. Many algorithms are very well known for sorting the unordered lists. Most important of them are Heap sort, Bubble sort, Quicksort, and Insertion sort [3]. Efficient sorting is important to optimize the use of other algorithms that require sorted lists to work correctly; it is also often in producing human-readable output [6]. Formally, the output should satisfy two major conditions:

- The output is in non-decreasing order.
- The output is a permutation, or reordering, of the input.

Since the early beginning of computing, the sorting problem has attracted many researchers, perhaps due to the time complexity of solving it efficiently [10]. As stated in [2, 4], sorting has been considered as a fundamental problem in the study of algorithms, that due to many reasons:

- The need to sort information is inherent in many applications.
- Algorithms often use sorting as a key subroutine.
- In algorithm design there are many essential techniques represented in the body of sorting algorithms.
- Many engineering issues come to the fore when implementing sorting algorithms.

In algorithm which uses divide-and-conquer approach, it divides the problem into smaller instances of the same problem, then solves that instances of the problem recursively (conquer), and then collects all solutions to get the main solution for the original input (combine). The principle of the divide-and-conquer algorithm design is that it is easier to solve several small instances of a problem than one large problem [5, 11].

In this paper, a new sorting algorithm is presented, called SMS-Algorithm (Scan, Move, and Sort). The study shows that the proposed algorithm is more efficient and faster as compared to the Quicksort algorithm when dealing with a large size (n) of the input array. So, we considered the SMS algorithm as an enhancement on the Quicksort algorithm. Section 2 presents the concept, steps, and the pseudocode of the SMS algorithm with an example. Section 3 introduces the detailed time and space analysis of the SMS algorithm. Section 4 presents a comparison between the SMS and the Quicksort algorithms. Also, a real-world case study simulation is introduces in section 5. Finally, conclusions were presented in section 6.

2. The SMS Algorithm

2.1. The Concept of the SMS Algorithm

The main concept of the SMS algorithm is distributing the elements of the input array on three additional temporary arrays. The sizes of these arrays are decided depending on the maximum and the minimum values of the input array. The first temporary array is called (PosArray) and contains the positive elements using the value of the element itself as its index in the array. The second array is (NegArray) which contains the negative elements using the absolute value of the element itself as its index in the array. The third array

is (FreqArray) and used to save the frequent elements of the input array.

2.2. The Steps of the SMS Algorithm

The SMS algorithm consists of three procedures, Scan, Move, and Sort. The first procedure is (Scan), which scans the array and gives the values of the minimum, the maximum, the number of positive elements, and the number of negative elements. Also, this procedure checks if *min* equals to *max*, then the input array is already sorted, otherwise, calls the procedure (Move).

The second procedure (Move) creates the three temporary arrays, FreqArray of size (*n*), PosArray of size (*max+1*), and NegArray of size (|*min*|*+1*), and then initializes the PosArray, the NegArray, and the FreqArray with the value (*min-1*) to denoting the indices that will be skipped in the next phase. Then, this procedure distributes the elements on the three arrays; the positive elements are saved in the PosArray using the element itself as its index, the negative elements are saved in the NegArray using the absolute value of the element itself as its index, and the frequent elements are saved in the FreqArray using a variable (*i*) as an index started from zero and incremented by one.

The third procedure (Sort) copies the elements of the NegArray starting from the last index with ignoring the values of (*min-1*). Then it copies the elements of the PosArray starting from the first index with ignoring the values of (*min-1*). The copying is done on the original input array with overwriting the original values with the sorted values. After each copying operation of an element from the NegArray and the PosArray to the original array, the procedure searches the FreqArray and copies all element that are equal to the element that copied in the last copying operation (current element).

2.3. The Pseudocode of the SMS Algorithm

The pseudocode of the first (Scan) procedure can be expressed as follows:

```
procedure Scan(array, size)
1   if size > 1 then
2     var a, max, min, NOP, NON
3     max:=array(0)
4     min:=array(0)
5     NOP:=0
6     NON:=0
7   for a:= 0 to size-1 do
8     if array(a) > max then
9       max := array(a)
10    else
11      min:=array(a)
12    end if
13    if array(a) ≥ 0 then
14      NOP:= NOP+1
15    else
16      NON:= NON+1
17    end if
18  end for
19  if min ≠ max then
20    Move(array, size, NOP, NON, max, min)
21  end if
22 end if
  end procedure scan
```

The pseudocode of the second (Move) procedure is expressed as follows:

```
Procedure Move(array, size, NOP, NON, max, min)
1   var b,c,d,i
2   i:=0
3   create a new array: FreqArray[size]
    and initialize by the value (min-1)
4   if NOP > 0 then
5     create a new array:PosArray[max+1]
6     for b:=0 to max do
7       PosArray(b):= min-1
8     end for
9   end if
10  if NON>0 then
11    create a new array: NegArray[|min|+1]
12    for c:= 0 to |min|+1 do
13      NegArray(c):= min-1
14    end for
15  end if
16  for d:= 0 to size-1 do
17    if array(d) ≥ 0 then
18      if PosArray(array(d))==min-1  then
19        PosArray(array(d)):=array(d)
20      else
21        FreqArray(i):=array(d)
22        i:=i+1
23      end if
24    else
25    if NegArray(|array(d)|)==min-1  then
26      NegArray(|array(d)|):= array(d)
27    else
28      FreqArray(i):= array(d)
29      i:= i+1
30    end if
31  end if
32 end for
33 Sort(array, NegArray, PosArray, FreqArray,
   NON, NOP, max, min, i)
end procedure move
```

The pseudocode of third (Sort) procedure is as follows:

```
procedure Sort(array, NegArray, PosArray, FreqArray,
NON, NOP,  max, min, I )
1   var index,x,y
2   index:=0
3   if NON > 0 then
4     for x:= |min| downto 0 do
5       if NegArray(x) ≠ min-1 then
6         array(index):= NegArray(x)
7         index:= index+1
8         for y:= 0 to i do
9           if FreqArray(y)==array(index-1)   then
10            array(index):= FrqArray(y)
11            index:= index+1
12          end if
13        end for
```

```
14        end if
15      end for
16    end if
17    if NOP > 0 then
18      for x:= 0 to max do
19        if PosArray(x) ≠ min-1 then
20          array(index):= PosArray(x)
21          index:= index+1
22          for y:= 0 to i do
23            if FreqArray(y)== array(index-1) then
24              array(index):=FrqArray(y)
25              index:= index+1
26            end if
27          end for
28        end if
29      end for
30    end if
end procedure sort
```

The following example illustrates the work of the SMS algorithm. If we have the following array to be sorted using the SMS Algorithm:

Original Array

8	-2	6	-4	8	3	0	2	6	-3

The size of this array is (10) elements. The first phase of the SMS algorithm (procedure Scan) gives the minimum value, the maximum value, the number of positive elements, and the number of negative elements. For this example, the *min* is (-4), the *max* is (8), the number of positive elements is (7), and the number of negative elements is (3). Since the number of positive elements and the number of negative elements are positive, the second phase of the algorithm (procedure Move) creates three new arrays. The first array is FreqArray of size (10), which is the size of the original array. The second is the PosArray of size (9), which is (*max+1*). The third array is NegArray of size (5), which is (|*min*|*+1*), (the absolute value of *min*, plus one). The elements of the three arrays will be initialized with the value (*min*-1), which is (-5). At this moment, the algorithm distributes the elements of the original array on the new three arrays, as follows:

FreqArray

8	**6**	-5	-5	-5	-5	-5	-5	-5	-5

PosArray

0	-5	**2**	**3**	-5	-5	**6**	-5	**8**

NegArray

-5	-5	**-2**	**-3**	**-4**

Note that the values in italic are the default values and will be skipped during the third phase. In the third phase (procedure Sort), the original array will be updated with the values of NegArray, PosArray, and FreqArray arrays, as follows, starting with the NegArray at the last element, (-4) will be placed at index (0) of the original array. Notice that the updated values are in bold.

Original Array

-4	-2	6	-4	8	3	0	2	6	-3

The task now is to copy all elements that are equal to (-4) from the FreqArray. Since there are no frequent values of (-4), the next negative element will be copied from the NegArray array into the original array.

Original Array

-4	**-3**	6	-4	8	3	0	2	6	-3

Also the last negative element (-2) will be copied.

Original Array

-4	**-3**	**-2**	-4	8	3	0	2	6	-3

Copying the positive elements from the PosArray array is differing from copying negatives. The difference is that the algorithm must start copying the positive elements from the first index of the PosArray, not from the last as in copying the negative elements. But as in copying the negative elements; the algorithm searches in the FreqArray for equal elements of the element being copied, as follows:

Original Array

-4	**-3**	**-2**	**0**	8	3	0	2	6	-3

Original Array

-4	**-3**	**-2**	**0**	**2**	3	0	2	6	-3

Original Array

-4	**-3**	**-2**	**0**	**2**	**3**	0	2	6	-3

Original Array

-4	**-3**	**-2**	**0**	**2**	**3**	**6**	2	6	-3

At this moment, the value (6) also occurs in the FreqArray and will be copied to the original array.

Original Array

-4	**-3**	**-2**	**0**	**2**	**3**	**6**	**6**	6	-3

Original Array

-4	**-3**	**-2**	**0**	**2**	**3**	**6**	**6**	**8**	-3

Also, the value (8) will be copied from the FreqArray.

Original Array

-4	**-3**	**-2**	**0**	**2**	**3**	**6**	**6**	**8**	**8**

3. Analysis of the SMS Algorithm

3.1. Time Analysis of Procedure Scan

The goal of procedure Scan is to get the maximum value, the minimum value, the number of positive elements, and the number of negative elements. This requires scanning the array and reaching each element one time in a single pass. The *for*-loop (lines 7-18 of procedure Scan) takes $\Theta(n)$ time complexity.

3.2. Time Analysis of Procedure Move

The best case of procedure Move is when all elements of the original array are positive and the *max* is small, or when all of them are negative and the *min* is small.

If all elements are positive and there are no negative elements, then the *for*-loop (lines 6-8 of procedure Move) takes O(*max*) time for initializing the PosArray, and the *for*-loop (lines 16-32 of procedure Move) takes O(*n*) time. So, in this case, the overall time complexity of procedure Move is O(*n* + *max*).

In the other hand, if all elements of the original array are negative, then the *for*-loop (lines 12 -14 of procedure Move) takes O(|*min*|) time for initializing the NegArray, and the *for*-loop (lines 16-32 of procedure Move) takes O(*n*) time. So, in this case, the overall time complexity of procedure Move is O(*n*+|*min*|). We may say that in the average and worst cases, if there are positive and negative elements in the original array, then the overall time complexity of procedure Move is O(*n*+*max*+|*min*|).

3.3. Time Analysis of Procedure Sort

The best case of procedure Sort is when all elements are positive and distinct and the *max* is small, or, when all of them are negative and distinct and the *min* is small.

If all elements are positive and distinct then the *for*-loop (lines 18-29 of procedure Sort) takes O(*max*) time, since the inner loop (lines 22-27 of procedure Sort) takes O(1) time in this case. And if all elements are negative and distinct, then the *for*-loop (lines 4-15 of procedure Sort) takes O(|*min*|) time, since the inner loop (lines 8-13 of procedure Sort) takes O(1) time in this case. So, we may say, the best, average, and worst cases of procedure Sort have Θ(*max***f*)+Θ(|*min*|**f*), where *f* is the number of frequent elements. In other words, the time complexity of procedure Sort is Θ(f*(*max*+|*min*|)).

The time complexity of the best case of the SMS algorithm is Θ(*n*), when the input array is already sorted. This means, when *max* is equal to *min* (lines 19-21 of procedure Scan) then the input array is already sorted. In the average and worst cases, procedure Scan takes Θ(*n*) time, procedure Move takes Θ(*n* + *max* + |*min*|) time, and procedure Sort takes Θ(*f**(*max*+|*min*|)) time. If we suppose a normal distribution of data, the frequency of elements should be little, and because most of real applications have *n* much greater than *max* and |*min*|, we may consider *max* and *min* as constants and eliminate them.

The overall complexity of the SMS algorithm in the average and worst cases is O(*n*+*f**(*max*+|*min*|)), where *f* is the number of frequent elements.

3.4. Space Analysis of the SMS Algorithm

The algorithm creates a new array (FreqArray) of size *n* to save the frequent elements. In the best case, if all elements of the original array were positive, the algorithm creates a new array (PosArray) of size (*max*+*1*), or, if all elements were negative, the algorithm creates a new array (NegArray) of size (|*min*|+*1*). So, in the best case, the algorithm needs O(*n*+*max*+*1*) or O(*n*+|*min*|+*1*) additional space. In the average and worst cases, the algorithm needs O(*n*+*max*+|*min*|+*2*) additional space.

4. Comparison with Quicksort Algorithm

Quicksort is a well-known sorting algorithm developed by Hoare [8] that, on average, makes *O*(*nlogn*) comparisons to sort *n* items. However, in the worst case, it makes Θ(n^2) comparisons. Quicksort is a comparison sort and, in efficient implementations, is not a stable sort, since stable sorting algorithms maintain the relative order of records with equal keys. This means, a sorting algorithm is stable if whenever there are two records *R* and *S* with the same key and with *R* appearing before *S* in the original list, *R* will appear before *S* in the sorted list [1, 9].

Quicksort is characterized as a "hard division and easy combination" algorithm. As mentioned in [16], there are three divide-and-conquer processes for sorting a typical sub array *A*[*p..r*]:

- Divide: the array *A*[*p..r*] is partitioned into two nonempty sub arrays, *A*[*p..q*] and *A*[*q+1..r*] such that each element of *A*[*p..q*] is less than or equal to each element of *A*[*q+1..r*]. The index *q* is computed as part of this partitioning procedure.
- Conquer: the two sub arrays *A*[*p..q*] and *A*[*q+1..r*] are sorted by recursive calls to Quick sort.
- Combine: since the sub arrays are sorted in place, no work is needed to combine them, and the entire array *A*[*p..r*] is now sorted.

The main steps of Quicksort as stated in [8, 12] are:

- Pick an element, called a *pivot*, from the list.
- Reorder the list so that all elements which are less than the pivot come before the pivot and so that all elements greater than the pivot come after it (equal values can go either way). After this partitioning, the pivot is in its final position. This is called the partition operation.
- Recursively sort the sub-list of lesser elements and the sub-list of greater elements.

The base cases of the recursion are lists of size zero or one, which are always sorted. The code of the Quicksort algorithm consists of two parts. The first part is a procedure (quicksort), which decides the correct place of the *pivot* and recursively divides the array into two parts [13].

The second part is the function (partition) which partitions the portion of the array between indexes *left* and *right*, inclusively, by moving all elements less than or equal to array[pivotIndex] to the beginning of the sub-array, leaving all the greater elements following them. In this process it also finds the final position for the pivot element, which it returns. It

temporarily moves the pivot element to the end of the sub-array, so that it does not get in the way. Because it only uses exchanges, the final list has the same elements as the original list [14, 15]. Notice that an element may be exchanged multiple times before reaching its final place. The main differences between SMS algorithm and Quicksort algorithm are:

- The SMS algorithm is stable (maintains the relative order of records with equal keys) but Quicksort is unstable sorting algorithm.
- In the best case, the SMS algorithm takes O(*n*) time while Quicksort takes O(*nlgn*) time to sort an array of size *n* elements.
- The SMS algorithm is faster than Quicksort algorithm when dealing with a large size (*n*) of the input array, and when the values (*max*) and (|*min*|) are much less than the value (*n*). In this case, the time complexity of the average and worst cases of the SMS algorithm approaches O(*n*), while Quicksort algorithm takes O(*nlgn*) in the average case, and O(n^2) in the worst case.
- The SMS algorithm is best used to sort an array of distinct elements. In this case, the value of (*f*) will be equals to 1, and the algorithm takes O(*n*+*max*+|*min*|) time.
- The SMS algorithm enhances the way that Quicksort algorithm divides the input array. Quicksort moves the *pivot* to reside in its correct place and then divides the array into two parts, and recursively it makes the same procedures for both parts, until it reaches the base case. Instead of doing this, the SMS algorithm divides the original array into three parts (arrays), positive, negative, and frequent elements, and moves each element into its correct place in a single pass.
- We may say that the SMS algorithm is also a divide-and-conquer method, since it divides, sorts, and then it combines.
- In the other hand, Quicksort algorithm needs O(*lgn*) additional space, as mentioned in [8], but the SMS algorithm needs O(*n*+*max*+|*min*|+2) to sort *n* elements.

5. Case Study

To prove that the proposed algorithm is faster than Quicksort when dealing with a large size of the input array and little values of *min* and *max*, especially when sorting an array of distinct elements values; a real-world case study has been simulated. In this case study, the SMS and the Quicksort algorithms have been applied to sort a list of (10,000) distinct elements.

The simulator was built using Visual C++ 6.0 and the built-in function (clock()) is used to measure the elapsed time of both algorithms on the same computer using the same data set. The *min* value of the input array was (1) and the *max* was (10,000). Figure 1 shows the interface of the used simulator.

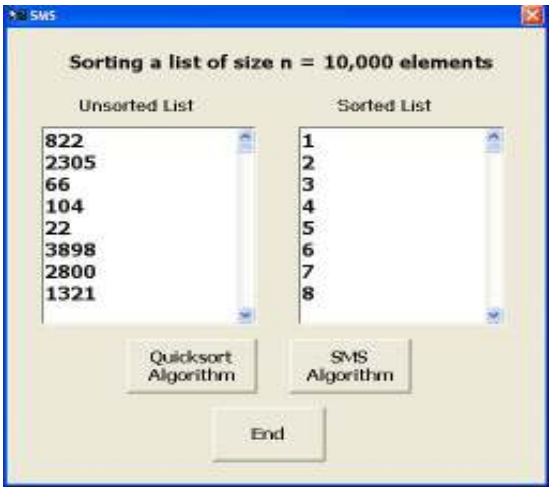

Figure 1. The interface of the simulator.

Table 1 shows the recorded elapsed execution time in milliseconds of Quicksort and SMS algorithms. These results represent the average execution time of the recorded execution times of multiple runs of the programs to sort the same data on the same computer under the same operating system.

Table 1. Execution time for Quicksort and SMS algorithms.

Algorithm	Elapsed Time
Quicksort	425 ms
SMS	259 ms

Figure 2 shows a comparison of the elapsed execution time in milliseconds of the Quicksort and the SMS algorithms.

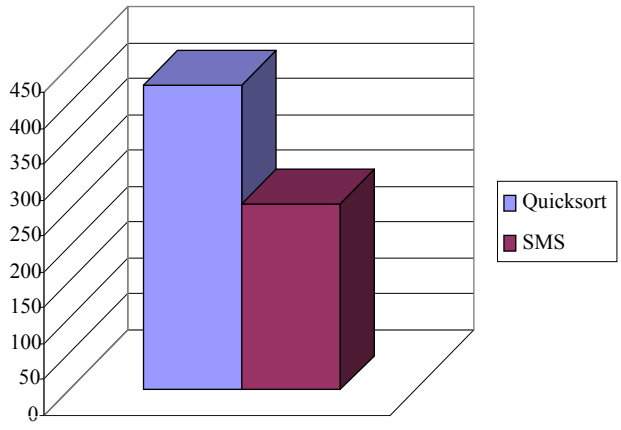

Figure 2. Comparison of sorting techniques.

6. Conclusions

In this paper, a new sorting algorithm is presented called SMS (Scan, Move, and Sort). The SMS algorithm considered as an enhancement on the Quicksort algorithm. The SMS algorithm enhanced the way that Quicksort algorithm divides the input array. Quicksort moves the *pivot* to reside in its

correct place and then divides the array into two parts, and recursively it makes the same procedures for both parts, until it reaches the base case. Instead, the SMS algorithm divided the original array into three parts (arrays), positive, negative, and frequent elements, and moved each element into its correct place in a single pass.

Quicksort takes O(*nlgn*) time in the best case while SMS algorithm takes O(*n*) time complexity. Also, in the average case, the Quicksort algorithm takes O(*nlgn*) time but the SMS algorithm takes O($n+f*(max+|min|)$) time, where *f* is the number of frequent elements. The enhancement on the average case occurs when *n* is much greater than *max* and |*min*|, where the time complexity be near to O(*n*). Also, when dealt with an array of distinct elements, SMS algorithm was more efficient and faster than Quicksort.

Also, in the worst case, Quicksort algorithm takes $O(n^2)$ time while the SMS algorithm takes O($n+f*(max+|min|)$).

We may say that the SMS algorithm is faster than Quicksort algorithm when dealing with a large size (*n*) of the input array with small *max* and *min* values of that array, especially, if the elements are distinct.

Acknowledgements

I would like to thank Dr. Jehad Alnihoud for providing support and material related to the area of this research, and for his suggestions and helpful comments on the manuscript.

References

[1] Aho A., Hopcroft J., and Ullman J., *The Design and Analysis of Computer Algorithms*, Addison Wesley, 1974.

[2] Bell D., "The Principles of Sorting," *The Computer Journal*, vol. 1, no. 2, pp. 71-77, 1958.

[3] Box R. and Lacey S., "A Fast Easy Sort," *Computer Journal of Byte Magazine*, vol. 16, no. 4, pp. 315-321, 1991.

[4] Cormen T., Leiserson C., Rivest R., and Stein C., *Introduction to Algorithms*, McGraw Hill, 2001.

[5] Dean C., "A Simple Expected Running Time Analysis for Randomized Divide and Conquer Algorithms," *Computer Journal of Discrete Applied Mathematics*, vol. 154, no. 1, pp. 1-5, 2006.

[6] Deitel H. and Deitel P., *C++ How to Program*, Prentice Hall, 2001.

[7] Friend E., "Sorting on Electronic Computer Systems," *Computer Journal of ACM*, vol. 3, no. 3, pp. 134-168, 1956.

[8] Hoare R., "Quicksort," *The Computer Journal*, vol. 5, no. 1, pp. 10-15, 1962.

[9] Knuth E., *The Art of Computer Programming Sorting and Searching*, Addison Wesley, 1998.

[10] Kruse R. and Ryba A., *Data Structures and Program Design in C++*, Prentice Hall, 1999.

[11] Ledley R., *Programming and Utilizing Digital Computers*, McGraw Hill, 1962.

[12] Levitin A., *Introduction to the Design and Analysis of Algorithms*, Addison Wesley, 2007.

[13] Moller F., *Analysis of Quicksort*, McGraw Hill, 2001.

[14] Nyhoff L., *An Introduction to Data Structures*, McGraw Hill, 1987.

[15] Thorup M., "Randomized Sorting in O(n log log n) Time and Linear Space Using Addition Shift, and Bit Wise Boolean Operations," *Computer Journal of Algorithms*, vol. 42, no. 2, pp. 205-230, 2002.

[16] Weiss M., *Data Structures and Problem Solving Using Java*, Addison Wesley, 2002.

Rami Hasan Mansi obtained his BSc in Information Technology with a major in Software Engineering from Philadelphia University in 2006 and his MSc in Computer Science from Al al-Bayt University in 2009. His research interests include Design and Analysis of Algorithms, String Matching, Sorting Algorithms, Bioinformatics and Optimization Algorithms.