



Eötvös Loránd University  
Faculty of Informatics  
Department of Algorithms and Their Applications

# Quicksort algorithm illustration

*Supervisor:*

Ásványi Tibor

Associate Professor, Computer Science PhD

*Author:*

Aliia Bazarkulova

Computer Science BSc

Budapest, 2024

**EÖTVÖS LORÁND UNIVERSITY**  
FACULTY OF INFORMATICS

## Thesis Registration Form

**Student's Data:**

**Student's Name:** Bazarkulova Aliia

**Student's Neptun code:** HZ4BV8

**Course Data:**

**Student's Major:** Computer Science BSc

I have an internal supervisor

**Internal Supervisor's Name:** Ásványi Tibor

Supervisor's Home Institution:

Address of Supervisor's Home Institution:

Supervisor's Position and Degree:

**Department of Algorithms and Applications**

**1117, Budapest, Pázmány Péter sétány 1/C.**

*Associate Professor, Computer Science PhD*

**Thesis Title:** Quicksort algorithm illustration

**Topic of the Thesis:**

*(Upon consulting with your supervisor, give a 150-300-word-long synopsis of your planned thesis. )*

The QuickSort algorithm illustration program in Java showcases different variants of the algorithm made for arrays, one-way linked lists, and two-way linked lists.

Upon launch, the program prompts the user to select a specific version and the input elements; however, if the input elements are absent, the program will generate random integers in the range of 0 to 999.

Once the input is set, the program initiates an illustrative step-by-step animation of the QuickSort algorithm. Users have the interactive control over the process, with options to pause, navigate forwards or backwards, enabling a detailed examination of the process.

This program helps users to understand the nuances of QuickSort algorithm such as functionality and efficiency in sorting various data structures.

Budapest, 2023. 11. 27.

# Contents

<b>1. Chapter 1 - Introduction</b>	<b>5</b>
1.1. Motivation.....	5
1.2. Thesis structure .....	6
<b>2. Chapter 2 - Algorithm</b>	<b>7</b>
2.1. Quicksort.....	7
2.1.1. Quicksort for array.....	9
2.1.2. Quicksort for one-way linked lists with a header node.....	10
2.1.3. Quicksort for cyclic two-way lists.....	11
<b>3. Chapter 3 - User Documentation</b>	<b>12</b>
3.1. Flowchart of the program .....	12
3.2. System requirements.....	14
3.3. Installation.....	14
3.4. User guide.....	14
3.4.1. Main page.....	14
3.4.2. Tutorial page.....	19
3.4.3. Test page.....	19
3.4.4. About page.....	21
<b>4. Chapter 4 - Developer Documentation</b>	<b>22</b>
4.1. Specification.....	22
4.1.1. User Stories.....	22
4.1.2. Use case diagram.....	25
4.1.3. Architecture of the program.....	25
4.1.4. Classes and class diagrams.....	27
4.2. Choice of stack.....	28
4.2.1. Back End.....	29

4.2.2.	Front End.....	29
4.2.3.	Other tools.....	30
4.3.	Implementation.....	30
4.3.1.	User Interface.....	30
4.3.2.	Application Logic.....	31
4.3.3.	Control.....	31
4.4.	Testing.....	31
4.4.1.	Testing plan.....	31
4.4.2.	Application logic testing with JUnit.....	32
4.4.3.	User interface testing with React Testing Library.....	33
4.4.4.	Testing results.....	34
<b>5.</b>	<b>Conclusion</b>	<b>35</b>
5.1.	Conclusion and Future work.....	35
	<b>References</b>	<b>37</b>
	<b>List of Figures</b>	<b>38</b>

# Chapter 1

## Introduction

Sorting algorithms play a significant role in computer science, serving as a gateway to understanding the core algorithm principles. They introduce us to the concept of big-O notation, mathematical notation that describes the limiting behavior of a function when the argument tends towards a particular value or infinity. This is crucial in understanding the time and space complexity of algorithms, which is important in determining their efficiency.

Among these algorithms, we will be discussing divide-and-conquer algorithms. Divide-and-conquer is an algorithm design paradigm that recursively breaks down the problem into two or more sub-problems of the same type until they become simple enough to be solved directly. This approach is not only efficient, but also elegant, demonstrating the power of recursion.

One of the most famous divide-and-conquer algorithms is Quicksort. It is efficient, with its best and average case time complexity,  $\Theta(n \log n)$ , where  $n$  is the number of elements being sorted. Even though its worst case time complexity is  $\Theta(n^2)$ , it is often faster than other sorting algorithms such as Insertion Sort and Mergesort.

Quicksort also introduces us to the concept of in-place sorting in a broad sense. Even though the sorting happens within the dataset, broad sense means that quicksort needs  $\Theta(\log n)$  storage for controlling the recursion. Comparatively to other sorting algorithms, like Mergesort, it is a good choice if the memory has a limitation factor.

### 1.1 Motivation

The primary motivation behind this illustration program is to enhance the learning experience and to understand the practical usage of the Quicksort algorithm. It aims to allow learners to interact with the algorithm, giving the inputs and observing the generated output. The output is a detailed explanation of each step in the process. This approach makes it possible to take a glimpse at an actual visualization of the algorithms, and becomes a visual aid in learning alongside passive learning methods such as reading or listening to the lecture.

Furthermore, the illustration program can help users to identify their errors and misconceptions. By providing immediate results, it allows users to verify their own solutions on running the Quicksort algorithm for specific inputs since the implementation was carefully designed according to the specifications taught in the Algorithms and Data Structures course at ELTE. Hence, it can be a valuable tool for learners to detect mistakes or misinterpretations about the Quicksort algorithm.

## 1.2 Thesis structure

This paper is divided into five main chapters, followed by a bibliography and a list of figures. Chapter 1 briefly introduces the sorting algorithms, with an emphasis on Quicksort. It outlines the main motivation behind the program and the goals it aims to achieve.

In chapter 2 we will go deeper into Quicksort. It explores the mechanics, time and space complexities with different inputs, and its differences in usage across various data structures, such as arrays, one-way lists and cyclic two-way lists. The main goal of this chapter is to provide the theoretical understanding of the versatility of the algorithm.

Chapter 3 is dedicated to the user experience of the program. It delivers comprehensive details including the system requirements, installation and program preparation. Additionally, a user's guide will be available, ensuring that users are aware of all features and other capabilities.

The 4th chapter focuses on the developer documentation. It includes the tech stack, user stories, architecture of the program, class diagrams, testing plan and the results of testing, and other relevant information from the developer's perspective. This section offers insights into the solutions and challenges encountered during the development phase of the program.

Lastly, in chapter 5 we can see the conclusion. It reflects the impact and the limitations of the current state of the program. This chapter also outlines the potential improvements, suggesting how the Quicksort illustration program can be enhanced and extended.

# Chapter 2

## Algorithm

In this chapter we will learn and delve deeper into understanding the essence and steps of the Quicksort algorithm for the following three data structures: arrays, one-way lists with a header node and cyclic two-way lists.

### 2.1 Quicksort

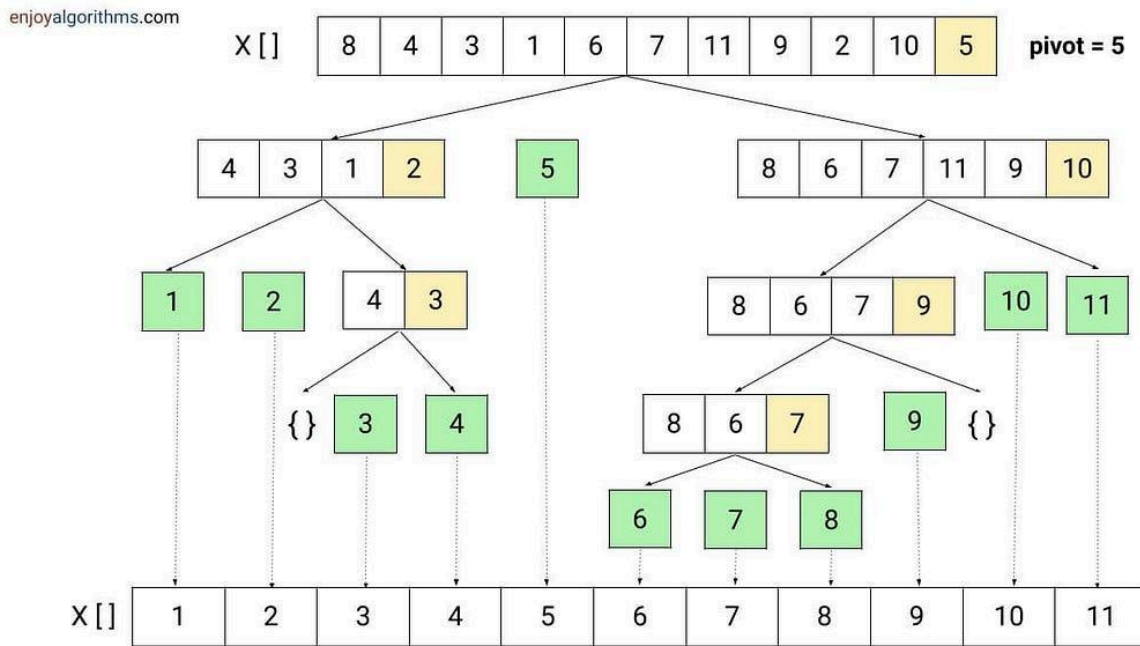
As discussed earlier, Quicksort is a divide-and-conquer algorithm, it is based on the partitioning routine. It should be used on a range of at least two elements. Partitioning creates two divisions of sub-ranges, in a way that all elements of the first sub-range are not greater than any element of the second sub-range.

After executing the first partition, it will recursively apply the same logic to each sub-range and at the end of the partitions, there is one element that finds its position in the whole range. That element is called the pivot. The pivot is an arbitrary element in the dataset, around which the other elements will be partitioned. This process can be done in-place, requiring small amounts of memory to perform the sorting.

Algorithm steps:

- Choose a pivot: Select an element from the dataset to act as a pivot. The choice of pivot can be different - it can be the first, last, middle or random element. Note, that the performance of the algorithm relies greatly on the choice of a pivot.
- Partitioning: The dataset will be rearranged in a way that the values less than the pivot will be put into the left sub-range, while the elements greater to the pivot will be in the right sub-range. The elements equal to the pivot go into any of the left or right sub-range. At the end the pivot will find its final position.
- Recursion: The two steps above will be recursively applied to the left sub-range with smaller or equal values, and then to the right sub-range with greater or equal values separately.
- Combine: The results are combined to get the final sorted result.

Figure 2.1 Partitioning, recursion, combine (google images)



The efficiency of the Quicksort algorithm depends on the good choice of the pivot. A bad choice of the pivot will decrease the performance significantly. For example: if the dataset is already in a particular order, such as reversed order, selecting the first or last element as pivot would lead to poor performance. This is because the pivot fails to divide the range equally, leading to the quadratic time complexity -  $\Theta(n^2)$ . While the optimal choice of the pivot selection would be the median, as it perfectly divides the range into two equal halves, finding the true median can be time-consuming for practical purposes. A more feasible strategy would be choosing the pivot randomly. Through randomization, the algorithm avoids the worst-case scenarios of a fixed pivot choice and improves its average time complexity -  $\Theta(n \log n)$ .

Figure 2.2 Quicksort time complexities

Best case	Average case	Worst case
$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n^2)$



### 2.1.1 Quicksort for arrays

The array is a fundamental data structure used to store a collection of elements of the same type in contiguous memory blocks. Each element of the array is accessible through its index.

Quicksort for arrays is the most popular and widely used variant of the algorithm, however it is important to note that quicksort for arrays is not stable. Unstable sorting means that equal elements are not guaranteed to retain their original relative order. The main reason for this is the swap function, which interchanges the elements that have to be swapped during the process.

Figure 2.3 Quicksort for array pseudocode

```
function quicksort(arr):
    QS(arr, 0, arr.length - 1)

function QS(arr, p, r):
    if p < r:
        q = partition(arr, p, r)
        QS(arr, p, q - 1)
        QS(arr, q + 1, r)

function partition(arr, p, r):
    i = random(p, r)
    swap(arr, i, r)
    i = p
    while i is less than r and arr[i] is less than or equal to arr[r]:
        i++
    if i is less than r:
        j = i + 1
        while j is less than r:
            if arr[j] is less than arr[r]:
                swap(arr, i, j)
                i++
            j++
        swap(arr, i, r)
    return i

function swap(arr, i, j)
    temp = arr[i]
    arr[i] = arr[j]
    arr[j] = temp
```

There is an optimization possibility to be considered. For small arrays, insertion sort usually behaves better than quicksort, due to its lower overhead. Insertion sort is another sorting algorithm that builds the final result one element at a time by comparisons. So in the implementation of the quicksort, when the sub-array is smaller than predefined threshold, the algorithm can switch to insertion sort, which will speed-up the process significantly.

### 2.1.2 Quicksort for one-way lists with a header node

One-way lists or singly linked lists is a linear data structure consisting of keys (elements) where each key has a pointer to the next key in the list. The last element has the reference NULL which represents the end of the list. In this work, we are working with one-way lists with a header node. This header node, situated at the zero-th place, serves as a starting point that points to the first actual element in the list. Compared to arrays, linked lists are better in terms of efficient insertion and deletion operations due to its dynamic memory allocation.

Quicksort for singly linked lists follows the base structure of the algorithm, but with one key difference. As mentioned earlier, quicksort for arrays is not stable, but it is a different situation for linked lists. Quicksort can be made stable by replacing the swap with two alternative common linked list operations: unlink and precede. By using these functions, now we have a stable sorting algorithm. For simplicity and effectiveness, the pivot should be the first element of the list. This choice simplifies the logic in the following way: if the next element is greater than the pivot the algorithm moves to the next element; if the next element is less than the pivot, it is unlinked from its current position without disrupting the rest of the elements, and then reinserted right before the pivot element.

Figure 2.4 Quicksort for linked list with a header node pseudocode

```
function quicksort(H):
    QS(H, null)

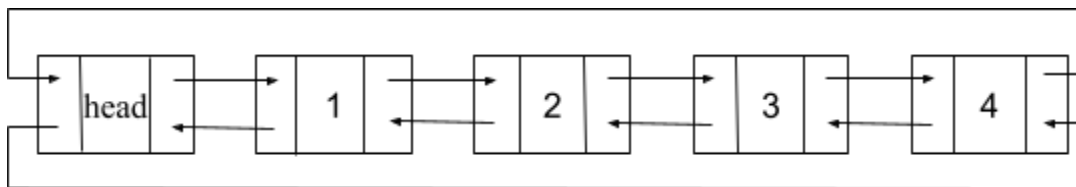
function QS(p, r):
    t = p.next // t refers to the pivot
    if t is not equal to r and t.next is not equal to r:
        partition(p, t, r)
        QS(p, t)
        QS(t, r)

function partition(p, t, r):
    s = t.next
    ps = t
    while s is not equal to r:
        if s.val is greater than or equal to t.val:
            ps = s
            s = s.next
        else
            q = s
            s = s.next
            ps.next = s
            q.next = p.next
            p.next = q
            p = q
```

### 2.1.3 Quicksort for cyclic two-way lists

Two-way list or a doubly linked list is a more complex type of linked list. Each element in a doubly linked list contains three main components: a key, a pointer to the next element, and a pointer to the previous element in the sequence. In addition to these features, cyclic two-way lists have one unique characteristic: they are circular. This means that the list does not terminate with null. Instead, the last element's next pointer directs back to the header, and conversely, the first element's previous pointer refers back to the header. This makes it possible to traverse the list without keeping the track of the last element of the list.

Figure 2.5 Cyclic two-way list



Quicksort for cyclic two-way lists is very similar to the implementation of singly linked lists. In both cases, the algorithm uses unlink and precede operations to sort the list. They ensure the original order of the list maintaining stability.

Figure 2.6 Quicksort for cyclic two-way list pseudocode

```
function quicksort(H):
    if H.next is not equal to H:
        QS (H, H)

function QS(p, r):
    if p.next is not equal to r and p.next is not equal to r.prev:
        t = partition (p, r)
        QS(p, t)
        QS(t, r)

function partition (p, r):
    t = p.next // t refers to the pivot
    s = t.next
    while s is not equal to r:
        if s.val is greater than or equal to t.val:
            s = s.next
        else:
            q = s
            s = s.next
            unlink(q)
            precede (q, t)
    return t
```

# Chapter 3

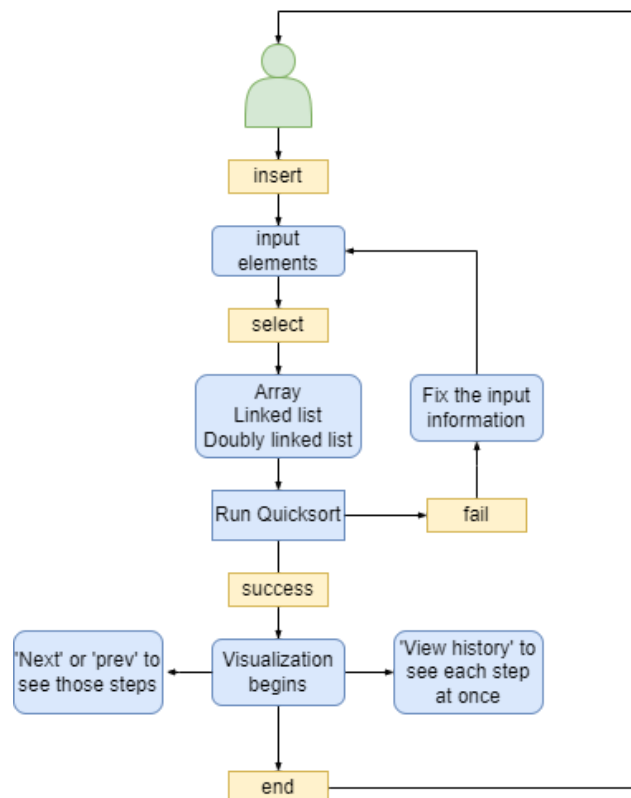
## User Documentation

This chapter provides clear guidance on the necessary prerequisites, installation steps and system requirements. By following this guide, users will be able to fully understand and utilize all the features available.

### 3.1 Flowchart of the program

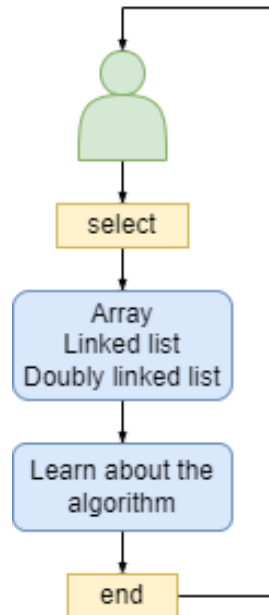
Upon visiting the main page, users will see input boxes where they can specify the number of elements or insert the input elements directly. Users may also select the data structure (array, linked list, doubly linked list) and initiate the sorting process by pressing the “Run Quicksort” button. The sorting is animated, with “Next” and “Previous” buttons available, allowing users to navigate through each step of the algorithm. Additionally, “View history” button lets users review each step of the process at once.

Figure 3.1 Main page basic flowchart



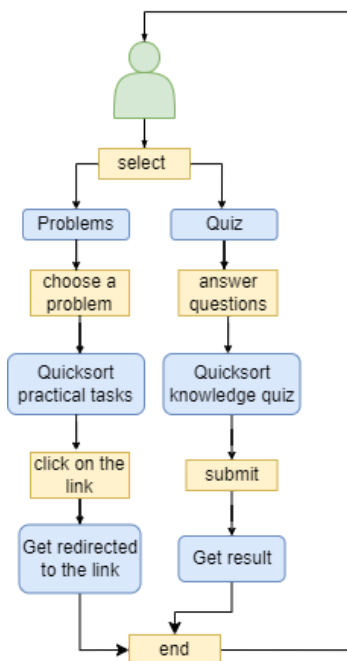
The program also includes a “Tutorial” page that offers a theoretical background on Quicksort, enhancing the users’ understanding.

Figure 3.2 Tutorial page flowchart



For users who want to test their new acquired knowledge, “Test” page is available.

Figure 3.3 Test page flowchart



Lastly, “About” page gives insights about the page’s purpose and origin.

## 3.2 System requirements

This program is compatible with most commonly used operating systems including Windows, MacOS, and Linux. You can interact with the page using any modern web browser, such as: Chrome, Microsoft Edge, or Safari.

## 3.3 Installation

To ensure the successful use of a program, follow the steps below:

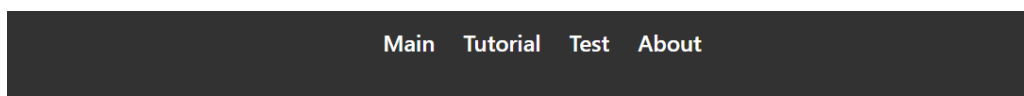
1. Download the latest JDK version from [Java Downloads](#). Creating a Java environment is crucial for program usage. Check it by running “java --version” on any command prompt.
2. If you have Windows operating systems download the zip file in the following link - [Quicksort-Windows](#), if you have Linux or MacOS - [Quicksort-MacOS-Linux](#)
3. Unzip the file, then double click on the ‘run’ file and the program will automatically start on your browser on ‘http://localhost:8080’. If the program is not running instantly, wait for 5-10 seconds.

## 3.4. User guide

### 3.4.1 Main page

Upon launching the program, first we see the Main Page.

Figure 3.4 Main page



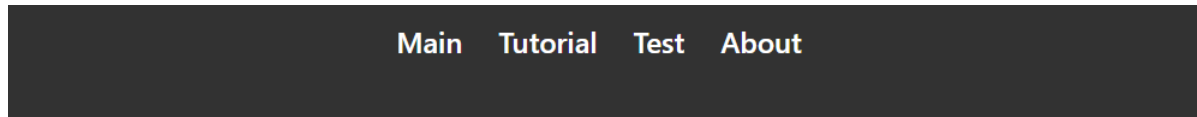
## Quicksort illustration

Enter comma-separated numbers for Quicksort:

<input type="text" value="Length (Optional)"/>	<input type="text" value="Data (Optional)"/>	<input type="text" value="Select Data Structure"/>	<input type="button" value="Run Quicksort"/>
--	--	--	--

If the input data is missing then an error is indicated on the page.

Figure 3.5 Incorrect inputs



## Quicksort illustration

Enter comma-separated numbers for Quicksort:

1	tyt	Array	▼	Run Quicksort
---	-----	-------	---	---------------

Please enter valid numbers separated by commas.

We can either specify the number of elements or input the numbers themselves and choose the data structure from the drop-down menu: array, linked list, doubly linked list.

Figure 3.6 Correct inputs on the Main Page

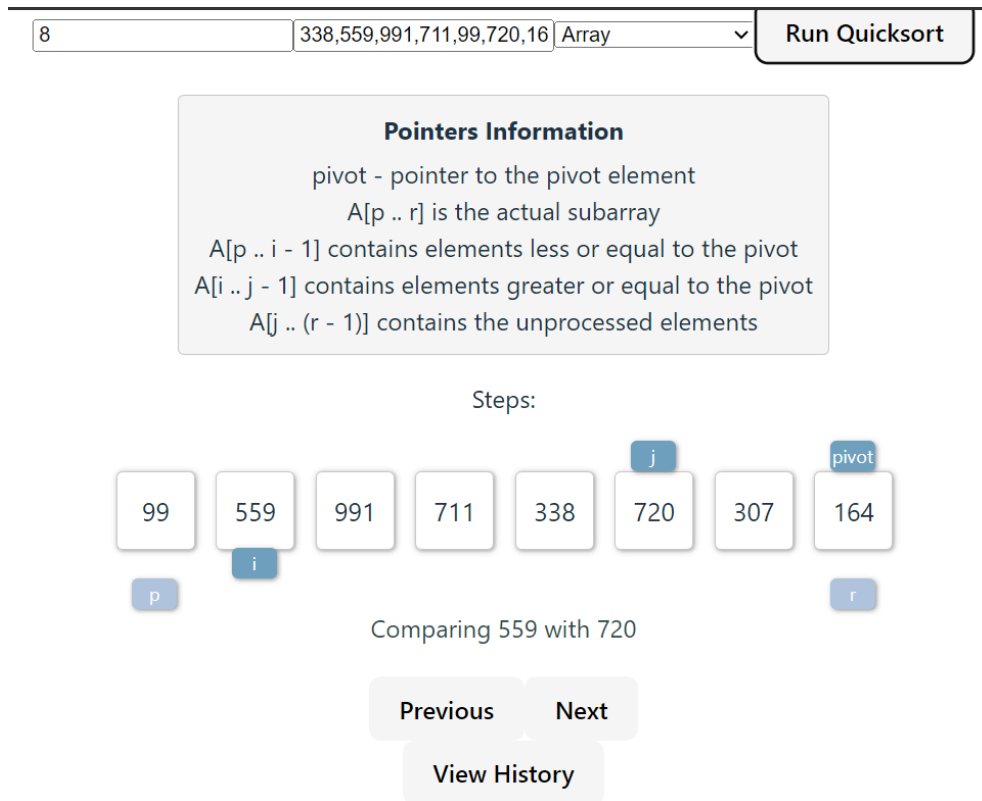
## Quicksort illustration

Enter comma-separated numbers for Quicksort:

8	Data (Optional)	Select Algorithm ▼	Run Quicksort
		Select Algorithm	
		Array	
		Linked List	
		Doubly Linked List	

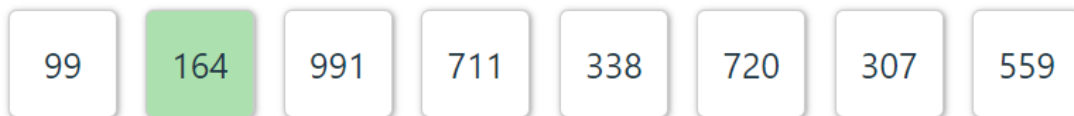
If the chosen data structure is an array, we see the following. The pivot, the pointers i, j, p, and p, as well as the buttons 'next', 'previous', and 'view history'.

Figure 3.7 Array sorting demonstration



After the end of each partitioning, an element will find its final place and we can see it indicated visually.

Figure 3.8 End of partition



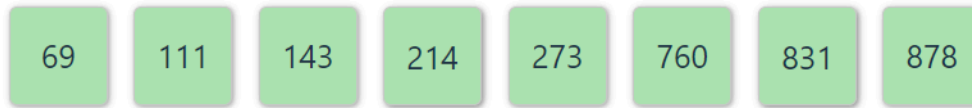
164 found its final position

Thus, at the end, the whole array will be painted green to indicate the end of the sorting process.



Figure 3.9 End of quicksort

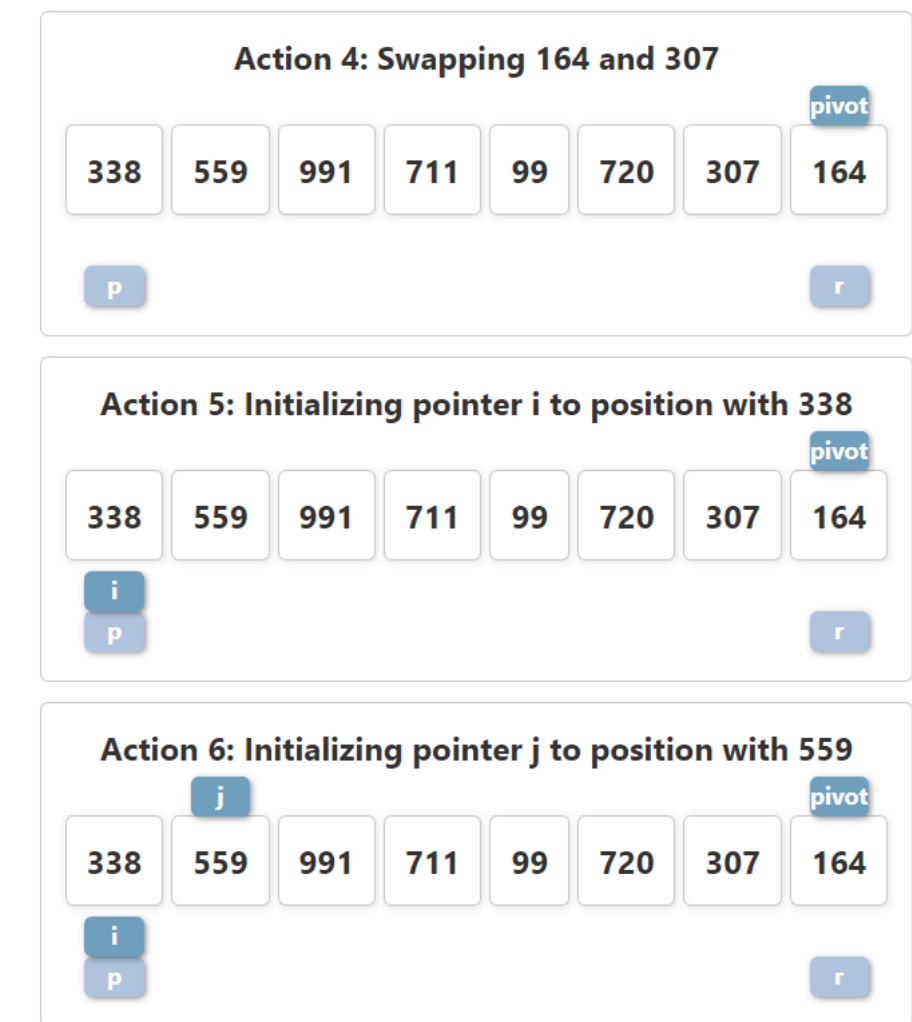
Intermediate Steps:



The list is sorted

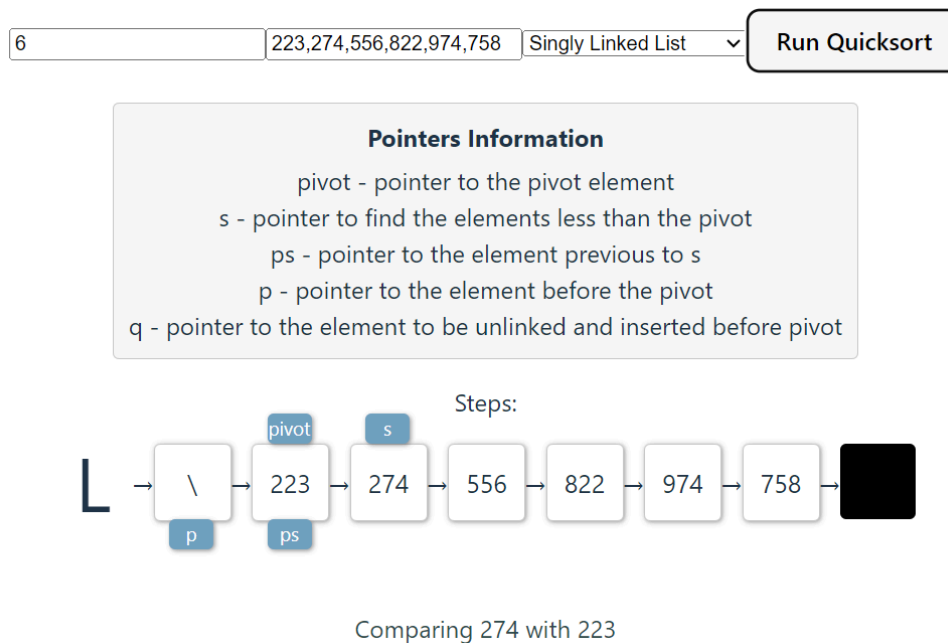
Additionally, the ‘View history’ button can be clicked anytime to show each step made from the beginning.

Figure 3.10 ‘View history’ button



If the chosen data structure is linked list, then the following information will be shown:

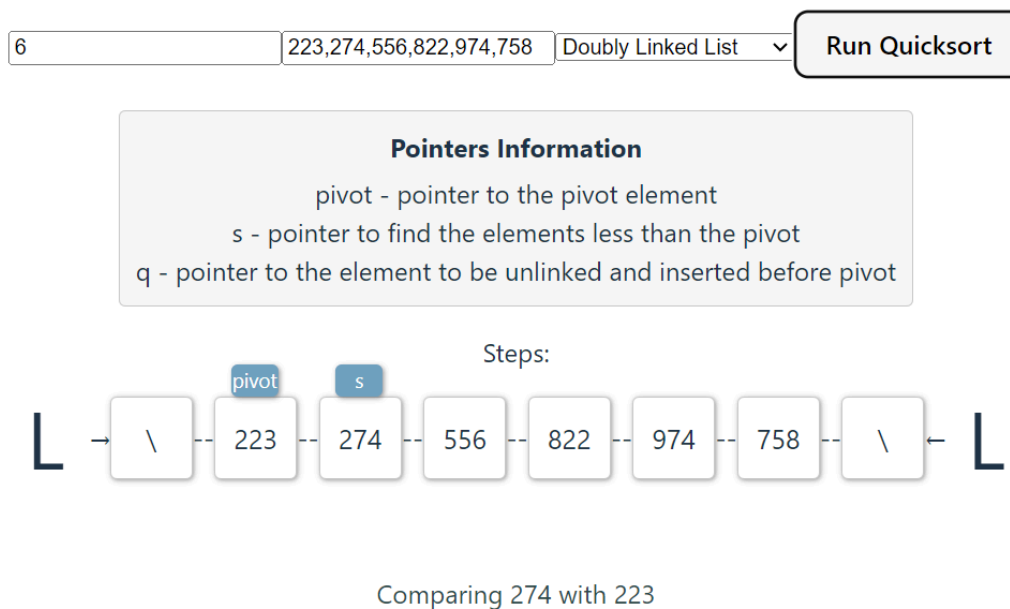
Figure 3.11 Singly linked list sorting demonstration



We can see the pivot element, p, ps, and s pointers. All of the buttons are available and one of the elements has already found its final place.

Lastly, the same is shown for doubly linked list:

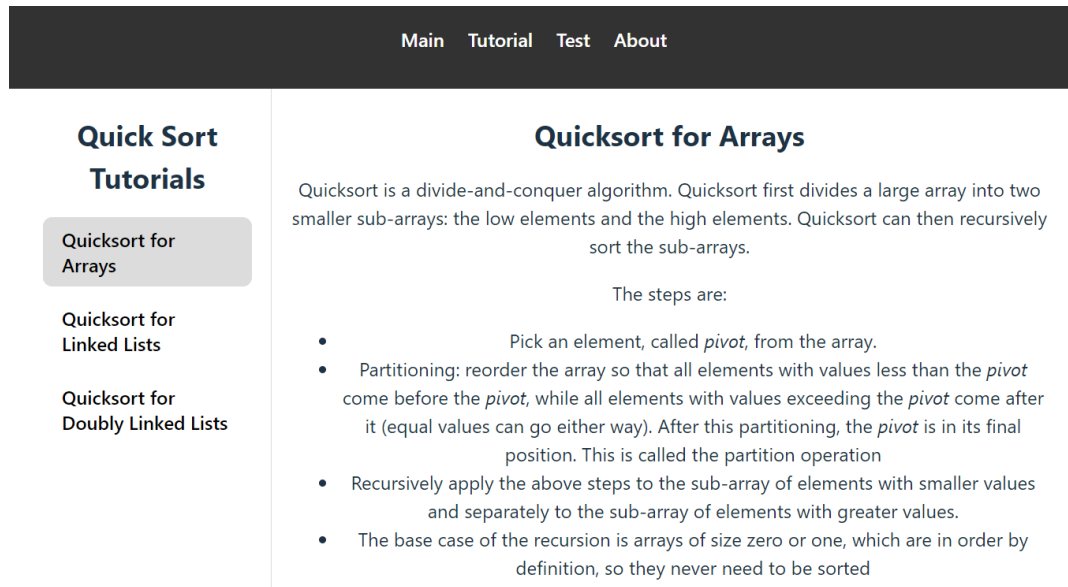
Figure 3.12 Doubly linked list sorting demonstration



### 3.4.2 Tutorial page

If we want to learn the theoretical background of the quicksort algorithm, this is possible to do in the “Tutorial” page. In this image, we see the content for Quicksort for Arrays. Also, as seen in the navbar on the left, we can switch between the data structures to learn about other ones as well.

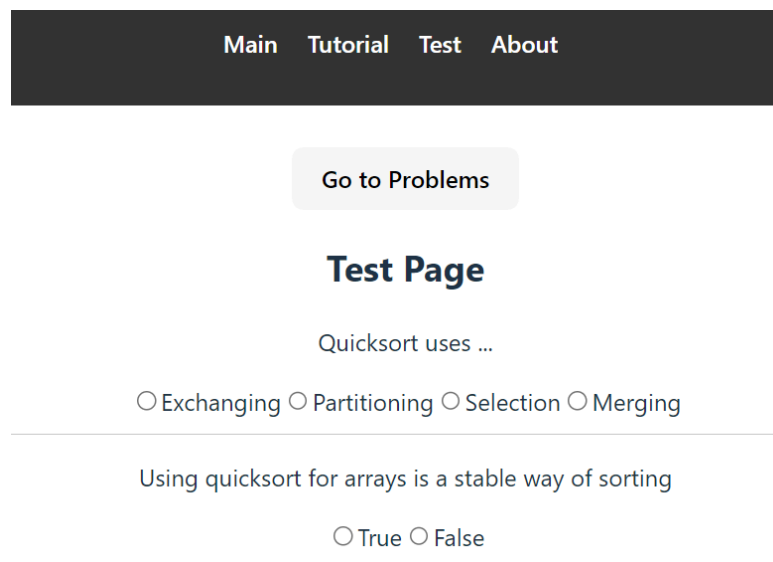
Figure 3.13 Tutorial page layout



### 3.4.3 Test page

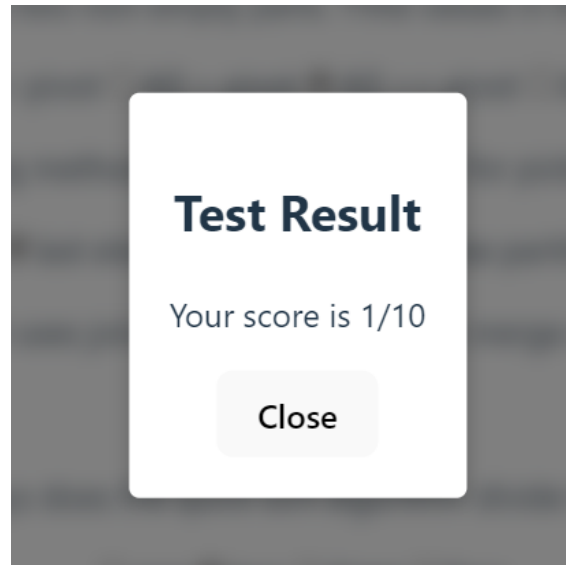
On the test page, you are able to see the list of questions related to quicksort.

Figure 3.14 Test page layout



Upon submission of your answers, you can see the result immediately.

Figure 3.15 Score window



If you click on ‘Go to Problems’, you can see the list of practical tasks related to Quicksort. By clicking on the link, the user will be redirected to the task’s website, where the user can solve and submit it.

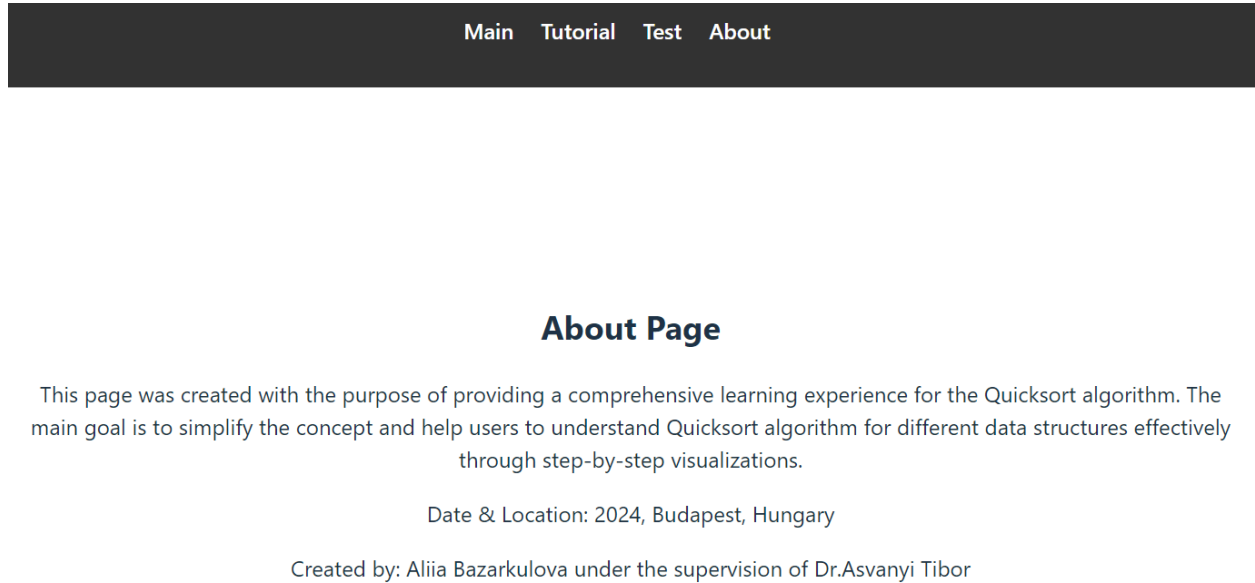
Figure 3.16 Problems Page

<div>Go to Quiz</div>			
Problems Page			
Name	Difficulty	Success Rate	Link
Quicksort - Partition	Easy	96%	Solve
Quicksort - Sorting	Easy	91%	Solve
Cricket tournament	Easy	66%	Solve
Eating apples	Easy	85%	Solve

### 3.4.4 About page

Lastly, the About page gives the details and the origins of the program's creation

Figure 3.17 About page layout



# Chapter 4

## Developer Documentation

This chapter explores the program from the perspective of a developer, discussing key aspects such as user stories, architecture and class diagrams. It will also cover the selection of technology stack, details of the implementation process, and the testing of the program.

### 4.1 Specification

The specification provides a comprehensive overview of the whole software development cycle for the program. User stories and use case diagrams illustrate the user's perspective, highlighting how they interact with the program. The architecture demonstrates the structure and layers that were created to showcase the connectivity of the front and back ends. Classes and class diagrams showcase the attributes of each class and their relationships.

#### 4.1.1 User Stories

User story is an informal, general explanation of features available for the user, written from their perspective.

**As a User, I want** to see how the quicksort algorithm works on different data structures.

1. **Given** I am in the main page  
**When** the main page is loaded  
**Then** I can choose which data structure I would like to use
2. **Given** I am in the main page  
**When** the main page is loaded  
**Then** I can input specific elements or opt for a random selection  
**When** I opt for a random selection for the elements  
**Then** I will specify the number of elements
3. **Given** I have given the elements and chosen the data structure  
**When** the input is incorrect or the mandatory fields are not filled

- Then** the mistakes will be shown to demonstrate what was incorrect
4. **Given** I have finished the pre-selected wishes for the illustration  
**When** I press the button 'Run Quicksort'  
**Then** the illustration starts
  5. **Given** I pressed the 'Run Quicksort' button  
**When** the illustration is running  
**Then** I can click on 'next' or 'prev' buttons to see each step of the animation
  6. **Given** the illustration is running  
**When** I clicked on the 'next' button at least once  
**Then** I can click on the 'View history' to see the full history of steps and actions

**As a User, I want** to learn the basic knowledge on quicksort

1. **Given** I am in the "Tutorial" page  
**When** When the page is loaded  
**Then** I can choose the quicksort tutorial for different data structures
2. **Given** The "Tutorial" page is loaded  
**When** I choose array as the data structure  
**Then** I can see the theoretical part of quicksort for arrays
3. **Given** The "Tutorial" page is loaded  
**When** I choose singly linked list as the data structure  
**Then** I can see the theoretical part of quicksort for singly linked list with a header node
4. **Given** The "Tutorial" page is loaded  
**When** I choose two-way lists as the data structure  
**Then** I can see the theoretical part of quicksort for cyclic two-way list

**As a User, I want** to test my knowledge about Quicksort

1. **Given** I am in the "Test" page  
**When** When the page is loaded  
**Then** I can see a quiz for quicksort knowledge
2. **Given** The page is loaded  
**When** I see the quiz for quicksort knowledge

- Then** I can choose the answers
3. **Given** Quiz for quicksort knowledge  
**When** I finished choosing the answers for all questions  
**Then** I can see submit and see the score immediately
4. **Given** I am in the “Test” page  
**When** I see “Go to Problems” button  
**Then** I can click in the “Go to Problems” button and be redirected to the ‘Problems’
5. **Given** I have clicked on “Go to Problems”  
**When** I see the list of problem  
**Then** I can choose one problem and click on its link
6. **Given** I have clicked on the link of a problem  
**When** I will be redirected to a website that has that problem  
**Then** I can solve and submit it there
7. **Given** I am in ‘Problems’ page  
**When** I see the “Go to Quiz” button  
**Then** I can click on “Go to Quiz” and be redirected to the test again

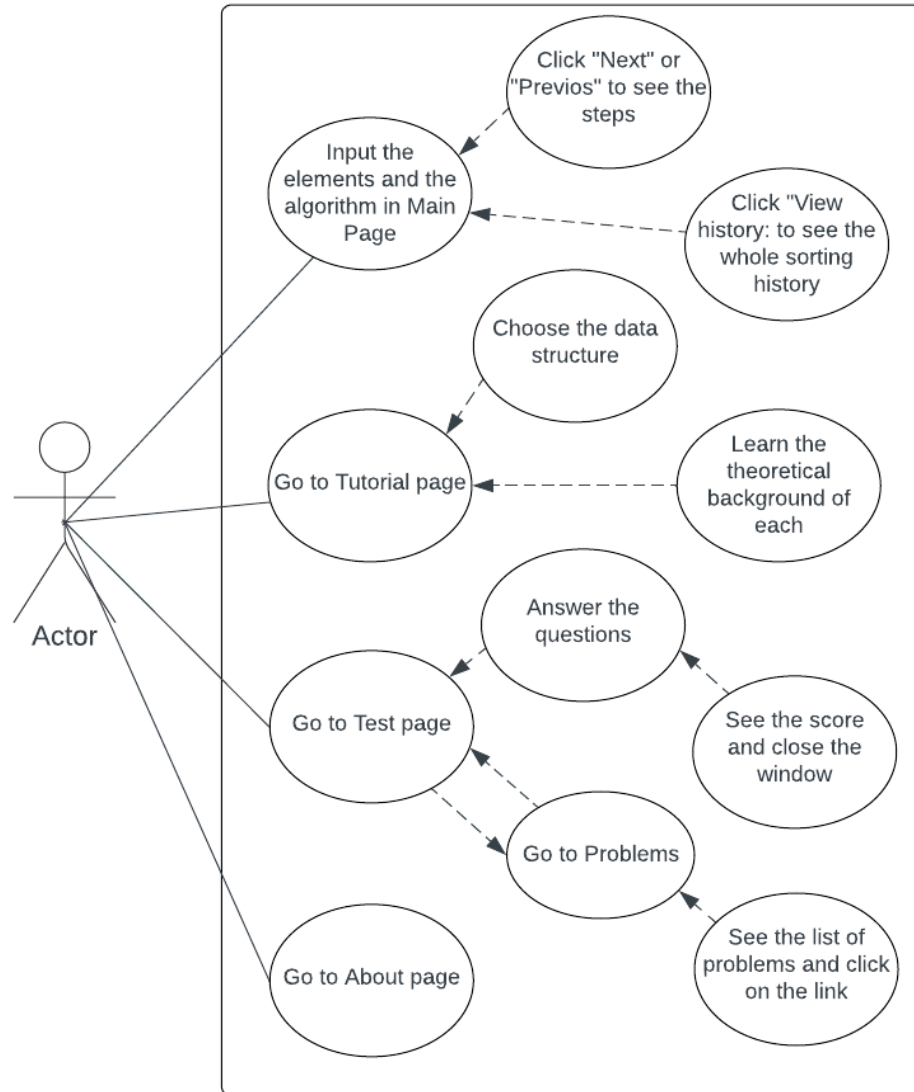
**As a User, I want to know about the creators, purpose and the origin of this page**

1. **Given** I am in the “About” page  
**When** When the page is loaded  
**Then** I can see the information about the creator of this page  
**And** I can also see when, where and why this page was created



#### 4.1.2 Use case diagram

Figure 4.1 Use case diagram



#### 4.1.3 Architecture of the program

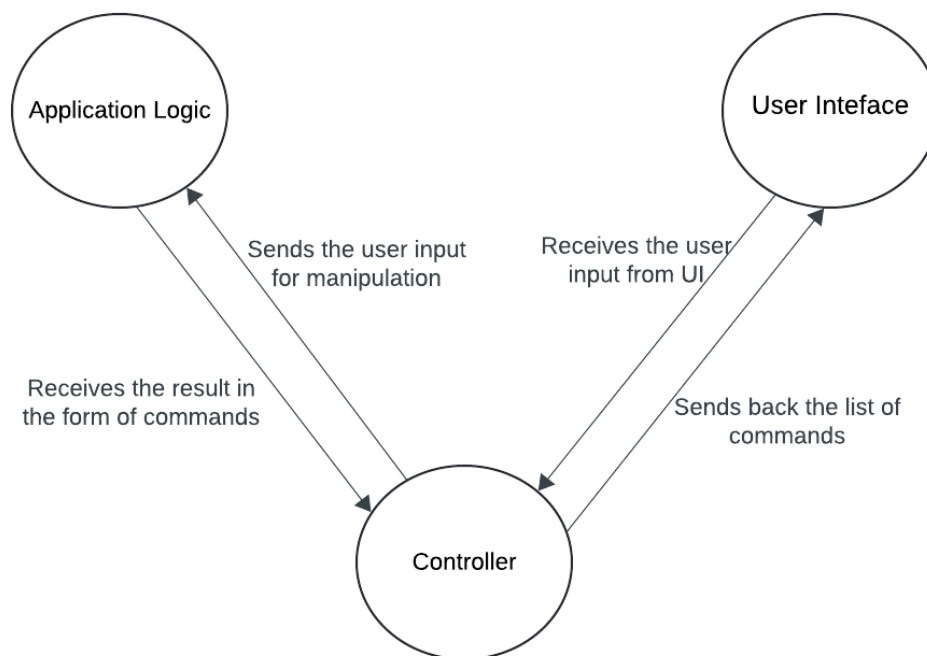
The software architecture is divided into three main parts:

- User Interface: This layer contains the main page, tutorial, test and about pages of the program. The layer interacts with the user and provides the control interfaces. It contains components such as: the input form, buttons for choosing the options on how we want to see the program. Moreover, it is responsible for rendering the visual representation of the

sorting process. It displays the unsorted initial data, partitions, swaps and sorted results. Additionally, the users can control the whole process, using the “Next” and “Previous” buttons, users have the chance to see all steps made with the “View history” button.

- Application logic: This layer contains the main logic of the Quicksort algorithm. There are three different classes for each data structure that will manage the smooth process.
- Controller: It is responsible for the connection of the other two layers. From the User Interface it receives the elements and the data structure. With the received data it will perform the sorting logic, then after applying the sorting method of a corresponding class, sends back each step of the process to achieve the result.

Figure 4.2 Architecture of the program



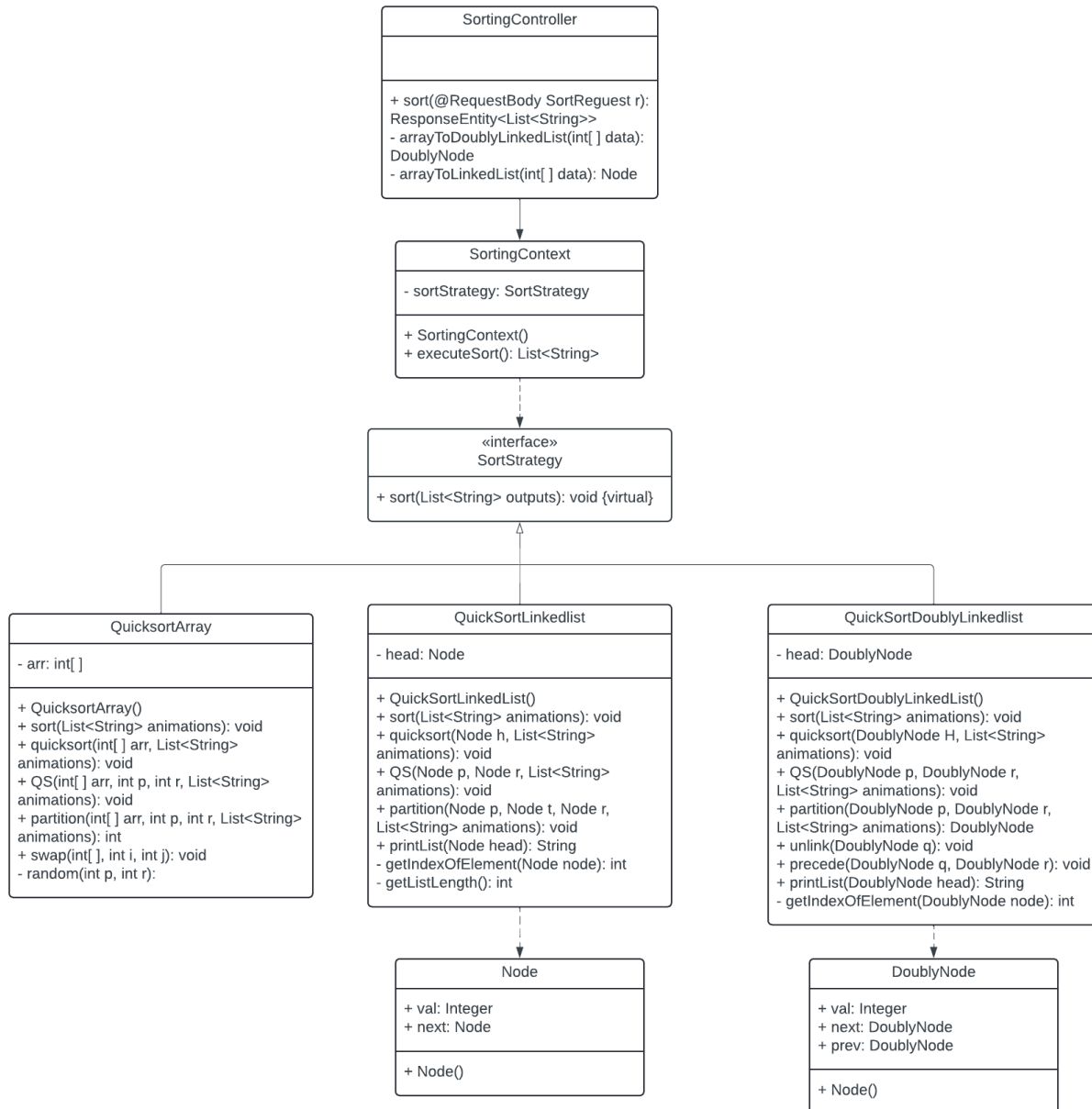
#### 4.1.4 Classes and class diagrams

Introduction of each class:

- Node.java: this class created the node type needed for the implementation of a linked list. It has two main attributes, the key (value) of the node and the pointer to the next node.
- DoublyNode.java : this class is responsible for creating the doubly linked list. It has the key, pointers to the next and previous nodes.
- QuickSortArray.java : the implementation of the sorting process for arrays.
- QuickSortLinkedList.java : the implementation of the sorting process for one-way lists with a header node.
- QuickSortDoublyLinkedList.java : the implementation of the sorting process for cyclic two-way lists.
- SortStrategy.java : interface with one method sort, which must be implemented by each class that adopts this strategy.
- SortingContext.java : the central class for the algorithm that receives the inputs and using SortStrategy.java calls the corresponding sorting method.
- SortingController.java : controller that manages interactions with the front end. It utilizes the SortingContext.java to process data received and then sends the results back as a response.

As seen in the picture (see Figure 4.3) and the description of each class, the application's logic uses a Strategy design pattern to allow a flexible and smooth process. "The Strategy design pattern suggests encapsulating different implementations of a specific task into distinct classes known *strategies*. The original class, called *context*, must have a field for storing a reference to one of the strategies. The context delegates the work to a linked strategy object instead of executing it on its own."

Figure 4.3 Backend logic class diagram



## 4.2 Choice of stack

The quicksort illustration was implemented using JAVA for the backend and ReactJS for the frontend. Java was chosen for its platform independence as it can run on any platform that

supports Java Virtual Machine (JVM), while ReactJS was selected for its component-based architecture, which encourages a clean and organized codebase.

#### 4.2.1 Back End

The core functionality of the program utilizes Java to implement the sorting methods, with Spring Boot managing the connection and sending the results of the Quicksort algorithm to the frontend.

##### Java

“Java is a widely used object-oriented programming language. The rules and syntax of Java are based on C and C++ languages. Java was first released back in 1995 and it is popular for developing applications for web, desktop and mobile devices. The principles for creating Java were simplicity, robustness, multithreaded, high-performance, portability, etc.”

##### Spring Boot

“Spring Boot is an open-source Java framework used to create micro services. It simplifies the configuration and setup processes, allowing developers to focus on writing code. Spring Boot is widely utilized to create and run both simple and web-based applications.”

#### 4.2.2 Front End

The visual part of the program was created using React JS, its associated libraries and CSS.

##### React JS

“React JS is a free and open-source front-end JavaScript library for building user interfaces based on components. React components are the building block of React Application. These components use reusable code blocks that encapsulates both logic and UI elements. They have the same purpose as JavaScript functions and return HTML. This approach simplifies the development of complex interfaces. While React primarily handles the UI and rendering the elements to the DOM, it often works with other libraries for routing and other client-side

functionality. For instance, Axios is used for the connection with the back end in this program. Axios works by making HTTP requests with NodeJS and lets developers make requests to their own or third-party servers to fetch data. If the request was successful, you receive a response with the requested data, and if not then you receive an error. Another library that was used in the development is React-Spring. It is an animation library that uses physical properties which results in animations that feel more natural and smooth. Its configuration accepts properties such as tension, mass, friction and velocity. React-Spring improves the user experience and helps to demonstrate a smooth animation of the quicksort process in this program.”

#### 4.2.3 Other tools

Other tools used in the development: NodeJS, Git, HTML, CSS

### 4.3 Implementation

In this section we will read about the detailed implementation process diving into the User Interface, Visualization and Control.

#### 4.3.1 User Interface

For User Interface React and its libraries are utilized. To facilitate the reuse of common functions across different components, custom hooks such as useStepNavigation.js were created. This hook is used to see next or previous steps in any data structure’s quicksort process. Additionally, to enhance the code modularity, createAnimation.js was created as a Higher Order Component (HOC), that can be used in each data structure’s animation, ensuring consistency and reusability.

Other tools and technologies that were used:

- CSS: Basic CSS was used for styling of the pages, pointers and other div elements, providing a clean and user-friendly interface.
- React-Spring: This library is used to render state changes to the DOM and make animations smooth.
- Axios: Axios is used as a solution for connectivity with the back end, enabling the application to fetch necessary data for animations.

### 4.3.2 Application Logic

The application logic utilizes Java for the codebase. As seen in the class diagram (see Figure 4.3), the Strategy Design Pattern keeps the code structure clean. To simplify interactions with the frontend, a list of actions is sent to instruct the sorting process. Each Quicksort class is implemented with its logic and maintains an "animations" list of strings, which records each step of the sorting process. For instance, "compare,56,1,67,2" indicates comparing the numbers 56 and 67 with indices 1 and 2, while "init,s,67,2" initializes the pointer 's' to 67 at index 2. Upon completion, the resulting list of strings contains all recorded actions.

### 4.3.3 Controller

Implementing the controller was quite a challenge due to the use of two different programming languages for the frontend and backend. Bridging this gap required establishing a connection that could effectively control and utilize both components. For the backend, the *sort()* method was created using Java Spring Framework's specialized annotations, *@PostMapping* and *@RestController*. This method handles requests expecting input data including the algorithm and elements, and sends back the resulting sorted data to the endpoint. On the frontend, *Axios*, a React library, was employed. The main page component collects user input and sends it to the backend for processing by wrapping it into a single message.

## 4.4 Testing

Testing is a vital part of software development, essential in identifying and fixing errors introduced during development. It ensures that all functionalities work correctly, thus preventing future failures and maintaining user satisfaction.

### 4.4.1 Testing plan

The program's testing plan involves two main components: application logic and UI. The application logic uses JUnit to test each class' sorting method and their helper functions. For UI testing, React Testing Library is used. This will involve verifying the proper rendering and responsiveness of components, as well as testing user interaction functionalities. By following

this testing plan, the reliability and accuracy of both the backend algorithms and the frontend user interface is achieved.

#### 4.4.2 Application logic testing with JUnit

“JUnit is a unit testing framework for the Java programming language. JUnit has been important in the development of test-driven development, and is one of a family of unit testing frameworks collectively known as xUnit, that originated with JUnit.”

##### Quicksort for array testing

Test the ‘swap’ function

- Create an array, swap any of the elements and check if the output is correct.

Test the ‘partition’ function

- Create an array of any size, call the partition function, and get the result of an element that found its final place. Iterate through the array and check if the elements before that partition result number are less than it, while the ones after are greater or equal.

Test the ‘sort’ function

- Create one element or zero element arrays. The result should be the same as the input.
- Create arrays that are already sorted, the result should be the same.
- Create arrays in descending order, the result should be sorted in ascending order.
- Create arrays of different sizes, the outputs should all be sorted in the ascending order.

##### Quicksort for one-way list with a header testing

Test ‘partition’ function

- Create a list of arbitrary size, invoke the ‘partition’ function, and check if the node was correctly placed.
- Repeat the ‘partition’ process to verify the subsequent partitions also get correct results.

Test ‘sort’ function

- Create one element or zero element lists. The result should be the same as the input.
- Create lists that are already sorted, the result should be the same.
- Create lists in descending order, the result should be sorted in ascending order.
- Create lists of different sizes, the outputs should all be sorted in the ascending order.



## Quicksort for cyclic two-way lists testing

### Test 'unlink' and 'precede' functions

- Create a two-way list and pick any element to remove. Invoke 'unlink' function and check if the result does not have the element that was picked.
- Invoke the 'precede' function with the removed node to be inserted before the first element of the list. Check if the new node was inserted correctly.
- Repeat the 'precede' operation with a new element to be positioned before the second element of the list and verify the accuracy of the result.

### Test 'partition' function

- Create a list of arbitrary size, invoke the 'partition' function, and check if the node was correctly placed.
- Repeat the 'partition' process until the whole list is sorted to verify each of the subsequent partitions also yields correct results.

### Test 'sort' function

- Create one element or zero element lists. The result should be the same as the input.
- Create lists that are already sorted, the result should be the same.
- Create lists in descending order, the result should be sorted in ascending order.
- Create lists of different sizes, the outputs should all be sorted in the ascending order.

## 4.4.3 User Interface testing with React Testing Library

“React Testing Library is a library for React that provides an intuitive and efficient API for testing React components. It is built on top of the DOM Testing Library and provides a more user-centered way of testing React components.”

### Application component

- Renders correctly without any crashes
- Router for navigation between Main, Tutorial, Test, and About pages works correctly

### Header component

- Renders the navigation links
- Navigates to the correct routes when links are clicked

### Main page component

- Renders input fields and buttons
- Displays error messages when input is invalid

### Tutorial page component

- Displays the default Array tutorial
- Displays other tutorials if the corresponding topic button is clicked

### Test page component

- Selects answers when clicked
- Submits answers, shows the score, and closes the score window

### About page component

- Renders correctly

## 4.4.4 Testing results

Figure 4.4 JUnit testing results

✓ testZeroAndOneElementList()	✓ testSortArray()
✓ testSwap()	✓ testReversedList()
✓ testUnlinkAndPrecede()	✓ testSortLinkedList()
✓ testSortDoublyLinkedList()	✓ testPartitionDoublyLinkedList()
✓ testSortedList()	✓ testSortedArray()
✓ testZeroAndOneElementArray()	✓ testSortedDList()
✓ testZeroAndOneElementDList()	✓ testPartitionLinkedList()
✓ testReversedArray()	✓ testPartitionArray()
✓ testReversedDList()	

Figure 4.5 React Testing Library results

```
✓ src/pages/Tutorial.test.jsx (3)
✓ src/pages/Test.test.jsx (3)
✓ src/Header.test.jsx (2)
✓ src/App.test.jsx (5)
✓ src/pages/AboutPage.test.jsx (1)
✓ src/pages/MainPage.test.jsx (2)

Test Files  6 passed (6)
Tests      16 passed (16)
```

# Chapter 5

## Conclusion

Short chapter to reflect back on the program. The initial plans, solutions, and challenges during the development are mentioned here. Additionally, this chapter contains some suggestions to improve the program with further plans for future work.

### 5.1 Conclusion and Future work

While working on this project I was reminded of the importance of algorithms and data structures. I realized once again that for students mastering theoretical concepts is as vital as learning to apply these concepts in practice. The main purpose was to present an interactive learning page with focus on the Quicksort algorithm. The development of the project was very challenging, particularly in setting up the correct connection between front and back end, and animating the data in a clean, user-friendly manner. Initially, I struggled to configure how to send each step of the sorting process from the backend to the frontend. While implementing the sorting algorithm itself was quite straightforward, capturing and animating each step of the process in detail turned out to be challenging. After some contemplation, I decided to create a list of animations, where each step will be represented as a string message. Another difficult part was receiving those messages and breaking them up into animations. Despite these challenges, the project goals were successfully achieved. The addition of Tutorial and Test pages further enrich the learning experience, serving as resources alongside the main content.

Looking ahead, I plan to implement several features to improve the user interaction and educational value. These include allowing the users to choose the pivot value for the algorithm, which will require ensuring stability for both one-way and two-way lists. I also aim to modify the animations to provide options of viewing only the main procedure, only the partitioning or both. Additionally, I am planning to give the option to see the steps as a video-animation. The process can keep going on its own, without the need of pressing ‘Next’ or ‘Previous’ buttons. With this, selecting the speed of animation, and pause/resume options can be added as well.

The scope of the project can be further extended to include a big variety of algorithms. Each algorithm can be implemented and shown with different data structures. This expansion

would involve adding new tutorials, tests, and practical tasks for each new algorithm. By introducing other sorting, searching, graph traversal, dynamic programming, and other advanced algorithms, the program can offer a broader range for the learners' needs.

Eventually, I envision transforming this program into a comprehensive application where users can track their engagement, progress, and test results, fostering a more personalized learning journey. The application can be a great tool for learners, providing hands-on experience in understanding the practical application of algorithms and data structures. This not only aids in the comprehension but also fosters a deeper appreciation of concepts in computer science.

# References

1. *Algorithms and Data Structures I. Lecture Notes*, Tibor Ásványi, last update - 2024.02.15, [AlgDs1LectureNotes-2024-02-15.pdf](#)
2. *Understanding the Quicksort Algorithm*, 2019.05.17, [Quicksort Algorithm | DeepAI](#)
3. *Linked List Data Structure*, last update - 2024.04.10, [Linked List Data Structure](#)
4. *Introduction to Doubly Linked List - Data Structure and Algorithm Tutorial*, last update - 2024.03.18, [Introduction to Doubly Linked List – Data Structure and Algorithm Tutorials](#)
5. *Types of Linked List in Data Structures*, last update - 2023.04.10, [Types of Linked List in Data Structures](#)
6. *What is Java?*, last update - 2024.03.13, [What Is Java?](#)
7. *Introduction to Java*, last update - 2024.05.01, [Introduction to Java](#)
8. *Spring Boot*, published date - 2017.06.06, [Spring Boot](#)
9. *React (JavaScript library)*, published date - 2015.01.02, [React \(JavaScript library\)](#)
10. *React Components*, last update - 2024.04.05, [React Components](#)
11. *Making HTTP requests with Axios*, last update - 2023.10.24, [Making HTTP requests with Axios](#)
12. *Using with React Spring*, last update - 2023.08.20, [React Three Fiber Documentation](#)
13. *Why you should use react-spring in your React application*, 2021.09.14, [Why you should use react-spring in your React application. | by John Sangalang](#)
14. *Strategy*, [Strategy](#)
15. *JUnit Tutorial*, [JUnit Tutorial](#)
16. *Quicksort-notes*, Clifford Stein, Fall 2003, [quicksort-notes.pdf](#)
17. *Unit Testing with the React Testing Library*, 2023.03.20, [Unit Testing with the React Testing Library](#)

# List of Figures

2.1 Partitioning, recursion, combine (google images).....	8
2.2 Quicksort time complexities.....	8
2.3 Quicksort for array pseudocode.....	9
2.4 Quicksort for linked list with a header node pseudocode.....	10
2.5 Cyclic two-way list.....	11
2.6 Quicksort for cyclic two-way list pseudocode.....	11
3.1 Main page basic flowchart.....	12
3.2 Tutorial page flowchart.....	13
3.3 Test page flowchart.....	13
3.4 Main Page.....	14
3.5 Incorrect inputs.....	15
3.6 Correct inputs on the Main Page.....	15
3.7 Array sorting demonstration.....	16
3.8 End of partition.....	16
3.9 End of quicksort.....	17
3.10 'View history' button.....	17
3.11 Singly linked list sorting demonstration.....	18
3.12 Doubly linked list sorting demonstration.....	18
3.13 Tutorial page layout.....	19
3.14 Test page layout.....	19
3.15 Score window.....	20
3.16 Problems page.....	20
3.17 About page layout.....	21
4.1 Use case diagram.....	25
4.2 Architecture of the program.....	26
4.3 Backend logic class diagram.....	28
4.4 Junit testing results.....	34
4.5 React Testing Library results.....	34