International Journal of
INTELLIGENT SYSTEMS AND APPLICATIONS IN ENGINEERING
a ISSN:2147-6799 www.ijisae.org Original Research Paper

Sorting Algorithms Demystified: A Comprehensive Guide with the ISort Visualizer

Vandana Gandotra[1], Sakshi Taaresh Khanna[2*], Rishabh Jain[3], Rishabh Chopra[4]

Submitted: 08/03/2024 **Revised:** 23/04/2024 **Accepted:** 05/05/2024

Abstract*:* Sorting algorithms are essential elements of computer science and data processing, and their comprehension is vital for programmers at all levels of expertise. This article aims to provide 'ISort,' an innovative and interactive sorting visualizer created to enhance understanding and study of different sorting algorithms."ISort" offers a unique approach to visualizing sorting algorithms, allowing users to gain a deeper understanding of how these algorithms function and how effective they are, all through an easy-to-use and intuitive interface. The 'ISort' visualizer employs diverse sorting algorithms, including bubble, insertion, selection, merge, quicksort, and others, to demonstrate their contrasting execution and performance characteristics. A visualizer tool allows users to modify the sizes of the input data, the speed at which the sorting occurs, and the visual signals to observe the process in a step-by-step manner and gain a fundamental understanding of the workings of any algorithm. The application also provides users real-time performance indicators, such as comparison count and time complexity, to help them accurately evaluate and compare different sorting methods. This paper not only discusses the technical features of 'ISort,' including its architecture and implementation but also emphasizes its pedagogical and practical importance. 'ISort' endeavors to be a beneficial resource for educators, students, and developers who wish to increase their comprehension of sorting algorithms. Additionally, it provides a platform for the exploration and experimentation of novel algorithms. 'ISort' enhances the accessibility and engagement of sorting algorithms for a wider audience by integrating an informative and user-friendly interface with extensive algorithm coverage.

Keywords: *Sorting algorithms, analysis*

1. Introduction

Sorting algorithms play a crucial role in computer science by facilitating efficient organization, retrieval, and data processing. Sorting is a key concept in the field of algorithms, encompassing tasks ranging from organizing files by date to sorting playlists by song title. It is applicable in various domains, including databases and search engines. Algorithms play a fundamental role in computer science education, as they impart crucial concepts, problem-solving techniques, and algorithmic reasoning. Although sorting algorithms are typically uncomplicated, comprehending and visualizing them can prove challenging.

The authors present 'ISort', a cutting-edge and interactive sorting visualizer that serves as a dynamic and instructional tool for examining a wide range of sorting algorithms, bridging the divide between abstract ideas and practical comprehension. 'ISort' allows users to acquire knowledge and study sorting algorithms in a captivating and enlightening fashion.

1 Department of Computer Science, Ram Lal Anand College, University of Delhi – 110021, INDIA.
e-mail: vandanagandotra17@gmail.com
*2*Department of Computer Science, Ram Lal Anand College, University of Delhi – 110021, INDIA.*
** Corresponding Author Email: sakshitaareshkhanna@gmail.com*
ORCID ID:0009-0006-2936-8708
ORCID ID: 0009-0003-0206-9220
3 Department of Computer Science, Ram Lal Anand College, University of Delhi – 110021, INDIA.
e-mail: rishabh4124@rla.du.ac.in
ORCID ID: 0009-0003-8312-9781
4Department of Computer Science, Ram Lal Anand College, University of Delhi – 110021, INDIA.
e-mail: rishabh4087@rla.du.ac.in
ORCID ID: 0009-0008-3084-7495

1.1. Key Features of 'ISort'

Customizable Input: A fundamental aspect of 'ISort' is its capacity to enable users to choose the number of bars to be sorted. By allowing users to customize, this feature allows for experimentation with sorting algorithms on datasets of varying sizes, facilitating a deeper understanding of the algorithms' performance in varied contexts.

Best-Case and Worst-Case Scenarios: 'ISort' allows users to generate scenarios representing the best and worst-case possible outcomes. Through observation of sorting algorithms operating in ideal and difficult conditions, users can better understand the algorithms' efficiency and flexibility.

Visual Speed Control: Sorting algorithms frequently manifest as a jumble of motion. This is remedied by 'ISort's' visual speed adjustment feature, which grants users command over the sorting procedure's velocity. This functionality enables a more comprehensive examination of algorithmic procedures.

Pause and Play Functionality: Functionality for Pausing and Playing: To optimize the learning experience, 'ISort' incorporates pause and play controls that allow users to suspend the visualization while sorting temporarily. The inclusion of the pause-and-play feature allows users to analyze sorting steps in detail and acquire a more profound comprehension of their mechanisms.

Pseudo Code Highlighting: The depiction of pseudocode makes it easier to comprehend sorting algorithms. This is accomplished through the use of pseudocode highlighting. The 'ISort' program not only displays the code of the algorithm, but it also highlights the line that is now being executed. This makes it much simpler for users to comprehend the reasoning behind the calculation.

Performance Metrics: The sorting process provides real-time data on the number of comparisons conducted, which gives

significant insights into the efficiency and complexity of the selected method.

Algorithms Overview: The 'ISort' repository is a comprehensive collection of sorting algorithms that includes a wide range of algorithms such as bubble sort, insertion sort, selection sort, merge sort, quicksort, and many more. Every algorithm is accompanied by a concise explanation, making it suitable for students of all skill levels, from novices to seasoned programmers.The integration of these characteristics in 'ISort' successfully closes the gap between academic comprehension and the practical application of sorting algorithms. 'ISort' is a powerful tool that provides an intuitive, educational, and immersive approach to visualizing sorting algorithms. It is suitable for educators looking to demonstrate sorting algorithms engagingly, students seeking to understand the complexities of these algorithms, and developers interested in experimenting with and evaluating different sorting methods.This paper explores the technical architecture, design, and instructional importance of 'ISort,' offering profound insights into its development and its transformational potential for computer science education, research, and practical implementation. The 'ISort' algorithm not only enhances the ease of use and understanding of sorting algorithms, but also motivates a new cohort of algorithm aficionados to delve into the captivating realm of data management and algorithmic problem-solving.
The subsequent sections of this research paper explore sorting algorithms and the development of "ISort - The Sorting Visualizer." The literature review provides an overview of existing research on sorting algorithms and visualization tools. The structure of ISort is outlined, followed by a comparison with other tools. The implementation details of ISort are discussed, along with code snippets for clarity. Performance evaluation analyzes sorting algorithms using ISort, offering empirical insights. Finally, the paper concludes with findings, implications, and references.

2. Literature Survey

In the course of the literature evaluation, a scarcity of pertinent online resources and applications was observed. The Sorting Visualiser and Sorting Algorithm Visualiser websites, which are accessible via Google, have the functionality to efficiently represent and sort data or bar values using different sorting algorithms like bubble sort, insertion sort, and many more. It uses a basic base page to show the desired output.Sort Visualiser is a website that has been registered by**mahfuzarifat7**on GitHub. It offers users the ability to pause and lay the Sorting algorithm at their discretion, as well as the capability to adjust the speed of the visualization either in starting of the process or after the ending of the desired output [7].Sorting Visualization, an additional website created by Clement Mihail is characterized by a convoluted structure and an interface that is not intuitive; it permits the user to enter the number of bar values, albeit in a laborious fashion. [12].
The alternative visualization tool, which sorts the algorithm and is also accessible on YouTube, employs sound effects to encourage user interaction. However, users may become confused by the noises associated with bar position adjustments, merging, and swapping.The remaining two sorting visualizers include “sorting visualizers using javascript by Abhishek Prakash”[10] and “visualizing sorter by Code Drifter”[13] exclusively available on YouTube. They employ an inverted axis format to deterministically sort the bars of value and use limited color to differentiate between them during sorting, swapping, and merging.Upon reviewing the literature, it was discovered that there are no existing applications or websites that have all the necessary elements to qualify as a comprehensive tool for visualizing sorting algorithms. The authors' primary goal was to create a sorting tool called "Isort - The Sorting Visualizer."

3. Structure of “Isort - The Sorting Visualizer”

3.1. Software and Hardware Requirements

Hardware Prerequisites - A laptop or desktop computer equipped with a reliable internet connection.The software utilized during the development of the application included:Operating System: Windows 11 Platform: Visual Studio Programming instructions or commands written in a specific language.
Programming Language: JavaScript. User Interface - HTML, CSS, and Javascript.

3.2 Understanding the structure with following diagram

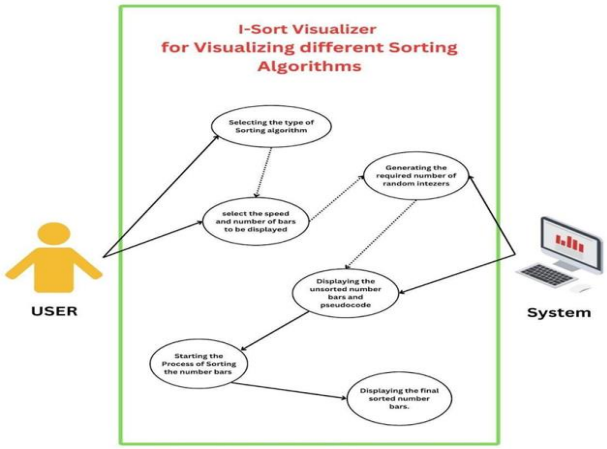

Fig.1. Use Case for ISort

The "Isort - The Sorting Visualizer" website starts with an impressive homepage that provides information about the website and highlights the benefits of sorting in the computer and algorithms fields. The website has a side panel and a feedback form, as well as the email address of the authors in the footers.

Fig. 2. ISort sorting visualizer home page

The website's side panel displays many sorting algorithms that the user can select. Each sorting algorithm is accompanied by detailed information, including its functionality, applications, and drawbacks. This information is presented in a well-organized manner under the functioning section of the website.

Fig. 3.ISort sorting visualizer side navigation

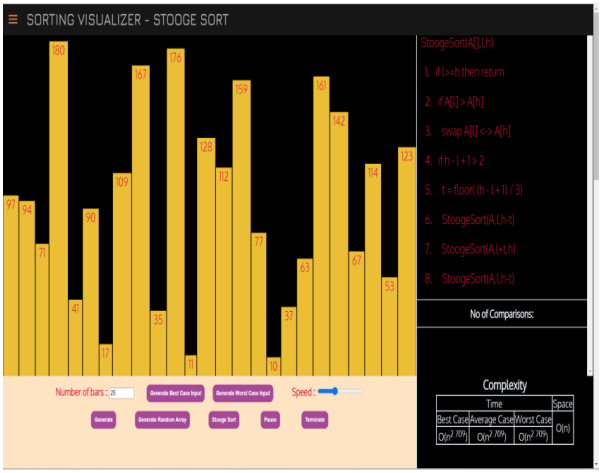

Fig. 4.ISort sorting visualizer sorting algorithm page

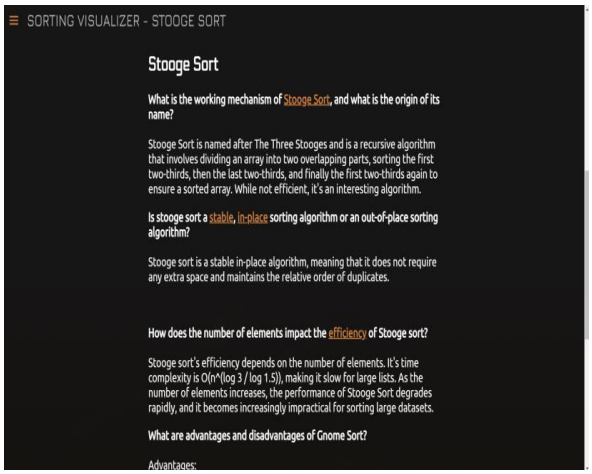

Fig. 5. ISort sorting visualizer sort description provided with each sort included

Additionally, developers offer a feedback form tha allows users to provide input, enabling them to make necessary modifications to enhance the user-friendliness and convenience of the website. The feedback includes-

1. User's name
2. Age
3. Email address
4. Rating (in form of emojis)
5. Message .

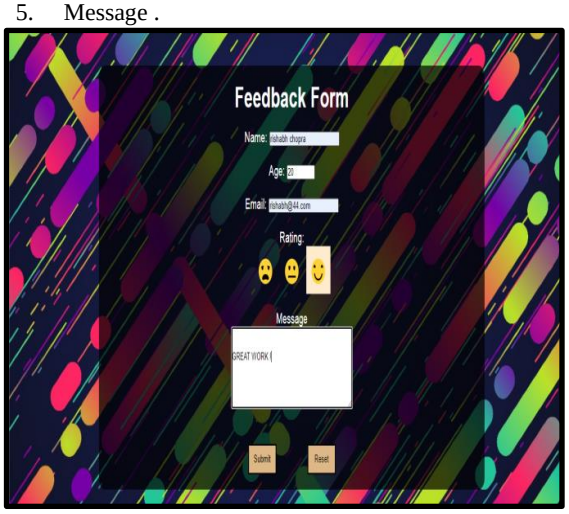

Fig. 6. ISort sorting visualizer feedback form

3.2. Comparing " Isort - The Sorting Visualizer ''with existing websites and tools

The **"Isort - The Sorting Visualizer"**is compared with other freely available Websites and sorting visualizers on Google, chrome, YouTube etc, on the basis of different parameters is presented in Table 1.

Name: This column simply lists the names of the sorting algorithm visualizer tools.

Complexity: This column indicates the complexity level of each tool. Tools are categorized as "Simple," "Moderately simple," or "Very simple" based on their user interface and features.

UI (User Interface): This column specifies whether the tool has a user interface. Tools marked "Yes" have a user interface, while those marked "No" do not.

Visual Speed Change option: This column indicates whether the tool allows users to adjust the speed of the visualization. If "Yes," users can typically speed up or slow down the sorting process visually.

Pause Play Button: This column denotes whether the tool provides a pause/play button, allowing users to pause and resume the sorting visualization.

Input number of Bars: Indicates whether users can input the number of bars (elements) to be sorted.

Random bars generation: Specifies if the tool offers the functionality to generate random bars for sorting.

Input bar values: This column indicates whether users can manually input values for the bars to be sorted.

Code showcase: Specifies whether the tool showcases the sorting algorithm's code.

Highlight Code line being executed: Indicates whether the tool highlights the line of code currently being executed during the sorting process.

Showcase number of comparisons: Specifies if the tool displays the number of comparisons made during the sorting process.

Termination button: Denotes whether the tool provides a button or option to terminate the sorting process prematurely.

Time complexity: Indicates whether the tool provides information about the time complexity of the sorting algorithm being visualized.

Best case and worst case: Specifies whether the tool showcases information about the best and worst-case scenarios for the sorting algorithm's performance.

User friendly: This column evaluates the overall user-friendliness of the tool.

Table 1. Detailed comparison of ISORT with other existing models

Name	UI	Visual Speed Change option	Pause Play Button	Input the number of Bars	Random bars generation	Input bar values	Code showcase	Highlight the Code line being executed	Showcase the number of comparisons	Termination button	Time complexity	Best-case and worst-case	User friendly
Visualgo.net	Complex	Yes	Yes	No	Yes	Yes	Yes	Yes	No	No	No	Yes	No
Sorting algorithms visualized by Kyle Smith	Simple	Yes	No	Yes	Yes	No	No	No	No	No	No	No	No
Sorting Algorithm Visualizer	Simple	Yes	No	Yes	Yes	No	No	No	No	No	No	No	No
clement mihaile sorting visualizer	Simple	Yes	No	Yes	Yes	No	No	No	No	No	No	No	No
mahfuzrifar7 's sorting visualizer	Simple	Yes	No	Yes	Yes	No	No	No	No	No	No	No	Yes
ramizrahman sorting-algorithms	Simple	Yes	Yes	Yes	Yes	No	No	No	No	No	Yes	No	Yes
coder stool	Moderately simple	Yes	No	Yes	Yes	No	No	No	No	No	No	Yes	Yes
Radu mariescu sorting visualizer	very simple	No	No	No	Yes	No	No	No	No	Yes	No	No	Yes
coder drift's sorting visualizer	Moderately simple	Yes	No	Yes	Yes	No	No	No	No	No	No	No	No
Abhishek Prakash sorting visualizer	Simple	Yes	No	No	Yes	No	No	No	No	No	No	No	Less
Isort Visualizer	Simple	Yes	Yes	Yes	Yes	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes

4. Implementation of ISort - The Sorting Visualizer

The development of 'iSort' is driven by a user-centric approach, prioritizing interactivity, customization, and educational significance. Users can provide the desired number of data components to be sorted, and 'iSort' will dynamically generate a visual dataset that corresponds to this input. 'iSort' possesses a notable characteristic of being capable of producing optimal and suboptimal situations. This is accomplished by adjusting the original data distribution and utilizing a collection of pre-established datasets. The program has a customizable visual speed slider, enabling users to modify the animation speed in real-time. This allows users to freely explore sorting algorithms at their preferred pace. The pause and play functionality is smoothly incorporated, enabling users to halt the sorting animation at any moment, thus providing a detailed examination of the algorithm's execution in a step-by-step manner. Moreover, 'iSort' ensures that the pseudo code of the algorithm is synchronized with the ongoing execution and emphasizes the currently executing line, hence improving user understanding. Users are provided with useful insights on algorithm efficiency during the sorting process through rigorous tracking of real-time performance measures, including comparison counts. The extensive use of 'iSort' distinguishes it as a versatile and user-friendly platform for the investigation and comprehension of sorting algorithms.

Compared to competing programs, 'iSort' distinguishes itself with its unique combination of proven skills. The 'iSort' application offers a range of user-driven capabilities, such as the ability to customize input, design scenarios, manage the speed visually, and smoothly pause and play. Additionally, it incorporates proven elements such as highlighting pseudo code and providing real-time performance data. The utilization of this technology ensures that 'iSort' delivers a dynamic and all-encompassing user experience, setting it apart from other visualization tools designed for sorting algorithms. The meticulous design of 'iSort' renders it an appealing and informative tool suitable for a diverse audience, including students, educators, programmers, and algorithm enthusiasts.

4.1Glimpse of implementation of the application

Fig. 7. Home Page

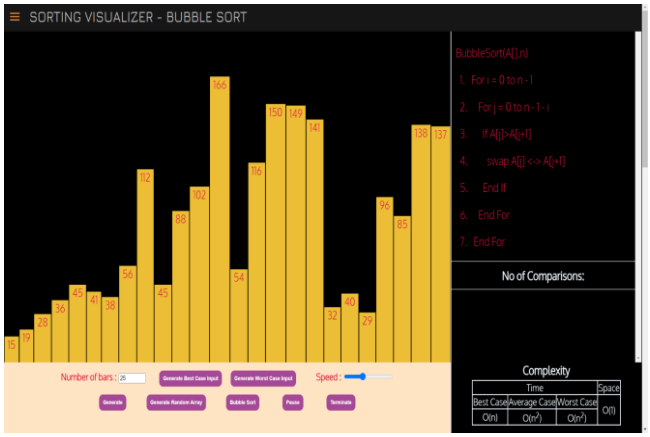

Fig. 8. Bubble Sort

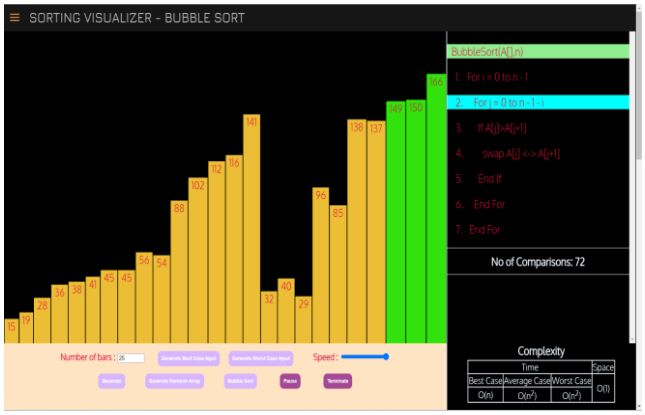

Fig. 9. Sort in process

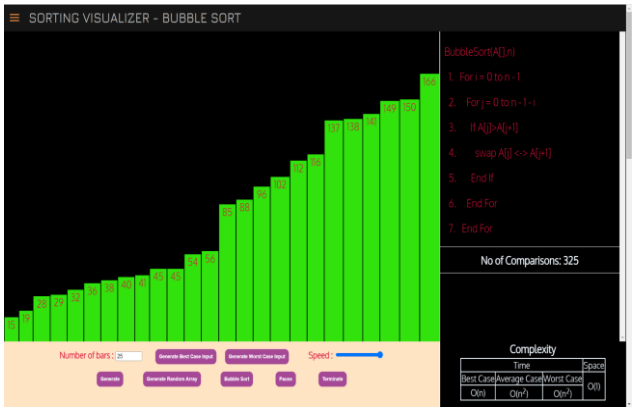

Fig. 10. Completed sort

Fig. 2 displays the Home page of the iSort website, which allows the user to select a sorting option. Fig. 3 displays the user interface (UI) of Bubble sort, showcasing the control panel, pseudocode, comparison table, and bars with their respective values.

Fig.4 depicts an ongoing sorting process, with the highlighted pseudocode indicating the current line of code being executed. Fig. 5 displays a fully sorted arrangement.

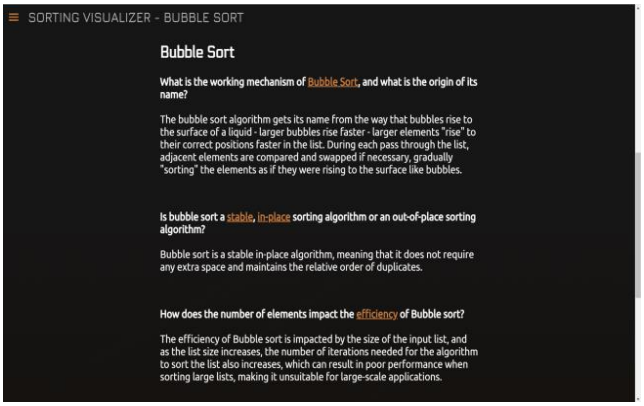

Fig. 11. Brief of a sort

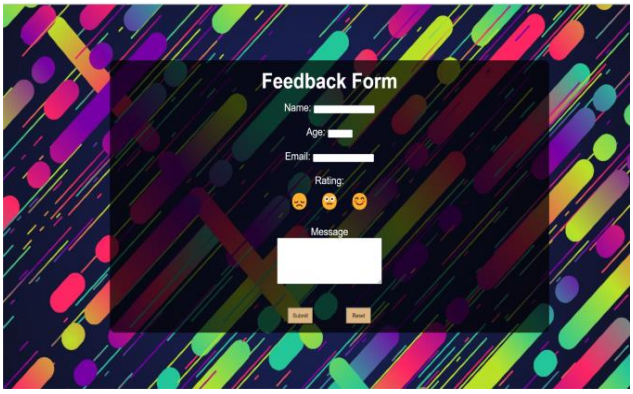

Fig. 12. Feedback Form

Fig. 13. Sidebar showing sorts

Fig.6 depicts a concise overview of a sorting algorithm. Fig.7 displays the user interface (UI) of the feedback form.Fig. 8 displays the side navigation bar that exhibits several types of sorts implemented by ISort.

4.2 Application Flow

Upon initiation of ISort, the user will be promptly redirected to the platform's homepage. Users have two primary navigation choices: they may either scroll down to find the "Let's Sort Away!" button at the bottom of the page, or they can access the side navigation bar by touching the three-line symbol at the top-left corner of the interface. The side navigation bar offers customers an extensive selection of sorting algorithms, enabling them to choose the particular method they want to investigate. Once the user chooses a specific sorting method, they are automatically redirected to a specialized page specifically designed for that algorithm.A conspicuous control panel is prominently showcased on the page dedicated to a specific sorting algorithm. This control panel offers a background with a distinctive skin color and houses all the essential tools for altering the array of data and regulating the sorting process itself. To begin the sorting process, users can tap on the button labeled with the name of the chosen sorting method. This will trigger a dynamic visualization where bars are sorted. After the sorting process is finished, the algorithm's execution updates and displays the number of comparisons made. This information is located right below the portion that contains the algorithm's pseudocode, giving users useful information about the algorithm's efficiency. Additionally, visitors are encouraged to submit feedback and suggestions via a readily accessible link situated at the bottom of the page, generating a sense of user engagement and continual progress.

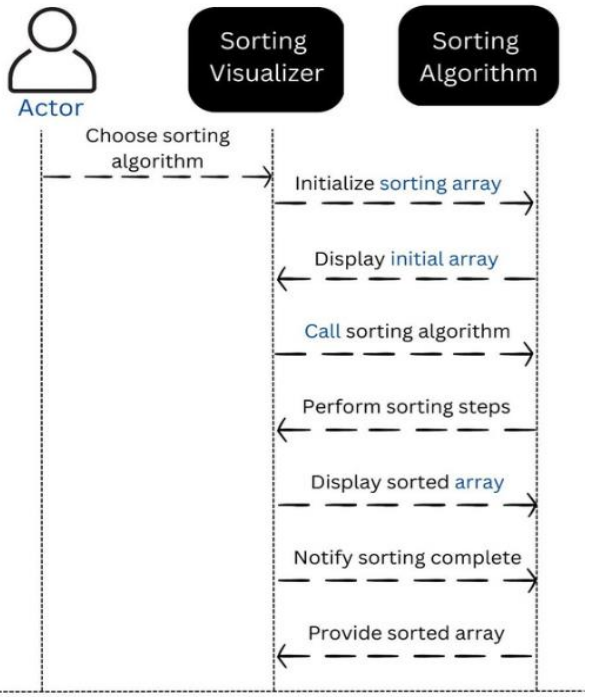

Fig. 14. Sequence Diagram

Upon entering the website, the user can choose from a variety of sorting algorithms based on their specific needs. This selection triggers the website to open the chosen sorting algorithm, allowing the user to customize parameters like the number of elements, algorithm speed, and specific elements within the algorithm. The initial array is then presented based on these adjustments in the form of number bars. Upon the user's initiation of the sorting algorithm, the website proceeds to execute the sorting process in the background. Simultaneously, it visually guides the user through each step by highlighting the currently executing line of code (pseudo code) and the corresponding number bar involved in the process. Once the array is successfully sorted, the final result is presented along with the total number of comparisons made during the process.

5. Code Snippets

5.1 Code snippet for Bubble Sort backend working

This JavaScript code defines functions for generating and visualizing bars on a webpage, particularly for sorting algorithms. The user can generate number bars randomly, in the best-case scenario, or the worst-case scenario. This module includes the Bubble Sort algorithm, with visualizations for each step, such as bar comparisons and swaps. Similar modules are available for other sorting algorithms. Buttons allow the user to control the sorting process, including pausing and resuming. Additionally, there are options to set the speed of the visualization. The code utilizes HTML elements and styling to create a dynamic and interactive sorting visualization on a webpage using different color highlighting methods.

```
const container = document.querySelector(".vrsplit1");
let isPlaying = false;
const pausePlayBtn = document.getElementById("pauseButton");
let terminate = false;
var count = 0;
const compare = document.getElementById("comp");
function generatebars(num) {
container.innerHTML=""
   for (let i = 0; i< num; i += 1) {
                const value = Math.floor(Math.random() * (180-9)+9) + 1;
                const bar = document.createElement("div");
            bar.classList.add("bar");
            bar.style.height = `${value/2}%`;
                        const barLabel = document.createElement("label");
                        barLabel.classList.add("bar__id");
            barLabel.innerHTML = value;
       if (num>80) {
barLabel.style.display='none';
       }
                        if (num<=40) {
                                if (num<=10) {
                                        barLabel.style.fontSize = 'xxx-large';
                                }
          else if (num<=20) {
barLabel.style.fontSize = 'xx-large';
          }
          if (num>20 && num<=30) {
                                        barLabel.style.fontSize = 'x-large';
                                }
          else if (num<=40) {
barLabel.style.fontSize = 'large';
          }
                        }
                        bar.appendChild(barLabel);
            container.appendChild(bar);
   }
}
generatebars(25);
function generate() {
            var n = document.getElementById("nele");
            var numele = parseInt(n.value);
   if (numele>400) {
window.alert("Upper bound is 400 bars. Kindly choose a value in that range!");
n.value=400;
generate();
   }
   else {
generatebars(numele);
   }
}
function generate2() {
            const value = Math.floor(Math.random() * 80) + 1;
            generatebars(value);
}
function generatebarsWorst(num) {
container.innerHTML=""
   let values = Array.from({length: num}, (_, i) =>Math.max(num - i + 9, 10));
   for (let i = 0; i< num; i += 1) {
      const value = values[i];
                const bar = document.createElement("div");
            bar.classList.add("bar");
            bar.style.height = `${value/2}%`;
                        const barLabel = document.createElement("label");
                        barLabel.classList.add("bar__id");
            barLabel.innerHTML = value;
       if (num>80) {
barLabel.style.display='none';
       }
                        if (num<=40) {
```

```
                                    if (num<=10) {
                                                barLabel.style.fontSize = 'xxx-large';
                                    }
        else if (num<=20) {
barLabel.style.fontSize = 'xx-large';
        }
        if (num>20 && num<=30) {
                                                barLabel.style.fontSize = 'x-large';
                                    }
        else if (num<=40) {
barLabel.style.fontSize = 'large';
        }
                        }
                        bar.appendChild(barLabel);
            container.appendChild(bar);
   }
}
function generatebarsBest(num) {
container.innerHTML=""
   let values = Array.from({length: num}, (_, i) =>i + 1 + 9);
   for (let i = 0; i< num; i += 1) {
      const value = values[i];
            const bar = document.createElement("div");
            bar.classList.add("bar");
            bar.style.height = `${value/2}%`;
                        const barLabel = document.createElement("label");
                        barLabel.classList.add("bar__id");
            barLabel.innerHTML = value;
      if (num>80) {
barLabel.style.display='none';
      }
                        if (num<=40) {
                                    if (num<=10) {
                                                barLabel.style.fontSize = 'xxx-large';
                                    }
        else if (num<=20) {
barLabel.style.fontSize = 'xx-large';
        }
        if (num>20 && num<=30) {
                                                barLabel.style.fontSize = 'x-large';
                                    }
        else if (num<=40) {
barLabel.style.fontSize = 'large';
        }
                        }
                        bar.appendChild(barLabel);
            container.appendChild(bar);
   }
}
function generatebest() {
   const value = Math.floor(Math.random() * 80) + 1;
generatebarsBest(value);
}
function generateworst() {
   const value = Math.floor(Math.random() * 80) + 1;
generatebarsWorst(value);
}
function disable() {
document.getElementById("Button1").disabled = true;
document.getElementById("Button1").style.backgroundColor = "#d8b6ff";
document.getElementById("Button2").disabled = true;
document.getElementById("Button2").style.backgroundColor = "#d8b6ff";
document.getElementById("Button3").disabled = true;
document.getElementById("Button3").style.backgroundColor = "#d8b6ff";
document.getElementById("Button4").disabled = true;
document.getElementById("Button4").style.backgroundColor = "#d8b6ff";
document.getElementById("Button5").disabled = true;
document.getElementById("Button5").style.backgroundColor = "#d8b6ff";
}
var delay = 5000;
```

```
async function BubbleSort() {
    count = 0;
    let bars = document.querySelectorAll(".bar");
    var l0 = document.getElementById("line0");
    l0.style.backgroundColor = "lightgreen";
    for (let i = 0; i<bars.length; i++) {
        var l1 = document.getElementById("line1");
        l1.style.backgroundColor = "cyan";
        await new Promise((resolve) =>setTimeout(() => { resolve(); }, delay));
        l1.style.backgroundColor = null;
        var c2 = 1;
        for (let j = 0; j <bars.length - i - 1; j++) {
            var l2 = document.getElementById("line2");
            l2.style.backgroundColor = "cyan";
            while (isPlaying) {
                if (terminate) {
                    l2.style.backgroundColor = null;
                    l0.style.backgroundColor = null;
document.getElementById("Button1").disabled = false;
document.getElementById("Button1").style.backgroundColor = "#a54997";
document.getElementById("Button2").disabled = false;
document.getElementById("Button2").style.backgroundColor = "#a54997";
document.getElementById("Button3").disabled = false;
document.getElementById("Button3").style.backgroundColor = "#a54997";
document.getElementById("Button4").disabled = false;
document.getElementById("Button4").style.backgroundColor = "#a54997";
document.getElementById("Button5").disabled = false;
document.getElementById("Button5").style.backgroundColor = "#a54997";
isPlaying = false;
pausePlayBtn.textContent = 'Pause';
                        compare.textContent=' ' + " No of Comparisons: ";
                    for (let k=0;k<bars.length;k++) {
                        bars[k].style.backgroundColor="rgb(236, 190, 53)"; }
                    terminate = !terminate;
                    return; }
                await new Promise((resolve) =>setTimeout(() => { resolve(); }, 1000));  }
            if (terminate) {
                l2.style.backgroundColor = null;
                l0.style.backgroundColor = null;
document.getElementById("Button1").disabled = false;
document.getElementById("Button1").style.backgroundColor = "#a54997";
document.getElementById("Button2").disabled = false;
document.getElementById("Button2").style.backgroundColor = "#a54997";
document.getElementById("Button3").disabled = false;
document.getElementById("Button3").style.backgroundColor = "#a54997";
document.getElementById("Button4").disabled = false;
document.getElementById("Button4").style.backgroundColor = "#a54997";
document.getElementById("Button5").disabled = false;
document.getElementById("Button5").style.backgroundColor = "#a54997";
                for (let k=0;k<bars.length;k++) {
                    bars[k].style.backgroundColor="rgb(236, 190, 53)"; }
                        compare.textContent=' ' + " No of Comparisons: ";
                terminate = !terminate;
                return; }
            var value1 = parseInt(bars[j].childNodes[0].innerHTML);
            var value2 = parseInt(bars[j + 1].childNodes[0].innerHTML);
            var l3 = document.getElementById("line3");
            l3.style.backgroundColor = "cyan";
            if (value1 > value2) {
                await new Promise((resolve) =>setTimeout(() => { resolve(); }, delay));
                var l4 = document.getElementById("line4");
                l4.style.backgroundColor = "cyan";
                bars[j].style.backgroundColor = "red";
                        await new Promise((resolve) =>setTimeout(() => { resolve(); }, delay));
bars[j + 1].style.backgroundColor = "red";
                var temp1 = bars[j].style.height;
                var temp2 = bars[j].childNodes[0].innerText;
                await new Promise((resolve) =>setTimeout(() => { resolve(); }, delay));
                bars[j].style.height = bars[j + 1].style.height;
                bars[j].childNodes[0].innerText = bars[j + 1].childNodes[0].innerText;
```

```
bars[j + 1].style.height = temp1;
bars[j + 1].childNodes[0].innerText = temp2;
            l4.style.backgroundColor = null;   }
        c2 = c2+1;
        var l5 = document.getElementById("line5");
        l5.style.backgroundColor = "cyan";
        l3.style.backgroundColor = null;
        await new Promise((resolve) =>setTimeout(() => { resolve(); }, delay));
        l5.style.backgroundColor = null;
        bars[j].style.backgroundColor = "rgb(236, 190, 53)";
bars[j + 1].style.backgroundColor = "rgb(236, 190, 53)";
        await new Promise((resolve) =>setTimeout(() => { resolve(); }, delay));
        l2.style.backgroundColor = null;       }
      count = count + c2;
           compare.textContent=' ' + " No of Comparisons: "+count;
      var l6 = document.getElementById("line6");
      l6.style.backgroundColor = "cyan";
bars[bars.length - i - 1].style.backgroundColor = "rgb(49, 226, 13)";
      await new Promise((resolve) =>setTimeout(() => { resolve(); }, delay));
      l6.style.backgroundColor = null;    }
   var l7 = document.getElementById("line7");
   l7.style.backgroundColor = "cyan";
   for (let i = 0; i<bars.length; i++) {
      bars[i].style.backgroundColor = "rgb(49, 226, 13)";    }
   await new Promise((resolve) =>setTimeout(() => { resolve(); }, delay));
   l7.style.backgroundColor = null;
document.getElementById("Button1").disabled = false;
document.getElementById("Button1").style.backgroundColor = "#a54997";
document.getElementById("Button2").disabled = false;
document.getElementById("Button2").style.backgroundColor = "#a54997";
document.getElementById("Button3").disabled = false;
document.getElementById("Button3").style.backgroundColor = "#a54997";
document.getElementById("Button4").disabled = false;
document.getElementById("Button4").style.backgroundColor = "#a54997";
document.getElementById("Button5").disabled = false;
document.getElementById("Button5").style.backgroundColor = "#a54997";
compare.textContent=' ' + " No of Comparisons: "+count;
   await new Promise((resolve) =>setTimeout(() => { resolve(); }, delay));
   l0.style.backgroundColor = null;}
function delaySet() {
   delay = 5000;
   var s = document.getElementById("speeder");
   var d = parseInt(s.value);
   delay=delay/d;}
pausePlayBtn.addEventListener('click', () => {
   if (isPlaying) {
isPlaying = false;
pausePlayBtn.textContent = 'Pause';
   } else {
isPlaying = true;
pausePlayBtn.textContent = 'Resume';  }
  });
```

This JavaScript code segment constructs a sorting visualizer specifically tailored for the Bubble Sort algorithm. Upon initialization, it defines essential variables such as container, isPlaying, and pausePlayBtn to facilitate sorting operations and user interaction. Functions are then established to generate bars for visualization (generatebars), control animation speed (delaySet), and execute the Bubble Sort algorithm (BubbleSort). The heart of the functionality lies within the BubbleSort function, where a loop iterates through the bars, assessing adjacent elements and executing swaps as necessary until the array is sorted. Throughout this process, visual cues like color changes in bars and highlighted lines signify the algorithm's progression. Additionally, an event listener is incorporated to the pausePlayBtn button, allowing users to toggle between pausing and resuming the sorting animation. Furthermore, the delaySet function permits users to adjust the animation speed by modifying the delay between iterations of the sorting algorithm. This code segment provides an interactive platform for visualizing Bubble Sort, enhancing comprehension and engagement with the sorting procedure.

5.2 Code snippet for Feedback form

This HTML document represents a feedback form webpage. It includes a form with fields for name, age, email, a rating using emojis, and a message. The embedded JavaScript function validateForm() checks for the correctness of input data, including name validation with a regular expression for disallowing numbers. The form is styled using an external CSS file and has a background image. Upon submission, the form is set to be sent via email to specified recipients using the "mailto" attribute.

```
<!DOCTYPE html>
<html>
<head>
<title>
        Feedback Form
</title>
<link rel="stylesheet" href="feedbackcss.css">
<script>
    function validateForm() {var name = document.forms["feedbackForm"]["name"].value;
        var age = document.forms["feedbackForm"]["name"].value;
        var email = document.forms["feedbackForm"]["email"].value;
        var message = document.forms["feedbackForm"]["message"].value;
        var regEmail = /^\w+([\.-]?\w+)*@\w+([\.-]?\w+)*(\.\w{2,3})+$/g;  //JavascriptreGex for Email Validation.
        var regName = /\d+$/g;                          // Javascript reGex for Name validation
        if (name == "" || regName.test(name)) {alert("Name must be filled out");
            return false;   }
        if (age == "") {alert("Name must be filled out correctly");
            return false;       }
        if (email == "" || !regEmail.test(email)) {
alert("Email must be filled out correctly");
            return false;       }
        if (message == "") {
alert("Message must be filled out");
            return false;   }
        return true;    }
</script>
</head>
<body background="bg.jpg" style="background-repeat:no-repeat; background-size:100% 100vh;">
<center>
<div id="div1" style="background-color: rgba(0, 0, 0, 0.7)">
<h1 font-size : 70px style="margin:20px;">Feedback Form</h1>
<form name="feedbackform" action="mailto: rishabh4124@rla.du.ac.in; rishabh4087@rla.du.ac.in" method="post" onsubmit="return
validateForm()" enctype="text/plain">
<label for="name">Name:</label>
<input type="text" id="name" name="name" required><br><br>
<label for"age">Age:</label>
<input type="number" min="15" max="100" name="Age" required><br><br>
<label for="email">Email:</label>
<input type="email" id="email" name="email" required><br><br>
<label for="" text-align:left>Rating:</label>
<div id="response">
<input type="radio" id="sad" name="response" value="sad" class="radio" />
<label class="emoji emoji1" for="sad">&#128542</label>
<input type="radio" name="response" id="average" value="average" class="radio" />
<label class="emoji emoji2" for="average">&#128528</label>
<input type="radio" name="response" id="happy" value="happy" class="radio" />
<label class="emoji emoji3" for="happy">&#128522</label>
</div>
<br>
<label id="label1" for="message">Message</label><br>
<textarea id="message" name="message" rows="7" cols="40"></textarea><br><br>
<input type="submit" value="Submit" class="button">
     
<input type="reset" value="Reset" class="button">
</form>
</div>
</center>
</body>
</html>
```

The provided HTML code constitutes a feedback form designed to collect user input. The form begins with the declaration of the document type and version of HTML being used. Within the `<head>` section, metadata such as the webpage title and a link to an external CSS file for styling purposes are specified. Additionally, JavaScript code is embedded within the `<script>` tags to validate user inputs before submission. The validation function, `validateForm()`, checks that the name, age, email, and message fields are not empty and meet specific criteria defined by regular expressions. Moving on to the `<body>` section, the background image of the webpage is set, and the content is centered using the `<center>` tag. The feedback form itself is contained within a `<div>` element with a semi-transparent black background. It includes input fields for name, age, email, rating, and message, along with submit and reset buttons. Upon submission, the form data is sent via email to the specified recipients. Overall, the HTML code constructs a visually appealing and functional feedback form with client-side validation to ensure the accuracy of user-provided information before submission.

6. Performance Evaluation of different sorting algorithms.

For evaluating the performance of various sorting algorithms, the authors conducted comprehensive tests by considering different input sizes and types. The algorithms' performance was assessed under varying scenarios, including best and worst-case scenarios for the same input size. This evaluation involved noting the number of comparisons made by each algorithm and determining their respective time complexities. Our analysis encompassed a range of sorting algorithms, including Bubble Sort, Merge Sort, Selection Sort, Insertion Sort, Heap Sort, Gnome Sort, and Stooge Sort. Through this systematic evaluation process, authors gained insights into the efficiency and effectiveness of each algorithm across different input scenarios.

1) Bubble Sort's performance was evaluated across different input sizes and scenarios. For an input size of 25 elements, the algorithm demonstrated its best-case scenario efficiency with only 25 comparisons required when the input was already sorted, showcasing its linear time complexity of O(n). However, in the worst-case scenario with random input, Bubble Sort required 160 comparisons, revealing its quadratic time complexity of O(n^2). As the input size increased to 50 and 100 elements, Bubble Sort's inefficiency became more pronounced, with 325 comparisons for the former and 2995 comparisons for the latter, both confirming its quadratic time complexity. While Bubble Sort is straightforward and suitable for small datasets or pre-sorted inputs, its performance significantly deteriorates for larger datasets, making it inefficient compared to more advanced sorting algorithms.

Table 2. Analysis of Bubble sort

No of inputs	No of comparisons	Type of Input	Complexity
25	25	Best Case	O(n)
25	160	Random	$O(n^2)$
25	325	Worst Case	$O(n^2)$
50	676	Random	$O(n^2)$
100	2995	Random	$O(n^2)$

Bubble sort is named for the way larger elements move up the list faster, like bubbles rising in liquid. Adjacent elements are compared and swapped during each pass, gradually sorting the list.

2) Insertion Sort reveals its performance across varying input sizes and scenarios. With an input size of 25 elements, Insertion Sort demonstrates optimal efficiency in the best-case scenario, requiring only 25 comparisons, showcasing its linear time complexity of O(n). However, in the worst-case scenario with random input, it necessitates 221 comparisons, indicating its quadratic time complexity of O(n^2). As the input size increases to 50 and 100 elements, Insertion Sort's inefficiency becomes more evident, with 433 and 2950 comparisons, respectively, further confirming its quadratic time complexity. While Insertion Sort is effective for small datasets or pre-sorted inputs, its performance diminishes significantly for larger datasets, aligning with other quadratic-time sorting algorithms.

Table 3. Analysis of Insertion sort

No of inputs	No of comparisons	Type of Input	Complexity
25	25	Best Case	O(n)
25	221	Random	$O(n^2)$
25	433	Worst Case	$O(n^2)$
50	658	Random	$O(n^2)$
100	2950	Random	$O(n^2)$

Insertion sorting is akin to sorting playing cards in your hand. The array is divided into sorted and unsorted parts. Values from the unsorted part are selected and inserted into the correct position within the sorted part.

3) Selection Sort like other quadratic-time sorting algorithms, exhibits varying performance depending on the input size and characteristics. For an input size of 25 elements, Selection Sort showcases its best-case scenario efficiency with only 25 comparisons needed, reflecting its linear time complexity of O(n). However, in the worst-case scenario with random input, the Selection Sort requires 379 comparisons, revealing its quadratic time complexity of O(n^2). As the input size increases to 50 and 100 elements, the inefficiency of Selection Sort becomes more pronounced, with 668 and 2894 comparisons respectively, further confirming its quadratic time complexity. While Selection Sort may be suitable for small datasets or educational purposes due to its simplicity, its performance diminishes significantly for larger datasets, making it less practical for real-world applications compared to more efficient sorting algorithms.

Table 4. Analysis of Selection sort

No of inputs	No of comparisons	Type of Input	Complexity
25	25	Best Case	O(n)
25	379	Random	$O(n^2)$
25	480	Worst Case	$O(n^2)$
50	668	Random	$O(n^2)$
100	2894	Random	$O(n^2)$

Selection sort is a basic sorting algorithm that iteratively locates the minimum element from the unsorted portion of an array and positions it at the start of the sorted segment.

4) Quick Sort demonstrates efficient performance across different input sizes and scenarios, showcasing its effectiveness as a sorting algorithm. For an input size of 25 elements, Quick Sort exhibits optimal efficiency in both the best-case and random scenarios, requiring 200 and 251 comparisons respectively. This reflects its average-case time complexity of O(n*logn), where 'n' represents the size of the input. However, in the worst-case scenario with random input, Quick Sort necessitates 505 comparisons, indicating its potential to degrade to quadratic time complexity (O(n^2)). As the input size increases to 50 and 100 elements, Quick Sort maintains its efficiency, with 524 and 1184 comparisons respectively in random scenarios, reinforcing its average-case time complexity. Quick Sort's ability to achieve near-optimal performance in average and best-case scenarios makes it a preferred choice for sorting large datasets efficiently, especially when compared to quadratic-time sorting algorithms like Bubble Sort or Selection Sort.

Table 5. Analysis of Quick sort

No of inputs	No of comparisons	Type of Input	Complexity
25	200	Best Case	O(n(logn))
25	251	Random	O(n(logn))
25	505	Worst Case	$O(n^2)$
50	524	Random	O(n(logn))
100	1184	Random	O(n(logn))

QuickSort, like Merge Sort, is a Divide and Conquer algorithm that selects a pivot and partitions the array around it. Different versions of QuickSort choose the pivot in various ways, including

the first element, the last element, a random element, or the median.

5) Randomized QuickSort exhibits efficient performance across varying input sizes and scenarios, showcasing its effectiveness as an improvement over traditional Quick Sort. For an input size of 25 elements, Randomized Quick Sort demonstrates optimal efficiency in both the best-case and random scenarios, requiring 215 and 245 comparisons respectively. This reflects its average-case time complexity of O(n*logn), where 'n' represents the size of the input. However, in the worst-case scenario with random input, Randomized Quick Sort necessitates 502 comparisons, indicating its potential to degrade to quadratic time complexity (O(n^2)). As the input size increases to 50 and 100 elements, Randomized Quick Sort maintains its efficiency, with 568 and 1097 comparisons respectively in random scenarios, reinforcing its average-case time complexity. The randomized nature of Quick Sort helps mitigate the likelihood of encountering worst-case scenarios, making Randomized Quick Sort a preferred choice for sorting large datasets efficiently while reducing the risk of performance degradation compared to traditional Quick Sort.

Table 6. Analysis of Randomized Quicksort

No of inputs	No of comparisons	Type of Input	Complexity
25	215	Best Case	O(n(logn))
25	245	Random	O(n(logn))
25	502	Worst Case	$O(n^2)$
50	568	Random	O(n(logn))
100	1097	Random	O(n(logn))

Randomized quicksort prevents worst-case time complexity by randomly choosing the pivot, mitigating issues with already sorted inputs and excessive comparisons.

6) Merge Sort consistently demonstrates efficient performance across different input sizes and scenarios, establishing itself as a reliable sorting algorithm. For an input size of 25 elements, Merge Sort exhibits optimal efficiency in both the best-case and random scenarios, requiring 118 comparisons each time. This reflects its consistent average-case time complexity of O(n*logn), where 'n' represents the size of the input. Interestingly, even in the worst-case scenario with random input, Merge Sort maintains its efficiency with only 118 comparisons needed. As the input size increases to 50 and 100 elements, Merge Sort continues to demonstrate its efficiency, with 286 and 672 comparisons respectively in random scenarios, reaffirming its average-case time complexity. The stable and predictable performance of Merge Sort makes it an excellent choice for sorting large datasets efficiently, particularly when compared to other sorting algorithms.

Table 7. Analysis of Merge sort

No of inputs	No of comparisons	Type of Input	Complexity
25	118	Best Case	O(n(logn))
25	118	Random	O(n(logn))
25	118	Worst Case	O(n(logn))
50	286	Random	O(n(logn))
100	672	Random	O(n(logn))

Merge sort divides an array into halves, sorts each half, then merges them together, following the divide-and-conquer approach.

7) Heap Sort demonstrates efficient performance across various input sizes and scenarios, making it a reliable sorting algorithm. For an input size of 25 elements, Heap Sort exhibits optimal efficiency in both the best-case and random scenarios, requiring 74 and 275 comparisons respectively. This indicates its consistent average-case time complexity of O(n*logn), where 'n' represents the size of the input. Even in the worst-case scenario with random input, Heap Sort maintains its efficiency with 290 comparisons needed. As the input size increases to 50 and 100 elements, Heap Sort continues to perform well, with 738 and 1743 comparisons respectively in random scenarios, reaffirming its average-case time complexity. The stable performance of Heap Sort across different input sizes and scenarios makes it a suitable choice for sorting large datasets efficiently, particularly when compared to other sorting algorithms.

Table 8. Analysis of Heap sort

No of inputs	No of comparisons	Type of Input	Complexity
25	74	Best Case	O(n(logn))
25	275	Random	O(n(logn))
25	290	Worst Case	O(n(logn))
50	738	Random	O(n(logn))
100	1743	Random	O(n(logn))

Heap sort transforms an unsorted array into a binary heap and repeatedly extracts the maximum element until the array is sorted.

8) Gnome Sort's performance varies across different input sizes and scenarios. For an input size of 25 elements, Gnome Sort demonstrates optimal efficiency in the best-case scenario, requiring only 25 comparisons. This reflects its linear time complexity of O(n), where 'n' represents the size of the input. However, in the worst-case scenario with random input, Gnome Sort requires 600 comparisons, indicating its potential to degrade to quadratic time complexity (O(n^2)). Similarly, for an input size of 50 elements, Gnome Sort showcases its inefficiency in random scenarios, needing 1233 comparisons. As the input size increases to 100 elements, Gnome Sort's performance further deteriorates, requiring 5250 comparisons in random scenarios. Overall, while Gnome Sort may perform adequately for small datasets or nearly sorted inputs, its efficiency diminishes significantly for larger datasets, making it less practical compared to other sorting algorithms with better time complexities.

Table 9. Analysis of Gnome sort

No of inputs	No of comparisons	Type of Input	Complexity
25	25	Best Case	O(n)
25	315	Random	$O(n^2)$
25	600	Worst Case	$O(n^2)$
50	1233	Random	$O(n^2)$
100	5250	Random	$O(n^2)$

Gnome sort is a basic sorting algorithm that swaps adjacent elements until the list is sorted. Named after a "gnome" rearranging a garden, it is more efficient than bubble sort as it moves backward through the list.

9) Stooge Sort's performance exhibits a consistent time complexity of O(n^2.709), where 'n' represents the size of the input. This complexity is reflected in all scenarios, including the best case, worst case, and random inputs, across varying input sizes. For an input size of 25 elements, Stooge Sort requires 3151 comparisons in the best-case scenario and 3280 comparisons in random scenarios, showcasing its inefficiency even for relatively small datasets. The worst-case scenario with random input requires 3654 comparisons for the same input size. As the input size increases to 50 and 100 elements, Stooge Sort's performance deteriorates further, requiring 10149 and 24743 comparisons respectively in random scenarios. Overall, Stooge Sort's time complexity indicates its inefficiency compared to more commonly used sorting algorithms, particularly for larger datasets, making it less practical for real-world applications where efficiency is crucial.

Table 10. Analysis of Stooge sort

No of inputs	No of comparisons	Type of Input	Complexity
25	3151	Best Case	$O(n^{2.709})$
25	3280	Random	$O(n^{2.709})$
25	3654	Worst Case	$O(n^{2.709})$
50	10149	Random	$O(n^{2.709})$
100	24743	Random	$O(n^{2.709})$

Stooge Sort, named after The Three Stooges, is a recursive algorithm. It divides an array into overlapping parts, sorts them in a specific order, ensuring a sorted array. Though not efficient, it's an intriguing algorithm.

7. Conclusion and Future Work

To summarize, 'iSort' is a noteworthy achievement in the display of sorting algorithms. It is a versatile tool that prioritizes the user's needs and considerably enhances the understanding and exploration of sorting strategies. 'iSort' offers a versatile input system, live performance measurements, and inventive functionalities such as scenario creation and dynamic speed adjustment. This platform provides an immersive and all-encompassing environment for anyone involved in learning, teaching, and developing sorting algorithms. The successful integration of established features with unique components in this tool is highlighted by a comparative analysis with existing tools, solidifying its reputation as a user-friendly and comprehensive tool.

In the future, the development of 'iSort' presents promising opportunities for more improvements and growth. Potential future work could involve incorporating supplementary sorting approaches to enhance the algorithmic scope of the application. Applying sorting algorithms to real-world datasets can enhance their usefulness and enable users to test the algorithms using legitimate data. Utilizing advanced visualization tools could provide more profound understanding of algorithm behavior, while optimizing the code could enhance performance. Expanding the tool's capabilities to include a broader range of algorithms and transforming it into a more versatile algorithm visualizer holds great potential for further advancement. Integrating aural components into the sorting visualizer would enhance user engagement and interactivity, while offering crucial audible feedback to supplement the visual clues. These upcoming initiatives are positioned to solidify 'iSort' as a prominent platform for exploring sorting algorithms and comprehending algorithmic concepts.

References

[1] M. Marcellino, D. W. William, S. S. Suntiarko, and K. Margi, "Comparative of Advanced Sorting Algorithms (Quick Sort, Heap Sort, Merge Sort, Intro Sort, Radix Sort) Based on Time and Memory Usage," Oct. 2021. 10.1109/ICCSAI53272.2021.9609715.

[2] A. K. Thakkar, S. Dash, and S. Joshi, "Sorting Algorithm visualizer," Jan. 2023.

[3] A. Trivedi, K. Pandey, V. Gupta, and M. K. Jha, "AlgoRhythm - A Sorting and Path-finding visualizer tool to improve existing algorithms teaching methodologies," Feb. 2023. 10.1109/Confluence56041.2023.10048793

[4] B. Goswami,A. Dhar, A. Gupta, and A. Gupta, "Algorithm Visualizer: Its features and working," Jan. 2022.10.1109/UPCON52273.2021.9667586

[5] V. Gupta, "Visualizing, Designing, and Analyzing the Merge Sort Algorithm Retrieved September 10, 2023 from https://medium.com/javarevisited/visualizing-designing-and-analyzing-the-merge-sort-algorithm-904ceb78a592," Mar. 2023.

[6] W. H. Lim, Y. Cai, D. Yao, and Q. Cao, "Visualize and Learn Sorting Algorithms in Data Structure Subject in a Game-based Learning.," Dec. 2022.10.1109/ISMAR-Adjunct57072.2022.00083

[7] MahfuzRifat.(n.d.). Sorting Visualizer. Retrieved from https://mahfuzrifat7.github.io/SortingVisualizer/

[8] A. Jain, "Realizing Algorithms Using GUI," Dec. 2021. 10.1109/SMART52563.2021.9676269

[9] J. Lobo and S. Kuwelkar, "Performance Analysis of Merge Sort Algorithms," Aug. 2020.

[10] A. Prakash, "Sorting visualizers using JavaScript. Retrieved fromhttps://www.youtube.com/watch?v=cW16SGqr_Lg,"

[11] G. Prabhakar, S. Gaur, L. Deshwal, and P. Jain, "Analysis of Algorithm Visualizer to Enhance Academic Learning" Apr. 2022.10.1109/ICIPTM54933.2022.9753906.

[12] C. Mihailescu, "Sorting Visualizer [Computer software]. Retrieved fromhttps://clementmihailescu.github.io/Sorting-Visualizer/,"

[13] Code Drifter. (Year). Sorting Visualizer using JavaScript

[14] D. Khanduja and A. Dhawan, "Comparative analysis of sorting algorithms through visualization tools," International Journal of Computer Applications, vol. 42, no. 14, pp. 23-29, 2012.

[15] J. Hartman, "Sorting Visualizer. [Computer software]," 2021.

[16] Q. Cutts andJ. Bishop, "Sorting out sorting: A classroomactivity for teaching sorting algorithms," Journal of Computing Sciences in Colleges, vol. 30, no. 5, pp. 15-22, 2015.

[17] Y. Chrysanthou and C. Chrysanthou, "Interactive visualization of sorting algorithms using OpenGL," Computer Science Journal of Moldova, vol. 21, no. 3, pp. 352-375, 2013.

[18] E. Halverson and C. Rogers, "A review of research on sorting algorithms: A human-centered perspective. ACM SIGCSE Bulletin," vol. 50, no. 1, pp. 39-155, 2018.

[19] M. Moshtaghi and K. Wilkinson, "Improving student learning of sorting algorithms through algorithm animation and performance analysis," Computer Science Education, vol. 27, no. 3, pp. 255-275, 2017.

[20] S. Palmiter and K. Boese, “Visualizing sorting algorithms using music,” Journal of Computing Sciences in Colleges, vol. 31, no.5, pp. 101-108, 2016.

[21] G. Prabhakar, S. Gaur, L. Deshwal, and P. Jain, “Analysis of Algorithm Visualizer to Enhance Academic Learning,” IEEE, vol. 2, pp. 279-282, 2022.