An In-Depth Analysis of Selection Sort & Insertion Sort

Md. Barkullah
Dept. of CSE
RUET
2003071@student.ruet.ac.bd

Partha Paul
Dept. of CSE
RUET
2003078@student.ruet.ac.bd

Md. Abu Sufyan
Dept. of CSE
RUET
2003085@student.ruet.ac.bd

Md. Tameem Rahman
Dept. of CSE
RUET
2003089@student.ruet.ac.bd

Mir Ashikur Rahman
Dept. of CSE
RUET
2003109@student.ruet.ac.bd

Md. Atik Mouhtasim
Dept. of CSE
RUET
2003118@student.ruet.ac.bd

Abstract

The ordering of numbers in an integer array in computer science and mathematics is one of the most important topics. For this purpose, there are a large variety of sorting algorithms like Selection sort, Insertion sort, Quick sort, Radix sort, Merge sort and Bubble sort. Sorting algorithms play a crucial role in organising data efficiently, and two widely used techniques are Insertion Sort and Selection Sort. In this study, two sorting algorithms are investigated as Selection and Insertion sort. This study compares performance of Selection sort and Insertion sort algorithms that are used commonly in terms of running time.

1 Introduction

Sorting process in data structure is essential for transforming randomly distributed elements into either decreasing or ascending order. Various sorting algorithms, including Selection sort, Insertion sort, Quick sort, and Radix sort, have been developed to reduce complexity and enhance the performance of sorting.

In this study, the focus is on comparing the Selection sort and Insertion sort algorithms in terms of their running time. The following sections provide an in-depth explanation of the Selection sort algorithm, its working logic illustrated with an example, and its implementation in the C++ programming language. Additionally, the running time for sorting using the Selection sort algorithm is presented.**?**

Subsequently, the Insertion sort algo-

rithm is introduced, along with its working logic demonstrated through an example. The code for the Insertion sort algorithm is implemented in the C++ programming language, and the running time for sorting using this algorithm is provided.

Finally, a comparative analysis of the Selection sort and Insertion sort algorithms is presented, focusing on their time complexity.

2 Background Study

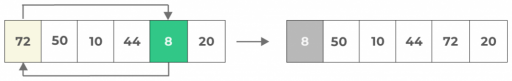
2.1 Selection Sort:

Selection sort is a simple comparison-based sorting algorithm. The algorithm begins by finding the smallest element in the array and then exchanges this smallest element with the element in the first position. After this initial step, the algorithm proceeds to select the smallest element in the unsorted part of the array in each step of the sorting process. It exchanges this selected smallest element with the element in the unsorted part of the current step. This process continues until there are no unsorted elements remaining in the array. Notably, the selection sort algorithm spends a significant amount of time finding the smallest element in the unsorted part of the array.

Work Logic of the Selection Sort Algorithm

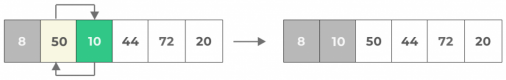
First Pass:

- For the first position in the sorted array, the entire array is traversed sequentially from index 0 to 4. The element at the first position is replaced with the smallest value found during traversal.
- After one iteration, the least value (in this case, 8) appears in the first position of the sorted list and 72 is placed at the position where 8 was originally placed in.



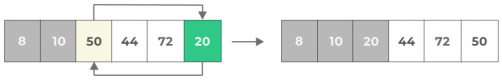
Second Pass:

- For the second position, where 50 is present, the rest of the array is traversed in a sequential manner.
- After traversing, it is determined that 10 is the second lowest value in the array, and it should appear at the second place in the array. Thus, these values are swapped.



Third Pass:

- For the third position, where 50 is present again, the rest of the array is traversed to find the third least value present in the array.
- While traversing, 20 is identified as the third least value, and it should appear at the third place in the array. Thus, a swap is performed between 20 and the element present at the third position.



Fourth Pass:

- Similarly, for the fourth position, the rest of the array is traversed to find the fourth least element in the array.
- As 44 is the fourth lowest value, it is placed at the fourth position.

Fifth Pass:

- At last,for the position where 72 is hold the array is traversed onwards and 50 is found the next least valued number and is swapped with 72.
- As there is onle one element left the resulting array is the sorted array becouse that last element is already at the right position according to its value.

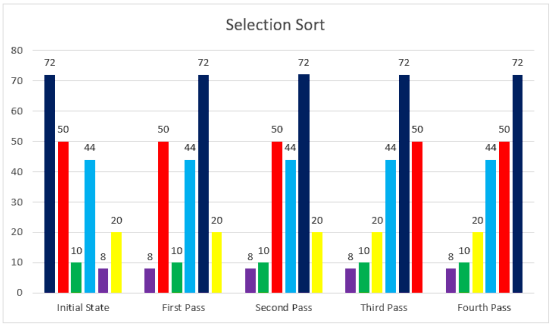


Chart 1 : Selection sort algorithm passes

Algorithm 1 : Selection Sort**?**

```
1: for i ← 1 to n − 1 do
2:     smallest ← i
3:     for k ← i + 1 to n do
4:         if arr[k] < arr[min] then
5:             min ← k
6:         end if
7:     end for
8:     Swap arr[min] and arr[j]
9: end for
```

Loop Invariant: At the start of each iteration of the *for* loop of lines 1 - 9, the subarray $arr[0..i-1]$ is always sorted.**?**

2.2 Insertion Sort:

Insertion sort is a simple comparison-based sorting algorithm. The algorithm starts by comparing the first two elements in the array. If the first element is greater than the second element, they are exchanged. This process is then implemented for all neighboring indexed elements.

Consider an example:

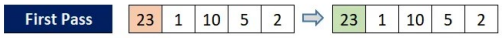
$$\text{arr}[] = \{23, 1, 10, 5, 2\}$$

Suppose n is the number of elements in the array. For example, consider an integer array with 5 elements, so n is 5 for this sample, and sorting of this integer array in ascending order is desired:

Work Logic of the Insertion Sort Algorithm

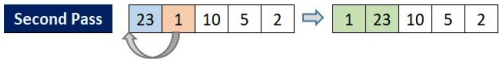
First Pass:

- The first item of the array is always sorted as there is no value to compare it with so relative to itself it is already sorted.



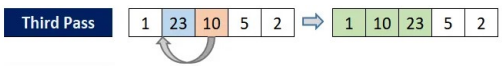
Second Pass:

- Now iterate to the 2nd element and compare with the first element.
- Here, 23 is greater than 1; hence, they are not in ascending order, and 23 is not at its correct position. Thus, swap 1 and 23.
- So, for now, 1 is stored in a sorted sub-array where 23 is in the 2nd position and 1 is at first.



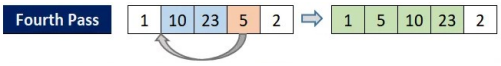
Third Pass:

- Now, move to the next two elements and compare them.
- Here, 10 is compared to 23, and as 23 is greater, a swap operation is performed.
- Then, 10 is compared with 1, and as 1 is smaller, no swapping occurs. Resulting in a partially sorted array.



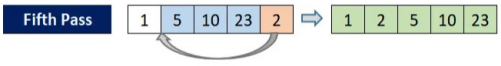
Fourth Pass:

- Now, comparison starts with 5 and 23, which results in swapping as 5 is smaller and is at a higher position in the array.
- After the swap, 5 is compared to 10 and is again swapped with 10, getting to the 2nd position in the array.
- Then, 5 is compared with 1 and is not swapped, resulting in a partially sorted array.



Fifth Pass:

- Now, the elements 1, 5, 10, 23 are sorted with respect to each other, leaving only 2.
- Comparison starts from 2 with 23, and as 23 is greater than 2, a swap operation is performed.
- This comparison and swapping continue until 2 reaches the 2nd position, as 10 and 5 are both greater than 2.
- As 1 is smaller than 2, no swap operation is performed, and the result is the complete sorted array.



Hence, the sorted array is:

$$\text{arr}[] = \{1, 2, 5, 10, 23\}$$

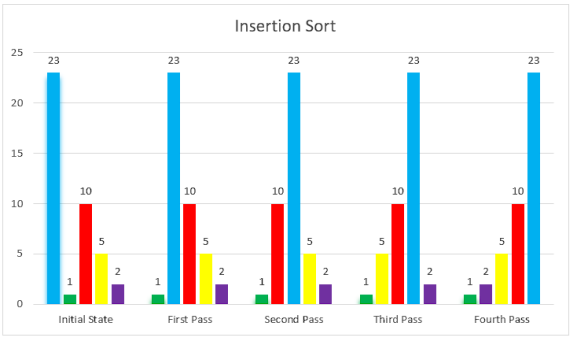


Chart 2 : Selection sort algorithm passes

Algorithm 2 : Insertion Sort**?**

```
1: for j ← 2 to Arr.lenght do
2:     key = Arr[j]
3:     i = j − 1
4:     while i > 0 and Arr[i] > key do
5:         Arr[i + 1] = Arr[i]
6:         i = i − 1
7:     end while
8:     Arr[i + 1] = key
9: end for
```

Loop Invariant: At the start of each iteration of the *for* loop of lines 1 - 9, the subarray $Arr[1..j-1]$ consists of the elements originally in $Arr[1..j-1]$,but in sorted order.**?**

3 Results and Analysis

Time Complexity:

- **Selection Sort:** The time complexity of Selection Sort is $O(n^2)$ in the worst and average cases. This is because, for each element, it needs to traverse the remaining unsorted part of the array to find the minimum element and swap it with the current element.

- **Insertion Sort:** The time complexity of Insertion Sort is also $O(n^2)$ in the worst and average cases. It involves iterating through the array and inserting each element into its correct position by comparing it with the elements to its left.

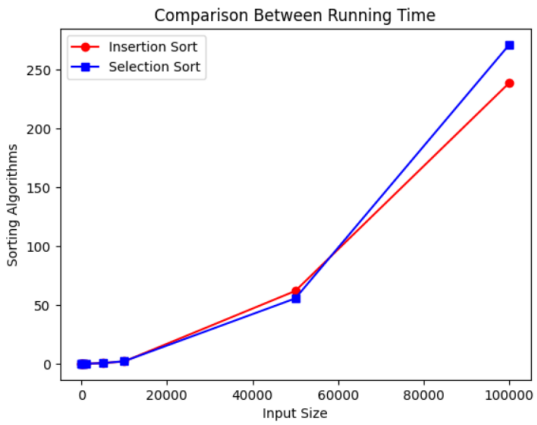
Best-Case Scenario:

- **Selection Sort:** The best-case time complexity of Selection Sort is $O(n^2)$. Even if the array is partially sorted, Selection Sort still needs to traverse the entire unsorted part for each element.

- **Insertion Sort:** The best-case time complexity of Insertion Sort is $O(n)$ when the array is already sorted. In such cases, it only needs to compare each element with its adjacent element to verify the order.

Size	Insertion (s)	Selection (s)
10	0.000 011	0.000 019
50	0.000 176	0.000 155
100	0.000 412	0.000 363
500	0.004 555	0.004 457
1000	0.020 648	0.018 857
5000	0.528 669	0.485 517
10 000	2.105 405	1.965 638
50 000	55.541 591	61.877 976
100 000	238.821 475	271.276 342

Table 1: *Performance Comparison of Insertion Sort and Selection Sort*



Graph 1 : Running time comparison of Insertion Sort and Selection Sort

Interpretation of the Graph:

- As the dataset size increases, the running time of both algorithms tends to increase, which is expected for sorting algorithms with quadratic time complexity.
- The lines illustrate the trend of how the runtime increases with the size of the input data.
- For smaller dataset sizes, Insertion Sort might perform better than Selection Sort. However, as the dataset size increases, Selection Sort may start to perform relatively better than Insertion Sort.

4 Conclusion

We compared insertion sort and selection sort to assess their performance across varying amounts of data, with a focus on understanding how their efficiency changes with different dataset sizes.

Our findings indicate that insertion sort tends to outperform selection sort on small datasets, especially when dealing with data that is already well-sorted. However, as the dataset size increases, selection sort emerges as a strong candidate due to potential efficiency benefits. This observation may be attributed to the decreasing constant factor in time complexity.

This comparative analysis provides insights into the strengths and weaknesses of these sorting methods. While both insertion sort and selection sort have their merits, the choice should be based on the specific characteristics of the data and the scale of the sorting task.

We acknowledge certain limitations in our study, including a focus on worst and average scenarios and an assumption of a uniform distribution for random data generation. This opens avenues for future research to explore the real-world effects of these algorithms and discover ways to enhance their capabilities. In summary, our study contributes to a deeper understanding of sorting algorithms, offering valuable insights for decision-making when selecting algorithms for diverse applications.

Examining insertion and selection sorts not only enhances comprehension of their inner workings but also stimulates ongoing efforts to optimize the sorting process for various computational situations.