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

Presentation Overview

Abstract

Abstract

This article presents two popular sorting algorithms named Selection Sort and TimSort. The goal of this article is to discuss the sorting techniques, compare their performance with real-life data, and discuss complexity analysis in terms of best-case and worst-case, and space or auxiliary complexity.

Introduction

Introduction

Selection Sort is akin to a brute force approach, while Tim Sort employs a divide-and-conquer strategy by incorporating merge sort.
Selection sort is a sorting algorithm that selects the smallest element from an unsorted list in each iteration and places that element at the beginning of the unsorted list. Tim Sort is a hybrid algorithm modified from merge sort and insertion sort. It was designed to perform real-life operations. Tim Sort is the default sorting algorithm used by Python's sorted() and list.sort() functions.

Background Study: Selection Sort

Selection Sort

- Selection Sort[Knuth, 1998b] is a straightforward but inefficient algorithm used for sorting arrays.
- The main idea is to find the smallest element in the array and move it to the front, repeating this process until the array is sorted.
- This algorithm involves a linear search to find the smallest element and utilizes swapping to rearrange the elements

Algorithm

Algorithm Selection Sort

```
1: procedure SelectionSort($(A, n)$)    ▷ A is the array to be sorted, n is the size of the array
2:     for $i \leftarrow 0$ to $n - 1$ do
3:         $minIndex \leftarrow i$
4:         for $j \leftarrow i + 1$ to $n$ do
5:             if $A[j] < A[minIndex]$ then
6:                 $minIndex \leftarrow j$
7:             end if
8:         end for
9:         Swap $A[i]$ and $A[minIndex]$
10:    end for
11: end procedure
```

Worst Case Analysis

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

- This case can be achieved when we need to sort an array in ascending order, but the array is initially in descending order. This uses a 'Swap' operation at every iteration. So, the overall time cost:

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + \frac{c_3 n(n-1)}{2} + c_4(n-1) + c_5(n-1) + c_6(n-1) + c_7(n-1) \\ &= \frac{c_3}{2}n^2 + (c_1 + c_2 - \frac{c_3}{2} + c_4 + c_5 + c_6 + c_7) - (c_2 + c_4 + c_5 + c_6 + c_7) \end{aligned}$$

This running time can be expressed as
$T(n) = an^2 + bn + c$
Here, $a = \frac{c_3}{2}$, $b = c_1 + c_2 - \frac{c_3}{2} + c_4 + c_5 + c_6 + c_7$, and $c = -c_2 - c_4 - c_5 - c_6 - c_7$.
Complexity: $O(n^2)$

Best Case Analysis

- **Best case:** Best case happens when the array is already sorted in ascending order as wanted. That time, no swap operation was needed during the total iterations. Thus, the total time cost will be reduced slightly. That is:

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_3 n(n-1)/2 + c_4(n-1) \\ &= (c_3/2)n^2 + (c_1 + c_2 - c_3/2 + c_4)n + (-c_2 - c_4) \end{aligned}$$

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 $c = -c_2 - c_4$.

Complexity: $O(n^2)$

Space Complexity and Stability

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

Stability: Selection Sort is an unstable sort.

Example - Initial Array And Passes

$$A = [64, 25, 12, 22, 11]$$

$$\begin{aligned} AfterFirstPass &: A = [11, 25, 12, 22, 64] \\ AfterSecondPass &: A = [11, 12, 25, 22, 64] \\ AfterThirdPass &: A = [11, 12, 22, 25, 64] \\ AfterFourthPass &: A = [11, 12, 22, 25, 64] \end{aligned}$$

Background Study: TimSort

Tim Sort

- TimSort, a hybrid sorting algorithm invented by Tim Peters in 2002[Peters, 2015].
- Designed mainly addressing real-world data concerns.
- It is derived by modifying Merge Sort[Knuth, 1998a] by involving Insertion Sort[Knuth, 1998c] to perform an optimized sorting algorithm.
- Python uses the refined verion proposed by Gouw et al. [2015], Java uses the initial TimSort version with tuning values[Auger et al., 2019]. GNU Octave and Android OS also use TimSort as the default sorting algorithm.

Algorithm

Algorithm Tim Sort

```
1: procedure TIMSORT($A$, $n$)
2:     Calculate and divide $A$ into small min_runs[1].
3:     Perform insertion sort on each min_run.
4:     Merge min_runs using modified merge sort. Collapse Runs until a single run is
       formed.
5: end procedure
```

[1]min_runs are small subsets of A for which the Insertion Sort will perform in best complexity. As stated by Peters [2015] min_run should range from 32 to 64 elements

Complexities and Stablities

- Best Case Complexity: $O(n)$ - Array is already sorted.
- Worst Case Complexity: $O(n\log(n))$ - which was firstly proven in the works of Auger et al. [2015, 2019].
- Auxiliary Space Complexity: $O(n)$ - Additional space for temporary storage and data structures.Uses auxiliary arrays and may employ a stack.
- Stable algorithm: Preserves relative order of equal elements. Merging mechanism and Insertion Sort maintain stability.

The Concept of Run

- **Definition of a Run:** A run is a contiguous subsequence of elements in an array with a specific order.
- Runs can be classified as either:
  - **Ascending Run:** Elements are non-decreasing ($a_0 \leq a_1 \leq a_2 \leq \ldots$).
  - **Descending Run:** Elements are strictly decreasing ($a_0 > a_1 > a_2 > \ldots$).

Efficiency in TimSort:

- Runs contribute to the efficiency of TimSort by minimizing unnecessary operations.
- In random data, runs tend to be of length 'min_run', leading to balanced merges.
- In Peters [2015] suggests run should be strictly ascending.
- In the best-case scenario, the array is already sorted and the entire array will work as run.

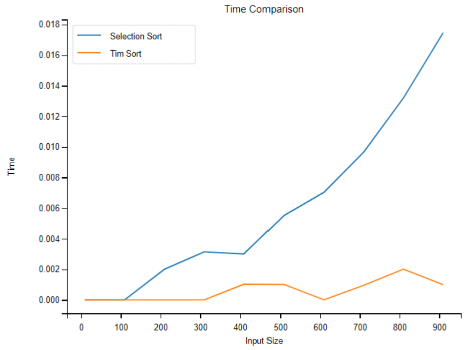
Results and Analysis

Results and Analysis

In this section, we present the results of our time analysis comparing Selection Sort and Tim Sort on datasets of different sizes (900, 9000, and 100000 data points). Additionally, we include visualizations of the theoretical time complexities represented by $O(n^2)$ and $O(n \log n)$ graphs for reference.

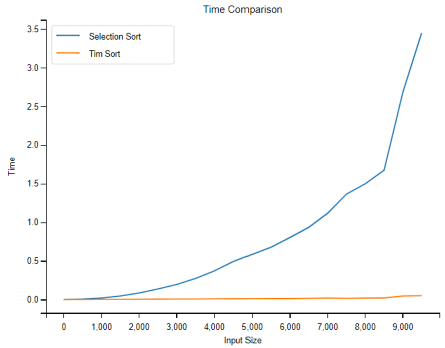
Time Analysis

We conducted a comprehensive time analysis to assess the performance of Selection Sort and Tim Sort across varying dataset sizes. The results are illustrated in Figures 1a, 2a, and 3a for dataset sizes of 900, 9000, and 100000 data points, respectively.



(a) For data size: 900

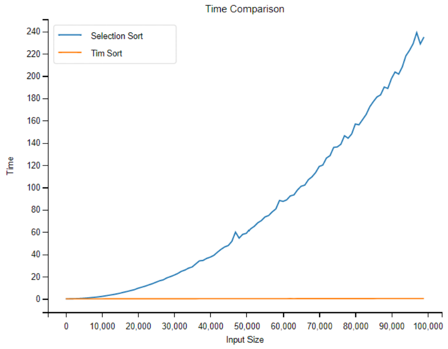
Comparison Graph



(a) For data size: 9000

Figure: Time Comparison between Selection and Tim Sort(9000)

Comparison Graph

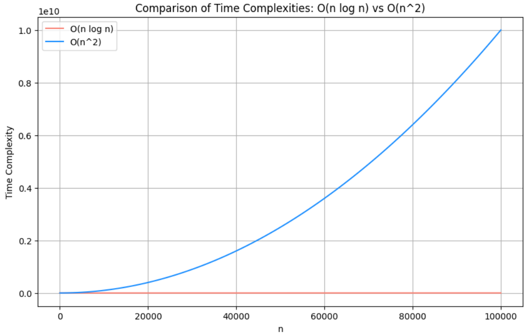


(a) For data size: 100,000

Figure: Time Comparison between Selection and Tim Sort(100 000)

Theoretical Time Complexities

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



Theoretical Time Complexities

The theoretical complexities serve as a reference point, confirming that the observed performance aligns with the expected behaviors of Selection Sort and Tim Sort.

In summary, our empirical results and theoretical considerations collectively suggest that Tim Sort is a more efficient sorting algorithm compared to Selection Sort, especially in scenarios with larger.

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.

Conclusion

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 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.

So, We can say that for any case Timsort is better than Selection Sort but in terms of space complexity Selection Sort is better as it consumes less coding space.

References

Nicolas Auger, Cyril Nicaud, and Carine Pivoteau. Merge strategies: from merge sort to timsort. November 2015.

Nicolas Auger, Vincent Jugé, Cyril Nicaud, and Carine Pivoteau. On the worst-case complexity of timsort. 2019.

S. Gouw, Jurriaan Rot, Frank Boer, Richard Bubel, and Reiner Hähnle. Openjdk's java.utils.collection.sort() is broken: The good, the bad and the worst case. pages 273–289, 07 2015. ISBN 978-3-319-21689-8. doi: 10.1007/978-3-319-21690-4_16.

Donald E. Knuth. Volume-3: Sorting and searching. In *The Art of Computer Programming* Knuth [1998d], page 160.

Donald E. Knuth. Volume-3: Sorting and searching. In *The Art of Computer Programming* Knuth [1998d], pages 138–158.

Donald E. Knuth. Volume-3: Sorting and searching. In *The Art of Computer Programming* Knuth [1998d], page 80.

Donald Erwin Knuth. Addison-Wesley, 2nd edition, 1998d.

Tim Peters. Timsort description. `http://svn.python.org/projects/python/trunk/Objects/listsort.txt`, 2015.

Contribution

- Paper Writing:
  Tanvir Mahamood(2003062)
  Jaeed Mahmud(2003074)
- Paper Formatting:
  Shahriar Rizvi(2003104)
  Rushnan Reaz(2003101)
- Slide Preparation:
  Rakebul Hasan Nur(2003066)
  Nafees Ahamed(2003079)

Thank You