Journal of Informatics
Electrical and Electronics Engineering, 2024,
Vol. 05, Iss. 01, S. No. 107, pp. 1-9
ISSN (Online): 2582-7006

Sorting Visualizer: A Visual Journey Through Sorting Algorithms

Shreya Singh[1], Shikha Singh[2], Vineet Singh[3], Bramah Hazela[4]

[1,2,3,4]Amity School of Engineering and Technology, Amity University, Uttar Pradesh, Lucknow, India
[1]shreya.singh16oct@gmail.com, [2]ssingh8@lko.amity.edu, [3]vsingh@lko.amity.edu, [4]bhazela@lko.amity.edu

How to cite this paper: S. Singh, S. Singh, V. Singh, B. Hazela, "Sorting Visualizer: A Visual Journey Through Sorting Algorithms," *Journal of Informatics Electrical and Electronics Engineering* (*JIEEE*), Vol. 05, Iss. 01, S No. 107, pp. 1–9, 2024.

https://doi.org/10.54060/a2zjournals.jieee.107

Received: 05/12/2023
Accepted: 10/03/2024
Online First: 25/04/2024
Published: 25/04/2024

Abstract

This paper, which is based on the importance of sorting algorithms, will carefully compare the features of various algorithms, beginning with their work effectiveness, algorithm execution, introductory concepts, sorting styles, and other aspects, and make conclusions in order to create more effective sorting algorithms. Searching techniques and sorting algorithms are not the same. Sorting is placing the provided list in a predetermined order, which can be either ascending or descending, whereas searching is predicated on the possibility of finding a specific item in the list. Only a section of the data is sorted, and the piece of data that's actually used to establish the sorted order is the key. The maturity of this data is being compared. Depending on the kind of data structure, there are several algorithms for doing the same set of duties and other conditioning, and each has pros and cons of its own. Numerous sorting algorithms have been analyzed grounded on space and time complexity. The aim of this relative study is to identify the most effective sorting algorithms or styles. This relative study grounded on the same analysis allows the user to select the applicable sorting algorithm for the given situation.

Keywords

Sorting, searching, algorithms, data structures, comparisons, performance, complexity

1. Introduction

Based on the advancement of knowledge and technology, individuals typically use computers to compare various impacts, particularly when organizing difficulties. Examples of such tasks include ranking test results, thoroughly analysing product recommendations, and categorizing challenges. The utilization of data and information processing causes the sorting problems to become less and less thick. Sorting is an initial function in data processing. Numerous examples of sorting can be

seen in daily life. For example, we can easily locate necessary information in a travel store or shopping promenade because the information is categorized.[1] Because all the words are provided in sorted form, picking a word at random from a word-book isn't a laborious operation. Additionally, one may find a phone number, name, or address in a phone book quite easily because to the advantages of sorting. Sorting is considered one of the most crucial procedures in computer knowledge due to its essential functions. Other examples of sorting include priority scheduling and shortest job first scheduling.[2] There are many different ways that the details need to be sorted: they can be randomly arranged as a whole, previously sorted, incredibly little or very large in quantity, sorted in reverse order, etc. [3] A method that works for sorting every kind of data is not available. Meanwhile, sorting the data typically takes a minimum of 35% of the time when a computer tackles a given problem. In relation to this issue, the most efficient method for refining the computer's operational efficiency would be to research sorting algorithms and, based on the analysis, select the most appropriate sorting algorithm for the given circumstance.

In this paper we're going to compare six (Bubble, Quick, Insertion, Selection, Merge and Heap) sorting algorithms for their performance on a given input in best, worst and average cases. This tool, known as SV (Sorting Visualizer), aims to help scholars who are generally visual learners and who comprehend material better when presented visually.[4] As a result, we've created a web application that helps us understand the sorting algorithmic process better by visualizing it. Our primary motive was to make the vitality web grounded so that it could be used by a large number of people across numerous platforms without taking the installation of fresh software or operations. Because Sorting Visualizer is an online tool, it may be used by a large number of individuals from anywhere in the world without requiring the download of any unnecessary features, which can be beneficial for everyone, but especially for students. Humans naturally prefer to communicate thoughts visually, and images have a significant impact on the way that information is processed. Early cave drawings demonstrate a prehistoric desire to utilize images to try and communicate ideas and meaning.[5] Even today, majority of kids try to make meaningful visualizations by sketching simple pictures that depict their thoughts. In an effort to identify visual patterns, sorting visualizer was developed to showcase visual representations of sorting algorithms. It was discovered that each algorithm's visual footprint is distinct from the others, which makes it easier for those unfamiliar with the idea of sorting to comprehend these algorithms and how they operate/work.

The process of developing the visualizer and drawing important conclusions are covered in detail in the paper. After the introduction, the following sections would discuss the project: the Literature Survey would provide a contextual background and briefly reference all the related works in the field that helped me with the project. Methodology would describe the development process; Algorithm Analysis would delve into theoretical and practical aspects of the algorithms used; Results and Discussion would present empirical findings; and finally, Conclusion and Future scope would summarize key insights from the paper and give a glimpse of potential future work that could be done in the field.

2. Literature Survey

The paper analyses and compares different sorting algorithms based on their time and space complexity. It describes popular sorting algorithms like bubble sort, insertion sort, selection sort, merge sort, quicksort, heap sort, radix sort, and bucket sort. It explains how the time complexity of these algorithms can be analyzed using Big O notation. It also discusses factors like average-case and worst-case time complexity, and space complexity. Real-world applications of sorting algorithms are mentioned in areas like databases, e-commerce, search engines, data analysis etc. The document concludes that no single sorting algorithm is best and the most appropriate one needs to be selected based on the characteristics of the input data.[6]

The paper proposes a new quadratic sorting algorithm and compares its performance to existing quadratic sorting algorithms like insertion sort, bubble sort, selection sort, quicksort, etc. It demonstrates that the new algorithm exploits the idea

that an unsorted data sequence can be viewed as a set of disjoint sorted sequences. The algorithm works by building a relation be-tween data elements in each iteration and sorting the elements accordingly. Graphical comparisons show that the algorithm often has better time complexity and execution time compared to other quadratic algorithms, especially on large datasets.[7]

The document discusses various sorting algorithms and proposes a new sorting algorithm called MinFinder. It describes properties of sorting algorithms, types of sorting algorithms, and complexity analyses. MinFinder finds the smallest element and places it in the first position of the array by shifting other elements to the right. It repeats this process to sort all elements. The document analyses time and space complexity, loop invariant, and performance comparisons of MinFinder with other algorithms.[8]

The document describes an approach called Alpha Dev that uses deep reinforcement learning to discover new and optimized algorithm implementations directly from assembly code. AlphaDev models the problem as a game called Assembly Game where the goal is to generate a correct and efficient algorithm by appending assembly instructions. It represents algorithms, inputs/outputs and machine state to learn policies and value functions to guide the search. AlphaDev is able to discover new sorting algorithms that improve on human benchmarks in terms of length and latency. This includes discovering new swap/copy moves that reduce instructions. It also finds improved variable sorting routines. The approach generalizes to other domains like protocol buffer decoding. Comparisons show AlphaDev explores more than stochastic search baselines and is able to escape local optima.[9]

This paper performs a comparative study of five sorting algorithms: Bubble Sort, Quick Sort, Selection Sort, Insertion Sort, and Merge Sort. It develops a program in C# to calculate the CPU time taken by each algorithm on different input sizes (1-150, 1-300, 1-950) in best, worst, and average cases. The results are presented in tabular and graphical form. Based on the experiments, the paper ranks the performance of the algorithms on the given data sets and input sizes.[10]

This article discusses efficient algorithms for sorting and searching strings or integers of length w (the word size of the RAM) beyond what is possible with comparison-based algorithms. It presents some simple algorithms using tries and packing keys tightly that allow sorting in O(n log log n) time and searching in O(log^c n) time, though they use super linear space. It also covers data structures like fusion trees that support sorting and searching in O(n log n/log log n) and O(log n log log n) worst-case time respectively using only linear space.[11]

This paper compares the performance of three sorting algorithms: selection sort, bubble sort, and gnome sort. It explains each algorithm, provides pseudocode examples, and analyses their time complexities. The paper implements the three algorithms in C# and measures their execution times on random unsorted and sorted data sets of varying sizes. The results show that gnome sort performs best on pre-sorted data, while selection sort is fastest for unsorted data.[12]

This document provides a comparative study of various sorting and searching algorithms. It discusses common sorting algorithms like quicksort, selection sort, bubble sort, insertion sort, merge sort and heapsort. It also covers searching algorithms like linear search and binary search. For each algorithm, it explains the steps, provides examples, and discusses the advantages and disadvantages. The document analyses the time and space complexity of these algorithms and concludes that different algorithms are suitable depending on the size of input data and requirements.[13]

This paper presents a general framework for analyzing the average-case complexity of sorting and searching algorithms when operating on data produced by a probabilistic source. It describes the algorithms in terms of the expected number of comparisons between elements, relating this to properties of the source like entropy. Asymptotic estimates for the average number of comparisons are obtained by expressing this expectation as a Dirichlet series and using properties like tameness of the source. The framework is applied to algorithms like quicksort, selection sort, insertion sort and bubble sort.[14]

This document discusses various algorithms for sorting and implementing dictionaries (data structures that support search, insert and delete operations). It begins with an introduction to basic data structures like arrays and linked lists. It then

covers sorting algorithms like insertion sort, shell sort, quicksort and compares their performance. The next section discusses implementations of dictionaries using hash tables, binary search trees, red-black trees, skip lists and compares their performance and trade-offs. The last section covers algorithms for sorting and implementing dictionaries for very large files that don't fit in memory, specifically external sorting and B-trees. Code examples are provided in C for many of the algorithms.[15]

3. Proposed System and Methodology

Here, we describe our general framework, or overview of the process we will be following through, in making of the tool.

3.1. Proposed System

The web application is written in HTML5, CSS3 and JavaScript. The physical components of the online application are coded in HTML5. CSS has complete control over the interactive layout, or appearance, of the online application. The remaining duties are completed by JavaScript, which is used for the algorithm design, bar movement and depiction. Many features of the recommended system increase user productivity when utilizing this web application. The user interface consists of six sorting method buttons, a new random array button, and adjustments to the array's size and speed. All of these components are located atop a bar graph that illustrates the entire process. The user clicks on any sorting button he/she wishes to implement, and the bars rearrange themselves according to the speed set and ultimately fall in their respective sorted positions. The basic process flow is shown in Figure 1.

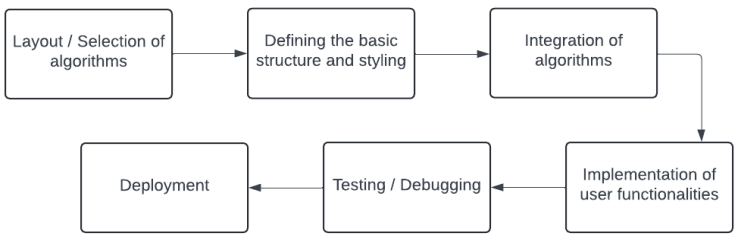

Figure 1. Process Flow

3.2. Methodology

1. **Project Analysis**:
   The Sorting Visualizer is an interactive web-based application which has to be designed to facilitate the understanding and visualization of various sorting algorithms. Employing a client-server architecture, the frontend would leverage HTML, CSS, and JavaScript for user interface rendering and interaction, while the backend utilizes JavaScript to implement the sorting algorithms.
2. **HTML Structure**:
   The HTML structure forms the backbone of the visualizer's user interface. The document is divided into sections, including a container for displaying the array, buttons for user interaction, and a set of buttons for selecting sorting algorithms. The structuring aims to enhance the user experience, ensuring clarity and simplicity in the presentation of sorting algorithm visualizations.
3. **CSS Styling:**
   A meticulous approach to styling ensures an aesthetically pleasing and coherent user interface. The CSS file defines rules

for various HTML elements, specifying layout properties, colour schemes, and animations. Emphasis is placed on creating a visually engaging representation of the array and user interface components, promoting a seamless user experience during sorting visualizations.

4. **JavaScript Logic**:
   The JavaScript logic is pivotal in orchestrating the functionality of the visualizer. Key functions include generating a randomized array, initiating the sorting process based on user input, and implementing individual sorting algorithms. Each algorithm adheres to a modular structure, allowing for easy integration and maintenance. Real-time visualization is achieved through timed updates to the HTML DOM, ensuring users witness the step-by-step progression of the sorting algorithm.
5. **Array Generation**:
   The array generation process involves a randomized approach to populate the array with integers. This function plays a crucial role in providing diverse datasets for sorting visualizations, contributing to a more comprehensive understanding of algorithmic behaviour under various input conditions.
6. **Sorting Algorithms**:
   The visualizer supports multiple sorting algorithms, with each algorithm encapsulated in its JavaScript function. The algorithms, such as bubble sort, employ visualization techniques to depict the sorting process. Careful consideration is given to the efficiency and clarity of visualization, aiming to enhance the educational value of the tool.
7. **User Interaction**:
   User interaction is a focal point of the visualizer, allowing users to generate new arrays, choose sorting algorithms, and initiate the visualization process. The seamless integration of HTML, CSS, and JavaScript ensures an intuitive and responsive user interface, promoting engagement and facilitating a deeper understanding of sorting algorithms.
8. **Testing**:
   The visualizer undergoes rigorous testing to ensure its reliability and effectiveness. User testing involves generating arrays, selecting different algorithms, and scrutinizing the visual representation to verify the accuracy of the sorting process. This phase also addresses potential edge cases and performance considerations.
9. **Extensibility**:
   Designed with extensibility in mind, the visualizer can be expanded by incorporating additional sorting algorithms, refining the user interface, or introducing new features. The modular structure of the JavaScript code facilitates seamless integration of new functionalities, promoting ongoing development and adaptability.
10. **Styling and Enhancements**:
    Continuous improvement is pursued through ongoing styling refinements and feature enhancements. User feedback is valuable in identifying areas for improvement, and periodic updates are released to address usability concerns, optimize performance, and enhance the overall visual appeal of the sorting visualizer.
11. **Deployment**:
    Upon successful completion of development and testing, the visualizer is deployed to a web server or hosting platform, making it accessible to a broader audience. This stage involves considerations of server-side technologies, scalability, and user accessibility, ensuring a reliable and efficient deployment of the sorting visualizer.

4. Algorithm Analysis

Here, we will analyse the results and draw out key conclusions on the basis of sorting procedures followed.

4.1. Performance Analysis

The aim of this section is to provide a thorough evaluation of sorting algorithms' performance. It includes:

Time complexity Analysis of sorting algorithms

Time complexity analysis measures how long an algorithm takes to execute when input volume increases. When using sorting algorithms, the input size is the total number of elements that need to be sorted. The temporal complexity of sorting algorithms can be understood using big O notation. The temporal complexity of some common sorting algorithms is listed here.

- Selection Sort: $O(n^2)$
- Bubble Sort: $O(n^2)$
- Insertion Sort: $O(n^2)$
- Merge Sort: O(n log n)
- Quick Sort: O(n log n) average case, $O(n^2)$ worst case
- Heap Sort: O(n log n)

Space Complexity Analysis of sorting algorithms

Space complexity analysis is a way to quantify how much memory a program requires as the size of the input grows. The degree of sophistication in sorting algorithms' use of memory depends on whether they sort in-place or require additional memory for temporary storage. The space complexity of a few common sorting algorithms is listed here.

- Selection Sort: O(1)
- Bubble Sort: O(1)
- Insertion Sort: O(1)
- Merge Sort: O(n)
- Quick Sort: O(log n) average case, O(n) worst case
- Heap Sort: O(1)

4.2. Efficiency Analysis

The effectiveness of sorting algorithms can be assessed; this is commonly measured in terms of space and time complexity. While space complexity refers to the amount of memory an algorithm consumes to sort the items, time complexity is the amount of time an algorithm takes to sort a list or array of objects. A "big-O" notation, which represents the upper bound for an algorithm's worst-case scenario, is typically used to indicate time complexity The best-case, average-case, and worst-case scenarios for the sorting algorithms that we examined for our report are arranged in TABLE 1.

Table 1. Time complexity analysis of different sorting algorithms

Sorting Algorithm	***Best Case***	***Average Case***	***Worst Case***
Insertion	O(n)	$O(n^2)$	$O(n^2)$
Selection	$O(n^2)$	$O(n^2)$	$O(n^2)$
Bubble	$O(n^2)$	$O(n^2)$	$O(n^2)$
Heap	O(n log n)	O(n log n)	O(n log n)
Merge	O(n log n)	O(n log n)	O(n log n)
Quick	O(n log n)	O(n log n)	$O(n^2)$

In general, the efficiency of sorting algorithms can be evaluated by comparing their time and space complexity and evaluating

how well they perform when dealing with various types of input data. To guarantee the optimum performance, it is essential to consider the specific needs of the application as well as the characteristics of the input data while choosing a sorting algorithm.

5. Results and Discussion

In The main goal was to keep the web page and user interface as simple as possible so that it's easy for the user to navigate through the page. All the similar elements are coupled together. In this system the user interface build is professional. It is the front end of the project or it can also be termed as user interface. Here the user the gets the multiple options to execute or access their task as per need. The user interface has multiple components: The main navbar allows the user to set the size of the array and to control the speed of visualization, the buttons to generate a new random array, and the sorting algorithm buttons. Figure 2 below depicts the bars that display the state after the visualization process completes upon selection of algorithm.

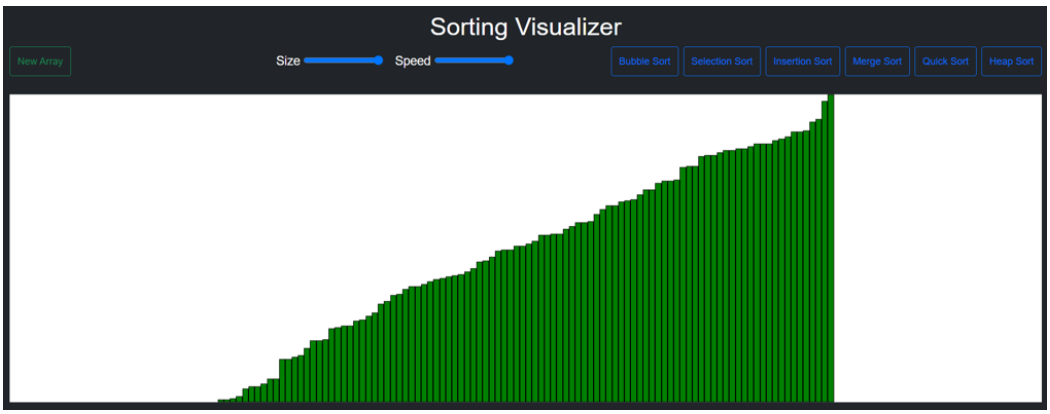

Figure 2. Bars after sorting

There are several benefits and drawbacks to each sorting method. The right algorithm can significantly boost efficiency and performance. The next is memory management. Since sorting algorithms require a lot of memory, memory management is essential to getting the greatest speed possible. Sometimes, sorting algorithms can reduce the amount of storage they use. Algorithms with the ability to alter the data they are given have the potential to function considerably better. An adaptive algorithm may switch to a different sorting algorithm based on the volume of the incoming data. Using parallel processing techniques can enhance sorting algorithms.

How to select the best sorting algorithm?

The effectiveness of sorting algorithms can be assessed; this is commonly measured in terms of space and time complexity. A few general observations can be made with the help of Table 2 below.

Table 2. Selection of sorting algorithms based on different scenarios

SCENARIO	***APPROPRIATE SORTING ALGORITHM***
When input dataset is small	Insertion sort
When input dataset is large	Heap sort, Merge sort, or Quick sort
When the data is almost sorted	Insertion sort

When the data is random	Quick sort, Merge sort, or Heap sort
When memory usage is an important consideration	Heap sort [O(1) extra space] or Quick sort [O(log n) extra space]
When linked lists have to be sorted	Merge sort
When in a parallel computing environment	Merge sort
When stability is a concern	Merge sort

6. Conclusion and Future Scope

Sorting algorithms are ultimately fundamental computer science methods with a wide range of real-world applications. They are used in many different industries and applications where large volumes of data need to be quickly and efficiently pro-cessed, categorized, and analysed. This work has addressed the several types of sorting algorithms, their space and time complexity, and their practical applications. We also examined the efficiency of sorting algorithms and discussed how com-plexity in space and time might be used to quantify it. Different sorting algorithms have different levels of efficacy, therefore it's important to choose the one that best suits the requirements of the assignment.

We also discussed performance analysis and compared the efficiency of several sorting algorithms based on a variety of cri-teria, such as stability, time complexity, and space complexity. The analysis showed that different sorting algorithms re-spond differently depending on the situation and that choosing the right algorithm might have a significant impact on per-formance. By understanding the various sorting algorithms, their efficacy, and optimization tactics, we can choose the opti-mal algorithm for the job and improve both the efficiency and performance of our apps. As technology and data collecting grow, sorting algorithms will remain an essential tool for organizing and analysing data.

Critical Ideas to think!

- How can we utilize the insertion sort's fast running time while its input is almost sorted to boost the quicksort's run-ning time?
- Is it possible to stabilize the heap sort or rapid sort algorithms?
- What is the iterative method for implementing quicksort and merge sort?
- Here are some more sorting algorithms to investigate: Tournament, Tree, Tim, and Shell sorts.

References

[1]. N. Akhter, "Sorting Algorithms – A Comparative Study," International Journal of Computer Science and Information *Security (IJCSIS)*, vol. 14, no. 12, pp. 930-936, (Published: December 2016)

[2]. G. Kocher and N. Agrawal, "Analysis and Review of Sorting Algorithms," *International Journal of Scientific Engineering and Research (IJSER)*, vol. 2, no. 3, pp. 81-84.

[3]. C. L. Liu, "Analysis of sorting algorithms," *AFIPS '73: National computer conference and exposition*, 1973.

[4]. W. Xiang, "Analysis of the time complexity of quick sort algorithm," *International Conference on Information Management, Innovation Management and Industrial Engineering,* pp. 408-410, 2011.

[5]. J. Lobo and S. Kuwelkar, "Performance analysis of merge sort algorithms," *Proceedings of the International Conference on Electronics and Sustainable Communication Systems (ICESC 2020)*, pp. 110-115, 2020

[6]. A. Aishwarya and R. Tiwari, "AN ANALYSIS OF SORTING ALGORITHMS"," *International Research Journal of Modernization in Engineering Technology and Science*, 2023.

[7]. A. Zutshi and D. Goswami, "Systematic review and exploration of new avenues for sorting algorithm"," *International Journal of Information Management Data Insights*, 2021.

[8]. S. Md, M. A. Rana, H. Hossin, and M. Jahan, "A New Approach in Sorting Algorithm"," in *8th International Congress of Information and Communication Technology*, (Published: 2019)

[9]. D. J. Mankowitz *et al.*, "Faster sorting algorithms discovered using deep reinforcement learning," *Nature*, vol. 618, no. 7964, pp. 257–263, 2023.

[10]. M. Idrees, "Sorting Algorithms - A Comparative Study," *International Journal of Computer Science and Information Security*, 2016.

[11]. A. Andersson, *Sorting and Searching Revisited"*. Sweden (Published, 2005.

[12]. J. Hammad, *A Comparative Study between Various Sorting Algorithms"*. Al-Quds Open University, 2015.

[13]. Ms ROOPA, Ms RESHMA J, "A Comparative Study of Sorting and Searching Algorithms", *International Research Journal of Engineering and Technology (IRJET)*, Vol. 05, Issue 01, 2018.

[14]. J. Clément, T. Hien Nguyen Thi, and B. Vallée, "A general framework for the realistic analysis of sorting and searching algorithms," in *30th International Symposium on Theoretical Aspects of Computer Science (STACS 2013)*, Kiel, Germany, 2013.

[15]. T. Niemann, "Sorting and Searching Algorithms: A Cookbook", July 2006.