Comparative Analysis of Selection Sort and TimSort

Shahriar Rizvi
Department of CSE
RUET
2003104@student.ruet.ac.bd

Tanvir Mahamood
Department of CSE
RUET
2003062@student.ruet.ac.bd

Rushnan Reaz
Department of CSE
RUET
2003101@student.ruet.ac.bd

Jaeed Mahmud
Department of CSE
RUET
2003074@student.ruet.ac.bd

Nafees Ahamed
Department of CSE
RUET
2003079@student.ruet.ac.bd

Rakebul Hasan Nur
Department of CSE
RUET
2003066@student.ruet.ac.bd

Abstract

This article presents two popular sorting algorithms naming Selection Sort and TimSort. Selection Sort is comparable to a brute force approach, while TimSort employs a divide-and-conquer strategy by incorporating merge sort. In particular, both merging sort and Selection Sort share the same best-case and worst-case complexities $O(nlogn)$ and $O(n^2)$, respectively. The goal of this article is to discuss about the sorting techniques, comparing their performance with real life data, discussing about complexity analysis in terms of best case and worst case and space or auxiliary complexity. These discussions will be enough to make a comparison between these algorithms.

I. Introduction

Sorting algorithms are essential to algorithmic design and optimization because they have a significant impact on the effectiveness and performance of numerous computer operations. In this article, TimSort[9] and Selection Sort[6]—two sorting algorithms that have been widely used in a variety of computing environments—are thoroughly compared.

An easy-to-understand algorithm, selection sort is a mainstay in many sorting implementations and is commonly taught as a core topic in introductory computer science classes. But there are trade-offs associated with its simplicity, especially when it comes to time complexity. Conversely, TimSort is a hybrid sorting algorithm that combines advanced strategies to improve performance on a range of data distributions. It is derived on Merge Sort and Insertion Sort.

Our investigation' main intention is to clarify Selection Sort and TimSorts' running time characteristics in a variety of scenarios, from the best to the worst. We also explore the space-time trade-offs that these algorithms entail, providing insight into their overall efficiency and memory needs.

We use datasets from reliable sources that specialize in machine learning datasets to provide a solid and insightful analysis. Moreover, locally generated data are integrated to emulate real-world situations, offering a thorough comprehension of the performance of various sorting algorithms in real-world scenarios.

We hope to provide researchers, developers, and educators with useful insights about the relative advantages and disadvantages of TimSort and Selection Sort as we traverse the complexities of algorithmic complexity. This article adds to the current discussion on algorithmic efficiency by providing a nuanced viewpoint on its performance characteristics. It also facilitates the process of making well-informed decisions when choosing sorting algorithms for particular computational jobs.

II. Background Study

Selection Sort

Selection Sort[6] is a simple but inefficient algorithm for sorting. In this algorithm, we have to find the smallest element of the array and move it to the front of the array. We need to use a linear search technique to find the smallest element. Here, moving an element means swapping. After that, we find the second smallest element and move it to its actual position. This process should continue until all the elements are in their right places. At that time, the array will be completely sorted in ascending order.Suppose 'A' is given as an array that

```
Algorithm                                                Cost   Number of Operations

selectionSort(arr: A)                                    C1     n
    for i from 1 to A.length-1 do                        c2     (n-1)
        minIndex = i
        for j from i+1 to A.length do                    c3     (n-1)+(n-2)+...+3+2+1
            if A[j] < A[minIndex] then                          = n(n-1)/2
                minIndex = j
            end if
        end for
        if A[i] < A[minIndex] then                       c4     (n-1)
            Temp = A[i]                                  c5     (n-1)
            A[i] = A[minIndex]                           c6     (n-1)
            A[minIndex] = temp                           c7     (n-1)
    end for
```

Total Complexity: $c_1 n + c_2(n-1) + c_3 n(n-1)/2 + c_4(n-1) + c_5(n-1) + c_6(n-1) + c_7(n-1)$

is to be sorted. The array is one indexed.

Worst Case Time Complexity: This can be achieved when we need to sort elements in ascending order, but the array is initially in descending order. The outer and inner loops do not depend on the order of arrangement of the array. It is the common cost for both the best and worst case. The only difference is the 'SWAP' operation. During each iteration, the swap operation will be performed. the overall time cost:
$T(n) = c_1 n + c_2(n-1) + c_3 n(n-1)/2 + c_4(n-1) + c_5(n-1) + c_6(n-1) + c_7(n-1) = (c_3/2)n^2 + (c_1 + c_2 - c_3/2 + c_4 + c_5 + c_6 + c_7)n + (-c_2 - c_4 - c_5 - c_6 - c_7)$

This running time can be expressed as
$T(n) = an^2 + bn + c$
Here, $a = c_3/2, b = c_1 + c_2 - c_3/2 + c_4 + c_5 + c_6 + c_7$ and $c = -c_2 - c_4 - c_5 - c_6 - c_7$.
So, the worse case complexity is $\mathrm{O}(n^2)$

Best Case Time Complexity: As mentioned before, the number of iterations of the outer and inner loops are free from the order of array arrangement, we should only focus on the swap operation. The best case happens when the array is already sorted in ascending order as we wanted. At that time, no swap operation was needed during the total iterations. Because the i^{th} element is smaller than all j^{th} elements. Thus, the total time cost will be reduced slightly. That is:
$T(n) = c_1 n + c_2(n-1) + c_3 n(n-1)/2 + c_4(n-1) = (c_3/2)n^2 + (c_1 + c_2 - c_3/2 + c_4)n + (-c_2 - c_4)$
This running time can be expressed as
$T(n) = a'n^2 + b'n + c'$
Here, $a' = c_3/2, b = c_1 + c_2 - c_3/2 + c_4$ and $c = -c_2 - c_4$.
So, the best case complexity: $\mathrm{O}(n^2)$
In short, time Complexity is $\mathrm{O}(n^2)$ for both best-case and worst-case. But the constant factor during the best case is smaller than in the worst case.

Auxiliary Space: As no extra space is needed to sort the array except the variable 'Temp' for swapping, the space complexity is O(1).

Stability: The stable order is one that preserves the original order of the input set. The Selection Sort is an unstable sort. Here is an example.
Suppose that the array is [5 3 4 5 2 6 8].
Let's distinguish the two 5's as 5(a) and 5(b). So, our array is: [5(a) 3 4 5(b) 2 6 8].
After the first iteration, 5(a) will be swapped with 2. So, our array becomes: [2 3 4 5(b) 5(a) 6 8]
Now, the array is sorted. We have seen that 5(a) comes before 5(b) in the initial array. But after sorting. Since now our array is in sorted order, we see that 5(a) comes before 5(b) in the initial array but not in the sorted array. So, we can see that the Selection Sort is not stable. In this paper, we will use the Selection Sort algorithm given in [6].

<u>TimSort</u>

TimSort, a hybrid algorithm was created in 2002 with an emphasis on real-world data by Tim Peters[9]. Python, the Java standard library, GNU Octave, and the Android operating system utilize different refined versions of TimSort as their default sorting method.
Before TimSort, QuickSort was widely used as the fastest algorithm for sorting elements. In the work [3], they proposed a hybrid algorithm which is a synergy of QuickerSort[10] and Straight Insertion Sort[7] for almost sorted elements[3]. In 2002, Tim Cook proposed a stable merge sort that was claimed to be faster than any other algorithms for sorting real-world elements, as sorting real-world elements rarely has a high out-of-order percentage [9]. Auger et al.[2] were the first to prove

TimSort in their work in 2015. Further improvements can be found in [1] and [4]. The refinement proposed in the work of Stijn de Gouw[4] is now used in Python. But, the Java standard library uses the first version of TimSort by adjusting tuning values which was exposed in [4].

Timsort mainly uses InsertionSort and Merge Sort. In TimSort, if the algorithm tends to find sub-problems ranging from 32 to 64 elements[9]. This approach stops dividing operations and the number of levels of recurrence is reduced. Now the remaining subarray is sorted using Insertion Sort. After sorting each sub-arrays, the algorithm merges these sub-arrays. The constant factor of the insertion sort is lower than the merge sort. It is known that Insertion Sort executes more efficiently than Merge Sort for a small number of elements[3]. That's why adding this new feature can increase runtime quality. So, the number of levels is decreased from $\log(n)$ to $(\log(n) - x)$, here, x is the number of levels that are ignored because of Insertion Sort.

Now, the algorithm is further modified to improve the execution time by introducing 'Run'. 'Run' is a sorted sublist of elements among the elements[8]. If we find a sub-array that is already sorted, we do not divide it further, and this sub-array will be recognized as 'run'. The 'Run' has to be strictly in Ascending order[9]. 'Run's do not need any sorting algorithm. All 'Run' and the other smallest subarrays are merged. Thus, the overall complexity is improved.

Best Case Time Complexity: In the best case scenario, the array is already sorted and the entire array will work as 'Run'. So, we do not need to take a divide-and-conquer approach. Simply, the time complexity is O(n).

Worst Case Time Complexity: In the worst case scenario, the Time complexity is $O(n \log(n))$ which is proven in the work of Auger et al.[2, 1].

Auxiliary Space: The auxiliary (or extra) space complexity of TimSort is O(n). TimSort uses additional space for temporary storage and various data structures during the sorting process. The primary data structure is the auxiliary array used for merging the sorted runs.TimSort may use a stack to keep track of the runs and their indices during the sorting process.

Stability: TimSort is stable because it uses a merging mechanism that preserves the relative order of equal elements during the sorting process. It also employs Insertion Sort for small sequences, which is inherently stable. The comparison and swapping criteria in TimSort are designed to maintain stability, ensuring that equal elements retain their original order.

Comparison

- TimSort is very optimized. Its best-case and worst-case time complexity is O(n) and $O(n \log(n))$ respectively. On the other hand, Selection Sort is very inefficient and runs in $O(n^2)$ for both worst and best cases.
- Selection Sort uses no extra space for sorting. It is space-optimized. TimSort is not space optimized.
- TimSort is a stable sorting algorithm. Selection Sort is unstable.

III. Results and Analysis

The algorithms under consideration were implemented in the Python programming language and rigorously tested using random sequence inputs of varying lengths, including 900, 9000, and 100000 elements. The experiments were carried out in a computing environment equipped with an 11^{th} generation Intel(R) Core(TM) i7-1195G7 processor operating at 2.90GHz, complemented by 16GB RAM.

Time Analysis

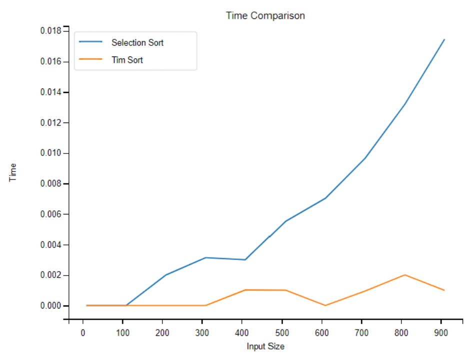
We conducted a comprehensive time analysis to assess the performance of Selection Sort and TimSort across varying dataset sizes. The results are illustrated in Figures 1a, 1b, and 1c for data set sizes of 900, 9000, and 100000 elements, respectively. The observed trends indicate distinctive behaviors in the growth of execution time as the dataset size increases. In particular, the **Selection Sort** algorithm (pictured in blue) exhibits exponential growth, showcasing a substantial increase in execution time with larger datasets. In contrast, the **TimSort** algorithm (pictured in salmon) demonstrates a more moderate growth rate, suggesting its efficiency in handling larger data sets.

A particularly noteworthy observation is the discernible difference in the curves for higher input sizes, such as 900 to 9000 and 100,000. In this range, the visual contrast between the performance of Selection Sort and TimSort becomes more pronounced, further emphasizing the superior scalability of TimSort for substantial datasets.

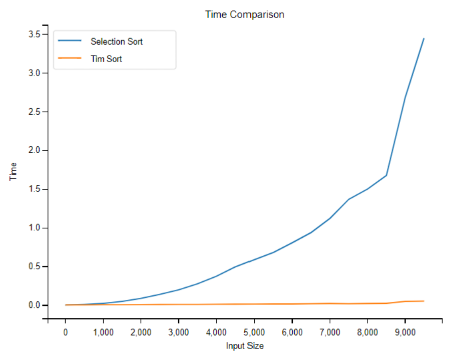
As depicted in the figures, TimSort consistently outperforms Selection Sort across all dataset sizes. The divide-and-conquer strategy employed by TimSort demonstrates its efficiency, particularly as the dataset size increases.

Theoretical Time Complexities

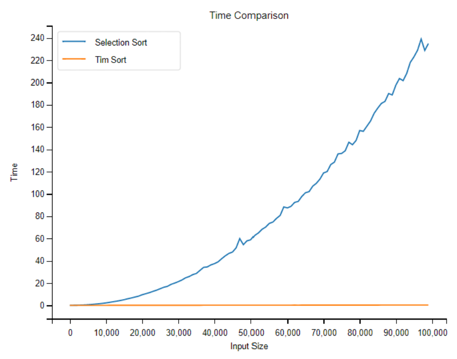
To provide a theoretical context for our empirical findings, we include graphs representing the time complexities of $O(n^2)$ and $O(n \log n)$ in Figure 1d. Which is generated in Python using 'matplotlib' library.



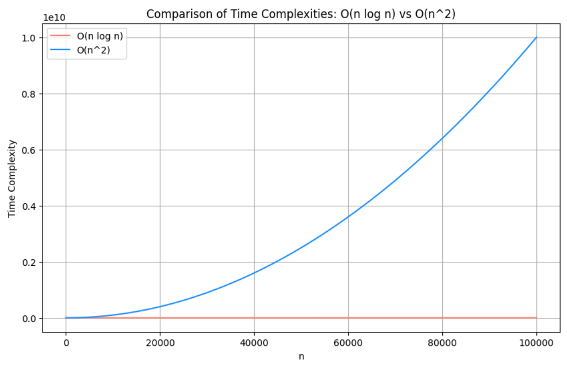
(a) For data size: 0 to 900



(b) For data size: 0 to 9000



(c) For data size: 0 to 100,000



(d) For theoretical $O(n^2)$ and $O(n \log n)$

Figure 1: Experimented and Theoretical Time Comparison between Selection and TimSort

The theoretical complexities serve as a reference point, confirming that the observed performance aligns with the expected behaviors of Selection Sort and TimSort.
In summary, our empirical results and theoretical considerations collectively suggest that TimSort is a more efficient sorting algorithm compared to Selection Sort, especially in scenarios with larger datasets.

IV. Conclusion

This paper represents two sorting algorithms (Selection Sort and TimSort), their sorting techniques, time complexity, and performance. To compare sorting algorithms, the time complexity is the main consideration. It analyzes the time complexity for the worst and the best cases. Moreover, several data sets are used to compare the performance of these two algorithms. It upholds the comparison of performance by graph, which highlights the high efficiency of TimSort for a large number of elements that are to be sorted.

References

[1] Auger, N., Jugé, V., Nicaud, C., and Pivoteau, C. On the worst-case complexity of timsort.

[2] Auger, N., Nicaud, C., and Pivoteau, C. Merge strategies: from merge sort to timsort.

[3] Cook, C. R., and Kim, D. J. Best sorting algorithm for nearly sorted lists. *Commun. ACM 23*, 11 (nov 1980), 620–624.

[4] Gouw, S., Rot, J., Boer, F., Bubel, R., and Hähnle, R. Openjdk's java.utils.collection.sort() is broken: The good, the bad and the worst case. pp. 273–289.

[5] Knuth, D. E. 2nd ed. Addison-Wesley, 1998.

[6] Knuth, D. E. Volume-3: Sorting and searching. In *The Art of Computer Programming* [5], pp. 138–158.

[7] Knuth, D. E. Volume-3: Sorting and searching. In *The Art of Computer Programming* [5], p. 80.

[8] Knuth, D. E. Volume-3: Sorting and searching. In *The Art of Computer Programming* [5], p. 160.

[9] Peters, T. Timsort description. `http://svn.python.org/projects/python/trunk/Objects/listsort.txt`, 2015. Accessed June 2015.

[10] Scowen, R. S. Algorithm 271: quickersort. *Commun. ACM 8*, 11 (nov 1965), 669–670.