The International Journal Of Engineering And Science (IJES)
|| Volume || 5 || Issue || 5 || Pages || PP -125-130 || 2016 ||
ISSN (e): 2319 – 1813 ISSN (p): 2319 – 1805

# Selection Sort with Improved Asymptotic Time Bounds

Mirza Abdulla
*College Of Computer Studies,*
*AMA International University*

**----------ABSTRACT----------**

*Sorting and searching are the most fundamental problems in computer science. Sorting is used for most of the times to help in searching. One of the most well known sorting algorithms that are taught at introductory computer science courses is the classical selection sort. While such an algorithms is easy to explain and grasp at the introductory computer science level, it is far from being an efficient sorting technique, since it requires* $\boldsymbol{O(n^2)}$ *time to sort a list of n numbers. It does so by repeatedly finding the minimum. In this paper we explore the benefit of reducing the search time for the minimum on each pass of the algorithm, and show that we can obtain a worst case time bound of* $\boldsymbol{O(n^2\sqrt[2]{n})}$ *by making only minor modifications to the input list. Thus our bound is a factor* $\boldsymbol{O(\sqrt[2]{n})}$ *of faster than the classical selections sort and other classical sorts such as insertion and bubble sort.*

***Keywords*** *– Selection sort, complexity analysis, time bounds, O notation.*

Date of Submission: 11 May 2016 Date of Accepted: 31 May 2016

## I. INTRODUCTION

Ever since the first computer was built the use of computers has multiplied many times and respectively the size of data handled by these computers increased even more. In their manipulation of data stored in these computer programmers had to find the most efficient way to retrieve these data. Sorting plays a major role in this area. Few decades ago, sorting amounted to about half the time spent on data manipulations in commercial applications. Indeed, sorting and searching are fundamental problems in computer science, and nowadays there are thousands of sorting techniques and variations of these techniques in various guises.

The two main ordering of data is in ascending and descending order, but in doing so there may be other criteria of importance [1], in addition to the running time efficiency. Such as:

- Stability of the sort techniques: A sorting technique is stable if given a list of values, say $a_1, a_2, a_3, \ldots, a_n$ with say $a_i = a_j$ and $i < j$, then in the sorted list $a_i$ would appear before $a_j$.
- Sorting is performed in situ or an extra array is used.
- Sorting is comparison based or not.
- Space efficiency of the sort technique.
- Best, Average, and Worst case time complexity of the algorithm used.
- Amount of data movement, and data comparisons.

Some of the oldest techniques for sorting and still a favorable subject to teach an introductory computing or programming courses are the Selection, sort, Bubble sort, and Insertion sort. These sort techniques share between them simplicity of implementation, sorting in place without using more than $O(1)$ extra memory space, and gross inefficiency in the running time on large sets of data ( $O(n^2)$ time in the worst case.)

In bubble sort [2] adjacent elements are compared and swapped if they violate the required ordering. The process is continued until no more swaps become necessary and the data is sorted. Insertion [3, 4] and selection sorts [5] rely on the idea of sorted and unsorted lists. In insertion sort we start with the sorted list containing only the first item in the list and the others are in the unsorted list. The items in the unsorted list are inspected one by one and inserted into the right position in the sorted list. This process is continued until all items have been sorted. In selection sort the sorted list is initially empty. We select the minimum of the unsorted list and place it as the next item in the sorted list. Again, this process is continued until all data have been sorted.

| **Sort Properties** | | |
|---|---|---|
| **selection** | **insertion** | **bubble** |
| Not stable | Stable | Stable |
| O(1) extra space | O(1) extra space | O(1) extra space |
| $\Theta(n^2)$ comparisons | $O(n^2)$ comparisons | $O(n^2)$ comparisons |
| $\Theta(n)$ swaps | $O(n^2)$ swaps | $O(n^2)$ swaps |
| Not adaptive | Adaptive: O(n) time when nearly sorted | Adaptive: O(n) when nearly sorted |

Therefore, in general these techniques work by performing about n-1 passes in the worst case and in each pass the largest or smallest element in the list is found and placed in the right position. In particular, for selection sort the process of finding the minimum of the unsorted list keeps inspecting "almost" the same elements that we inspected in a previous pass to find the largest or the smallest of these. Such time inefficiency can be reduced by restricting the search in each pass to relatively small number of potential minimum/maximum elements for that pass, depending on the required sort order. Moreover, the improved Selection sort can sort the elements of the list in place without the need to use another array as may occur in sorting techniques that use recursion.

In this paper, we present an improved worst case running time selection algorithm. We prove that it is substantially better and with $O(\sqrt{n})$ factor of time improvement than the classical Selection sort technique. Section 2 gives a brief background of the classical Selection sort and time analysis. It also presents the concept of Improved Selection Sort (ISS) and the pseudo code as well as the theoretical time analysis of the algorithm. Section 3 gives a comparison of time in the implementation of the classical Bubble, Insertion, and Selection sort and the Improved Selection Sort algorithm.

## II. CLASSICAL SELECTION SORT.

As mentioned earlier Selection Sort operates by having two lists one is the sorted list which is initially empty and the other is the unsorted list which initially contains the input list of items. Selection sort performs a maximum of n-1 passes over the unsorted list of n items, each time finding the minimum and appending it to the sorted list. Selecting the minimum of the unsorted list of items and places it at the end of a sorted list of items.

| | | |
|---|---|---|
| | selection_sort(int a[], int n) { | |
| | int aSize, minpos, i, tmp; | |
| 1 | for (aSize = 0; aSize < n; aSize ++) { | *n+1* |
| 2 | minpos=aSize | *n* |
| 3 | for (i = 1 + aSize; i < n; i++){ | $\frac{n(n+1)}{2}$ |
| 4 | minpos = a[i] < a[minpos] ? i : minpos; | $\frac{n(n-1)}{2}$ |
| 5 | tmp = a[minpos]; | *n* |
| 6 | a[minpos] = a[aSize - 1]; | *n* |
| 7 | a[aSize - 1] = tmp; | *n* |
| | } } | |

**Time Analysis of the Classical Algorithm.**

The for loop in step 1 in the algorithm is used to build the sorted list where in the body of the loop the minimum in the unsorted list is found and is placed at the end of the sorted list. This loop will be repeated *n* times (actually one more to exit the loop).

The for-loop in step 3 will be repeated on the unsorted list to find the minimum item in the list. The size of the list will decrease each time we find the min as it will be appended to the sorted list. Therefore, the number of times this loop is repeated is:

$[1 + (n-1)] + [1 + (n-2)] + [1 + (n-3)] + [1 + (n-4)] + [1 + (n-5)] \ldots . + [1 + (n-n)] = \sum_{j=1}^{n}(1 + (n-j)) = \frac{n(n+1)}{2}$ (Note: the extra 1 in the summation is to account for the iteration that breaks the loop.

Step 4 is the body of the inner for loop and is thus executed $\sum_{j=1}^{n}(n-j) = \frac{n(n-1)}{2}$ times.

Steps 5, 6, and 7 are used to swap the minimum found by the inner for loop with the element in the current position. These statements will be executed once in each iteration of the outer for loop and thus will be executed *n* times each.

Therefore, if we assume each instruction requires a constant time to execute, we see that the overall running time of the algorithm is $< c(1 + 5n + n^2)$ for some constant $c > 0$. It follows therefore that the running time of the algorithm is $\theta(n^2)$.

## III. IMPROVED SELECTION SORT

While the original Selection sort requires $O(n^2)$ operations in the best-average-worst case, we can improve this bound substantially by the treating the array of $n$ elements to be composed of $\sqrt[2]{n}$ consecutive blocks each of size $\sqrt[2]{n}$ elements. We shall refer to these blocks as level 0 or $L_0$ blocks. More formally:

**Definition**: given an array, A[1..*n*], holding *n* elements,

1. We shall refer to all consecutive locations in the array starting from location $1 + i * n^{1/2}$ to $(i + 1) * n^{1/2}$, for $0 \leq i < n^{1/2}$ as the ***i***th level 0 (or $L_0$) block.
2. The first element after all the elements of the sorted lists in the selection sort algorithm, or the leftmost element of the unsorted list is referred to as the ***current element*** and the $L_0$ block to which it belongs is called the ***current block***.
3. All elements in the current $L_0$ block but not yet in the sorted list are referred to as ***right elements*** to the current element.
4. All blocks in the unsorted list but not the current are referred to as the ***right blocks*** to the current block.

In selection sort we have two partitions: one sorted and the other unsorted. The use of blocks can significantly improve the time for the selection of the minimum element of the unsorted list during each iteration of Selection sort algorithm. The method we use to achieve this improvement requires that *we always keep the minimum of a block in the first location of that block.* During each iteration we only search thru the current and right items of the current block and the first element of each right block. We record the location of the minimum element and its block number. After the iteration is completed we swap the minimum item with the current item. We also find the new minimum of the block where the iteration minimum was found and swap it with the first item in that block. Such an action guarantees that all right blocks contain the minimum of their elements in the first position prior to the start of each iteration of the improved selection sort algorithm.

**Algorithm:**

Selection Sort (a[1..n])

Here a is the unsorted input list and n is the size of array. After completion of the algorithm array will become sorted. Variable min keeps the location of the minimum value.

```
0.    for crnt = 1 to no of blocks:  find minimum of each block and interchange with first location of that
block.
                                                  //initialize the blocks.
1.    for crnt= 1 to n-1                          //sort the array starting from location 1.
  {
2.  Set min=a[crnt]                               // find the min of the value at the current location
3.  Repeat for count= 1+crnt to end of current block
      if (a[count]<a[min])                        // and all right elements of the current block
  Set min=count
      End if
4.  Repeat for each right block                   // and all right blocks
5.    loc = first location of block
      If (a[loc]<a[min])
  Set min=loc
      End if
        //at the end of the loop we have the min of unsorted list
                                                  .
6.  locBlk = 1 + min / size of block;             //obtain the block number of location min
7.  Interchange data at location crnt and min.    // the minimum was found and is swapped with the item
8.  min = element location of block locBlk        // in current position.
9.  Repeat for each element in the block whose number is locBlk after min
10.   loc = element location in block             // make sure that the block from which the minimum
      if (a[loc]<a[min])                          // was obtained contains the minimum of the block
     Set min=loc                          // in its first location, by finding the minimum and
      End if                                      // interchanging it with the element in the 1st location
                                                  // of the block.
```

11. *Interchange data at location min and the first location of the block locBlk*
12.**}**

**Brief description of the algorithm**

**Step 0** finds the minimum element in each block and interchanges it with the first element in that block. Therefore, prior to step 1 all the blocks in the array would contain in the first position of the block the minimum of that block.

**Step 1** the main loop of the minimum selection in the selection sort algorithm

**Steps 2** and **3** find the minimum element in the current block. Note that first position of a block contains the minimum element of the block. When any of the other elements in the current block becomes the current element block, it may not have the minimum of the block in that position initially.

**Steps 4** and **5** find the minimum element in the entire unsorted list. The loop only inspects the first element in each block to compare it with the minimum of the unsorted list so far.

**Step 6** finds the number of the block where the minimum of the unsorted list was found. This value can be easily obtained by dividing the location by the size of a block. The quotient is incremented by one to allow the block numbers to start from 1.

**Step 7** now we have the location of the minimum element of the entire unsorted list and therefore, we can swap it with the current element.

**Steps 9** to **11** now that the minimum of the entire unsorted list was swapped with the current element, the block which contained that element before the swap may not have the minimum of the block in the first position. Steps 9 to 11 find the minimum of the block and interchange it with the first element of the block if need be.

**Correctness of the ISS algorithm**

**Proposition 1.** Before the start of each iteration of the main loop, each of the right blocks, if any, contains the minimum of the block in the first position.

**Proof.** As explained earlier step 0 takes care of this proposition prior to the start of the sort process. The proposition is violated only by a swap between the current element and the minimum of the entire unsorted list. However, this possible violation is corrected by steps 9 to 11. ■

**Proposition 2.** At the end of each iteration of the main loop the current position contains the minimum of the unsorted list.

**Proof.** The minimum of the entire unsorted list can only be in the first position in one of the right blocks or one of the right elements in the current block or the current element itself. These are the only elements the algorithm searches through to find the minimum. If we assume that it is another element in the unsorted list then it must in a right block but not in the first position of that right block. However, this would lead to a contradiction as per proposition 1 and the rule of maintaining the minimum element of a block in the first position of that block. ■

**Lemma 1.** The improved Selection Sort algorithm correctly sorts the entire list in ascending order.

**Analysis of algorithm**

Step 0 iterates on all the blocks and for each block it finds the minimum and swaps it with the first element of that block. The overall time is (number of blocks)*(size of each block)= $(\sqrt[2]{n})*(\sqrt[2]{n}) = O(n)$.

Step 3 would iterate for at most the size of a block = $O(\sqrt[2]{n})$

Steps 4 and 5 only search through the first element of the right blocks. As there are at most $\sqrt[2]{n}$ right blocks, it follows, therefore, that these steps require at most $O(\sqrt[2]{n})$ time in the worst case.

Step 6 is a mere swap and is performed in $O(1)$ time.

Steps 8 to 11 search the block where the minimum of entire unsorted list was found. This time is at most $O(\sqrt[2]{n})$.

It follows therefore, that each iteration of the main loop in the algorithm requires at most $O(\sqrt[2]{n})$ time. Repeating the loop for *n* times gives us an upper bound of $O(n\sqrt[2]{n})$ operations.

Thus we have.

**Theorem 1.** The Improved Selection sort algorithm runs in $O(n\sqrt[2]{n})$ time in the worst case.

## IV. RESULTS AND DISCUSSIONS

The classical selection sort together with insertion and bubble sort were implemented and their performance on the input was compared to obtain a "feel" of the time improvement of the modifications made in

the Improved Selection Sort that was the subject of this paper. The algorithms were implemented in java. Prior to the start of the sorting process the time was recorded, and similarly at the end of the process the time was recorded. The difference of the two times was recorded for each of these sorting techniques.

Two types of inputs were used. The first was random numbers generated and the second was by inputting numbers in reverse order to the required sort ordering, which should result in the worst case performance for all the aforementioned sort techniques. The sample size was ranged from 5000 to 30000 in steps of 5000. However, we expected the system's time not to be highly accurate and therefore, an average of the five executions was taken for each result.
The results are as follows:

a) Random Data

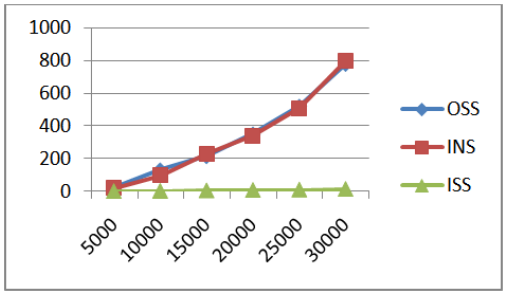
| ***n*** | **OSS** | **INS** | **ISS** | **BBL** |
|---|---|---|---|---|
| 5000 | 22 | 20 | 1 | 64 |
| 10000 | 130 | 100 | 2 | 350 |
| 15000 | 215 | 230 | 5 | 600 |
| 20000 | 350 | 340 | 8 | 960 |
| 25000 | 520 | 510 | 10 | 1420 |
| 30000 | 780 | 800 | 14 | 2000 |

b) Data in reverse order (worst case)

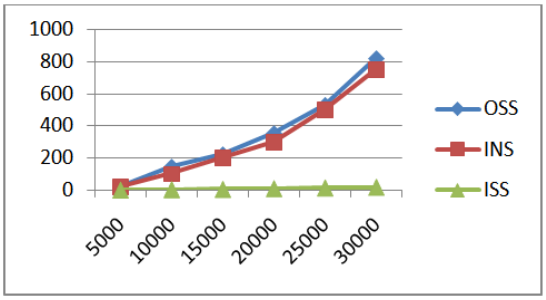
| ***n*** | **OSS** | **INS** | **ISS** |
|---|---|---|---|
| 5000 | 22 | 20 | 1 |
| 10000 | 146 | 100 | 4 |
| 15000 | 223 | 200 | 5.3 |
| 20000 | 354 | 300 | 10 |
| 25000 | 530 | 500 | 14 |
| 30000 | 820 | 750 | 18 |

The time is in milliseconds, and *n* is the size of the input data. It was obvious from the performance of the sorting algorithms that Bubble sort had much worse performance time than the other methods for the randomly generated data. It was therefore, not considered in the comparisons of the sorting techniques in the worst case scenario.

It is obvious from the results that the improved selection sort is superior to the classical selection, insertion, or bubble sort, supporting the theoretical results obtained earlier.




Performance of the classical selection sort (OSS), insertion sort (INS), and the improved selection sort (ISS) is exhibited in the above diagram for data obtained using random number generator. The x-axis gives the size of the data sorted and the y-axis gives the time in milliseconds.




The second graph displays the performance of the sort algorithms in the worst case scenario.

## V. CONCLUSION

An improved selection sort algorithm was presented and its performance was proved to be $\theta(n\sqrt[2]{n})$. This is $O(\sqrt[2]{n})$ factor of improvement over the classical selection sort, insertion sort and bubble sort in the worst case. A small sample of data was also obtained and the performance of these algorithms was analyzed. The results supported the theoretical findings.

## REFERENCES


[1]. D. Knuth, "The Art of Computer programming Sorting and Searching", 2nd edition, Addison-Wesley, vol. 3, (1998).

[2]. Astrachan, O. (2003, February). Duke University Computer Science Department. Retrieved Octorber 2, 2007, from Bubble Sort: An Archaeological Algorithmic Analysis: http://www.cs.duke.edu/~ola/papers/bubble.pdf

[3]. T. H. Coreman, C. E. Leierson, R. L. Rivest and C. Stein, Introduction to Algorithms, 2nd edition, MIT Press.

[4]. Anon., 1891, Untitled reports on Hollerith sorter, J. American Stat. Assoc. 2, 330-341.

[5]. J. L. Bentley and R. Sedgewick, 1997, "Fast Algorithms for Sorting and Searching Strings", ACM-SIAM SODA ″97, 360-369.

[6]. Folk, Michael J.; Zoellick, Bill (1992), File Structures (2nd ed.), Addison-Wesley, ISBN 0-201-55713-4

[7]. J. Alnihoud and R. Mansi, "An Enhancement of Major Sorting Algorithms", International Arab Journal of Information Technology, vol. 7, no. 1, (2010), pp. 55-62.

[8]. Flores I. ,Oct 1960, "Analysis of Internal Computer Sorting". J.ACM 7,4, 389- 409.

[9]. E. Kapur, P. Kumar and S. Gupta, "Proposal of a two way sorting algorithm and performance comparison with existing algorithms", International Journal of Computer Science, Engineering and Applications (IJCSEA), vol. 2, no. 3, (2012), pp. 61-78.

[10]. M. Mirza, "Data Structures and Algorithms for Hierarchical Memory Machines", Ph.D. dissertation, Courant Institute of Mathematical Sciences, New York University, 1990.

[11]. Mirza Abdulla, Marrying Inefficient Sorting Techniques Can Give Birth to a Substantially More Efficient Algorithm ,International Journal of Computer Science and Mobile Applications, Vol.3 Issue. 12, December- 2015, pg. 15-21.

[12]. Mirza Abdulla, Optimal Time and Space Sorting Algorithm on Hierarchical Memory with Block Transfer, International Journal of Electrical & Computer Sciences IJECS-IJENS Vol:16 No:01, 164701-9191-IJECS-IJENS © February 2016 IJEN

[13]. V.Estivill-Castro and D.Wood, 1992,"A Survey of Adaptive Sorting Algorithms", Computing Surveys, 24:441-476.

[14]. S. Jadoon, S. F. Solehria, S. Rehman and H. Jan, "Design and Analysis of Optimized Selection Sort Algorithm", International Journal of Electric & Computer Sciences (IJECS-IJENS), vol. 11, no. 01, pp. 16-22.

[15]. S. Chand, T. Chaudhary and R. Parveen, 2011, "Upgraded Selection Sort", International Journal on Computer Science and Engineering (IJCSE), ISSN: 0975-3397, vol. 3, no. 4, pp. 1633-1637.

[16]. A. D. Mishra and D. Garg, "Selection of the best sorting algorithm", International Journal of Intelligent Information Processing, vol. 2, no. 2, (2008) July-December, pp. 363-368.

[17]. E. Horowitz, S. Sahni and S. Rajasekaran, Computer Algorithms, Galgotia Publications.

[18]. S. Lipschutz, "Theory and Problems of Data Structure", McGraw Hill Book Company.

[19]. Shaw M., 2003, Writing good software engineering research papers, In Proceedings of 25th International Conference on Software Engineering, pp.726-736.

[20]. Aho, A. V.; Hopcroft, J. E.; and Ullmann, J. D. "Data Structures and Algorithms". Reading, MA: Addison-Wesley, pp. 369-374, 1987.