Available Online at www.ijcsmc.com

International Journal of Computer Science and Mobile Computing

A Monthly Journal of Computer Science and Information Technology

ISSN 2320–088X

IJCSMC, Vol. 4, Issue. 4, April 2015, pg.443 – 450

RESEARCH ARTICLE

Design and Analysis of Optimized Selection Sort Algorithm

Kirti Kaushik*
Roll No.15903, CS, Department of Computer science, Dronacharya College of Engineering, Gurgaon-123506, India
Email: kaushikkirti44@gmail.com

Jyoti Yadav
Roll No. 15040, CS, Department of Applied Computer science, Dronacharya College of Engineering, Gurgaon-123506, India
Email: riya.ana90@gmail.com

Kriti Bhatia
Roll No. 15048, CS, Department of Applied Computer science, Dronacharya College of Engineering, Gurgaon-123506, India
Email: kritibhatia02@gmail.com

ABSTRACT: One of the most frequent operations performed on database is searching. To perform this operation we have different kinds of searching algorithms, some of which are Binary Search, Index Sequential Access Method (ISAM), but these and all other searching algorithms work only on data, which are previously sorted. An efficient algorithm is required in order to make the searching algorithm fast and efficient. This research paper presents a new sorting algorithm named as "Optimized Selection Sort Algorithm, OSSA".OSSA is designed to perform sorting quickly and more effectively as compared to the existing version of selection sort. The introduction of OSSA version of selection sort algorithm for sorting the data stored in database instead of existing selection sort algorithm will provide an opportunity to the users to save almost 50% of their operation time with almost 100% accuracy.

INTRODUCTION

One of the basic problems of computer science is ordering a list of items. There are a number of solutions to this problem, known as sorting algorithms. Some sorting algorithms are simple and spontaneous, such as the bubble sort. Others, such as the quick sort are enormously complex, but produce super-fast results.
There are several elementary and advance sorting algorithms. All sorting algorithm are problem specific meaning they work well on some specific problem and do not work well for all the problems. All sorting algorithm are, therefore, appropriate for specific kinds of problems. Some sorting algorithm work on less number of elements, some are suitable for floating point numbers, some are good for specific range, some sorting algorithms are used for huge number of data, and some are used if the list has repeated values. We sort

data either in statistical order or lexicographical, sorting numerical value either in increasing order or decreasing order and alphabetical value like addressee key.

The common sorting algorithms can be divided into two classes by the difficulty of their algorithms. There is a direct correlation between the complexity of an algorithm and its relative effectiveness.

The complexity of algorithmic is generally written in a form known as Big – O (n) notation, where the O represents the complexity of the algorithm and a value *n* represents the size of the set the algorithm is run against. The two groups of sorting algorithms are O ( ), which includes the bubble, insertion, selection, and shell sorts; and O (*n* log n) which includes the heap, merge, and quick sort.

Since the advancement in computing, much of the research is done to solve the sorting problem, perhaps due to the complexity of solving it efficiently despite its simple, familiar statement. It is always very difficult to say that one sorting algorithm is better than another. Performance of various sorting algorithms depend upon the data being sorted. Sorting is used in many important applications and there have been a plenty of performance analyses.

However, earlier research is based on the algorithm's theoretical complexity or their non-cached architecture. As almost all computers now a day's contain cache, it is important to analyse them based on their cache performance. Quick sort was considered to be a good sorting algorithm in terms of average theoretical complexity and cache performance.

Sorting is one of the most significant and well-studied subject area in computer science. Most of the first-class algorithms are known which offer various trade-offs in efficiency, simplicity, memory use, and other factors. However, these algorithms do not take into account features of modern computer architectures that significantly influence performance. A large number of sorting algorithms have been proposed and their asymptotic complexity, in terms of the number of comparisons or number of iterations, has been carefully analysed.

In the recent past, there has been a growing interest on enhancements to sorting algorithms that do not have an effect on their asymptotic complexity but rather tend to improve performance by enhancing data locality.

Sorting is an essential task that is performed by most computers. It is used commonly in a large variety of important applications. Database applications used by universities, banks, and other institutions all contain sorting code. Due to the importance of sorting in these applications, quite a large number of sorting algorithms have been developed over the decades with varying complexity. Some of the time consuming sorting methods, for example bubble sort,insertion sort, and selection sort have a hypothetical complexity of O( ). Although these algorithms are very slow for sorting larger amount of data, yet these algorithms are simple, so they are not useless. If an application only needs to sort smaller amount of data, then it is suitable to use one of the simple slow sorting algorithms as opposed to a faster, but more complicated sorting algorithm.

ANALYSIS OF OLD SELECTION SORT ALGORITHM

A. Selection Sort

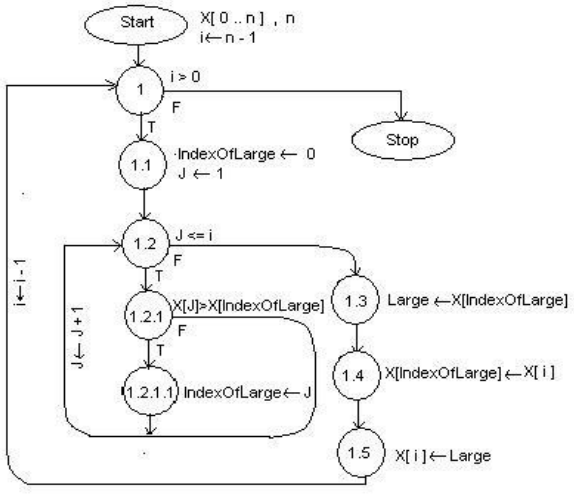
This is a very easy sorting algorithm to understand and is very useful when dealing with small amounts of data. However, as with Bubble sorting, a lot of data really slows it down. Selection sort does have one advantage over other sort techniques. Although it does many comparisons, it does the least amount of data moving. Thus, if your data has small keys but large data area, then selection sorting may be the quickest.

B. Pseudo Code of Old Selection Sort

```
Algorithm SelectionSort (X, n) =>X[0..n-1]
1.      for i ← n – 1 to 0
1.1.    IndexOfLarge ← 0
1.2.    for j←1 to i
1.2.1   if (X[j]>X[IndexOfLarge))
1.2.1.1         indexOfLarge ← j
```


1.3. Large ←X[IndexOfLarge]
1.4. X[IndexOfLarge] ← X[i]
1.5 X[i] ← Large

C. Execution Flow Graph of Old Selection Sort



D. Execution Time of Individual Statements

Pseudo code Instruction	Instruc. Exe Time	How Many times the instruction is executed by CPU
1. for i← n – 1 to 0	C_1	n
1.1. IndexOfLarge ← 0	C_2	n – 1
1.2. for j←1 to i	C_3	$\sum_{i=1}^{n-1}(i+1)$
1.2.1 if (X[j]>X[IndexOfLarge])	C_4	$\sum_{i=1}^{n-1} i$
1.2.1.1 indexOfLarge ← j	C_5	$\sum_{i=1}^{n-1} i(t_{i,j})$
1.3. Large ←X[IndexOfLarge]	C_6	n – 1
1.4. X[IndexOfLarge] ← X[i]	C_7	n – 1
1.5 X[i] ← Large	C_8	n – 1

In order to evaluate the execution time complexity of the given data of n elements. First we simplify the execution time of some inner loop statements in above algorithm.
Note that
$\sum t_{i,j} = 1$ when the if statement is true, 0 otherwise

$$C3 \sum_{i=1}^{n-1} (i + 1) = C3 \sum_{i=1}^{n-1} i + C3 \sum_{i=1}^{n-1} 1$$
$$= C3\, n(n-1)/2 + C3\,(n-1)$$
$$\sum_{i=1}^{n-1} C4 \sum i = C4\, n\,(n-1)/2$$
$$\sum_{i=1}^{n-1} C5 \sum i = C5\, n\,(n-1)/2, \text{ when } t_{i,j} = 1 \text{ \& 0 otherwise.}$$

i=1

E. Best-Case Time Complexity of Old Selection Sort.

Then for the best-case scenario we have that all ti,j = 0 so we get

T(n) =C1 n + C2 n - C2 + C3 n (n – 1)/2 + C3 (n – 1)+C4 n (n – 1)/2+ C6n – C6 + C7n - C7 + C8 n - C8

T(n) =C1 n+ C2 n - C2 +C3 n2 /2 – C3 n/2 + C3 n – C3 + C4 n2/2– C4n/2 + C6 n – C6 + C7 n - C7 +C8 n - C8

T(n) = n2 (C3 / 2 + C4 /2) + n (C1 + C2 + C3 – C3 /2 - C4 /2+ C6 + C7 + C8) – (C2 + C3 +C7 +C8)

Let

a = (C3 / 2 + C4 /2) ,

b = (C1+C2+C3– C3/2 – C4/2+C6+ C7 + C8)&

c = – (C2 + C3 + C7 +C8)

Then T(n) becomes

T(n) = a n2 + bn + c

Thus here in best-case, the complexity of execution time of an algorithm shows the lower bound and is asymptotically denoted with Ω. Therefore by ignoring the constant a, b,c and the lower terms of n, and taking only the dominant term i.e. n2, then the asymptotic running time of selection sort will be Ω(n2) and will lie in of set of asymptotic function i.e Θ(n2). Hence we can say that the asymptotic running time of old SS will be:

T(n) = Θ(n2)

F. Worst - Case Time Complexity.

Now for the worst-case scenario we have that all ti,j = 1 so we have

T(n)=C1 n + C2 n - C2+ C3 n(n–1)/2 + C3 (n–1) +C4n (n–1)/2+C5n(n – 1)/2 + C6 n – C6+ C7 n - C7 + C8n - C8

T(n)=C1 n + C2 n - C2+ C3 n2 /2 – C3 n/2 + C3 n – C3 + C4 n2 /2– C4n/2 + C5 n2 /2– C5 n/2 +C6n – C6+C7n-C7 + C8 n - C8

T(n)= n2 (C3 / 2 + C4 /2+ C5 / 2) + n (C1 + C2 + C3 – C3 /2 – C4 /2 - C5 /2+C6 + C7 + C8) – (C2 + C3 + C6 + C7 +C8)

Let

a = (C3 / 2 + C4 / 2 + C6 / 2)

b = (C1 + C2 + C3– C3 /2- C4 /2 - C5 /2+ C6 + C7 + C8) and

c = – (C2 + C3 + C6 + C7 +C8)

Then T(n) becomes

T(n) = a n2 + bn + c

Thus here in worst-case, the complexity of execution time of an algorithm shows the upper bound and is asymptotically denoted with Big-O. Therefore by ignoring the constant a, b,c and the lower terms of n, and taking only the dominant term i.e. n2 , then the asymptotic running time of selection sort will be of the order of O(n2) and will lie in of set of asymptotic function i.e Θ(n2). Hence we can say that the asymptotic running time of old SS will be:

T(n) = Θ(n2)

It means that the best and worst case asymptotic running time of selection sort is same i.eT(n)=Θ(n2), however there may be little difference in actual running time, which will be very less and hence ignored.

If we assume the Individual instruction cost that is :

C1= C2= C3= C4 = C5 = C6 = C7 = C8 = 1 then the value of constants a,b,c in the equations for worst – case scenario:

a = (1 / 2 + 1 / 2 + 1 / 2) = 3/2

b =(1 + 1 + 1 - 1 /2 - 1/2 - 1 /2 + 1 + 1 + 1)

= 6-3/2 = 9/2

c = – (1 + 1 + 1 + 1 + 1) = -5

by putting the value in equation given above we get the approximate actual running time of selection sort in the form of single dependent variable n, which will be:

T(n) = 3/2 n2 + 9/2n - 5 ------ (2.1)

Here if we put n = 100, i.e for 100 data elements the time taken by old selection sort will be:
T(100) = 3/2(100)2 + 9/2(100) – 5
= 15000 + 450 – 5
T (100) = 15445

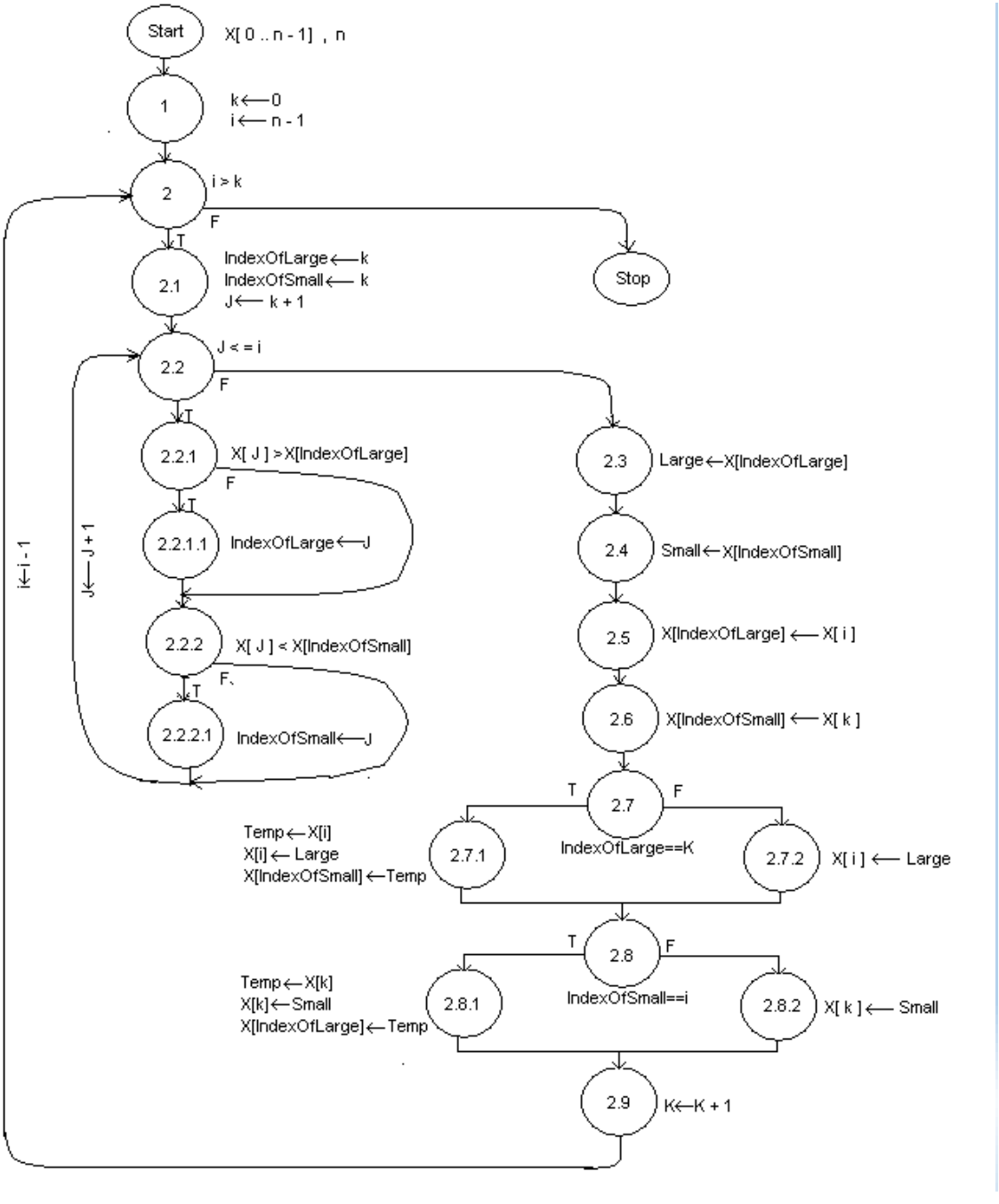
ANALYSIS OF OPTIMEZED SELECTION SORT ALGORITHM (OSSA)

As the selection sort has three main characteristic, the first that it is In-place algorithm and second it is a stable algorithm, and third it is simple while all other algorithm of order O (n2) do not have all these characteristics. So it is felt that why its time complexity should not improve. In this scenario, the author capture the idea from old selection sort that if we sort two data elements (smallest as well as largest) of the given data in single iteration, then its time can be reduced.
Therefore our OSSA sorts the data from front and rear ends of the array and finishes the execution of outer loop when it reaches at the middle of the array. In its first iteration it finds the smallest and largest data elements of array and place those in their desired locations, then it finds the next smallest and largest data elements from the remaining array and sorts those in their next respective locations in the array. In this way it executes half the iteration of the outer loop, while old SS only finds either smallest or largest (but not both) element of array and requires the full iteration of outer loop.
Although OSSA is still O(n2), but In this way its performance level has very much improved as compared to other sorting algorithm of said order i.e Bubble sort, insertion sort etc.

A. Pseudo Code of OSSA

Algorithm OSSA (X, n) □ X[0 .. n-1]
1. k ← 0
2. for i ← n – 1 to k
2.1. IndexOfLarge←IndexOfSmall←k
2.2. for j←k + 1 to i
2.2.1. if (X[j] > X(IndexOfLarge])
2.2.1.1 IndexOfLarge ← j
2.2.2. if (X[j] < X[IndexOfSmall])
2.2.2.1 IndexOfSmall← j
2.3 Large ← X[IndexOfLarge]
2.4 Small ← X[IndexOfSmall]
2.5. X[IndexOfLarge] ← X[i]
2.6. X[IndexOfSmall] ← X[k]
2.7 if (IndexOfLarg==k)
2.7.1 Temp ← X[i]
X[i] ← Large
X[IndexOfSmall] ← Temp
2.7.2 Else X[i] ← Large
2.8 if (IndexOfLarg==k)
2.8.1 Temp ← X[k]
X[k] ← Small
X[IndexOfLarge] ← Temp
2.8.2 Else X[k] ← Small
2.9 k ← k + 1

B. Execution Flow Graph of OSSA



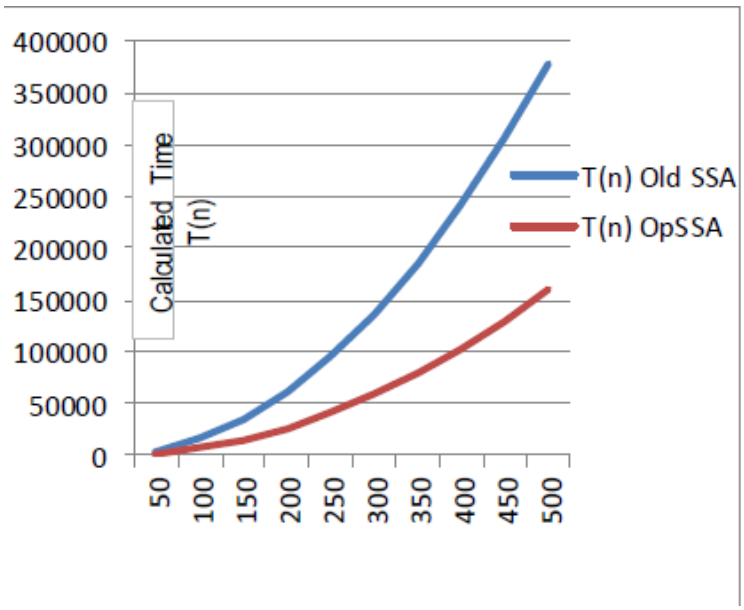
C. Execution Time of Individual Statements

Pseudo Code Instructions	Intruc. Exe Time	How many times the Instruction is executed by CPU
AVSS (X, n) ▷ X[0 .. n-1]		
1. k ←0	C_1	1
2. for i ← n – 1 to k	C_2	(n/2) + 1
2.1. IndexOfLarge←IndexOfSmall←k	C_3	n/2
2.2. for j←k + 1 to i	C_4	$\sum_{i=1}^{n/2} (i+1)$
2.2.1. if (X[j] > X[IndexOfLarge])	C_5	$\sum_{i=1}^{n/2} i$
2.2.1.1 IndexOfLarge ← j	C_6	$\sum_{i=1}^{n/2} i\ (t_{i,j})$
2.2.2. if (X[j] < X[IndexOfSmall])	C_7	$\sum_{i=1}^{n/2} i$
2.2.2.1 IndexOfSmall ← j	C_8	$\sum_{i=1}^{n/2} i\ (t_{i,j})$
2.3 Large ← X[IndexOfLarge]	C_9	n/2
2.4 Small ← X[IndexOfSmall]	C_{10}	n/2
2.5. X[IndexOfLarge] ← X[i]	C_{11}	n/2
2.6. X[IndexOfSmall] ← X[k]	C_{12}	n/2
2.7 if (IndexOfLarge==k)	C_{13}	n/2
2.7.1 Temp ← X[i] X[i] ← Large X[IndexOfSmall] ← Temp		
2.7.2 Else X[i] ← Large		
2.8 if (IndexOfSmall==i)	C_{14}	n/2
2.8.1 Temp ←X[k] X[k] ← Small X[IndexOfLarge] ← Temp		
2.8.2 Else X[k] ← Small		
2.9 k ← k + 1	C_{15}	n/2

COMPARISON OF OLD SELECTION SORT AND OPTIMIZED SELECTION SORT ALGORITHM

Below is the table representing the calculated time, when multiple values of n are used. n	**Calculated Time of Old SSA**	**Calculated Time of Optimized SSA**	**Percentage Improvement**
n	T(n) Old SS	T(n) OSSA	
50	3970	1877	52.7 %
100	15445	6877	55.5 %
150	34420	15002	56.4 %
200	60895	26252	56.9 %
250	94870	40627	57.2 %
300	136345	58127	57.4 %
350	185320	78752	57.5 %
400	241795	102502	57.6 %
450	305770	129377	57.7 %
500	377245	159377	57.8 %

Below is the line graph plotted in response of data given in above table.



CONCLUSION

Logic of OSSA based on the old selection sort algorithm. The main difference in old SS and our OSSA is that the former sorts the data from one end i.e from largest element of array to smallest element or from smallest to largest but the later starts sorting from both ends and finds the largest and smallest data elements of array in single iteration and places those at their proper locations then during second iteration it sorts the second largest and second smallest elements from the remaining array data and places those in their appropriate locations in the array. Similarly, it sorts rest of the data elements and puts those in their proper positions. OSSA sorts the data in half iterations as compared to old SS. The improvement is also of the order .

