# Kongu Engineering College KONGU ENGINEERING COLLEGE Kongu Engineering College (Autonomous)

(Autonomous)

Perundurai,Erode – 638060

**DEPARTMENT OF INFORMATION TECHNOLOGY**

**Quicksort – Best and Worst Case Analysis**

**A MICRO PROJECT REPORT**

**FOR**

**DESIGN AND ANALYSIS OF ALGORITHMS(22ITT31)**

**SUBMITTED BY**

**YOKARAJAN R (23ITR180)**

** KONGU ENGINEERING COLLEGE**  (Autonomous)

Perundurai,Erode – 638060

**DEPARTMENT OF INFORMATION TECHNOLOGY**

**Quicksort – Best and Worst Case Analysis**

**A MICRO PROJECT REPORT**

**FOR**

**DESIGN AND ANALYSIS OF ALGORITHMS(22ITT31)**

**SUBMITTED BY**

**YOKARAJAN R (23ITR180)**

# Kongu Engineering College KONGU ENGINEERING COLLEGE Kongu Engineering College (Autonomous)

# (Autonomous)

Perundurai,Erode – 638060

**DEPARTMENT OF INFORMATION TECHNOLOGY**

**BONAFIED CERTIFICATE**

|  |  |
| --- | --- |
| Name | : YOKARAJAN R |
| Course Code | : 22ITT31 |
| Course Name | : DESIGN AND ANALYSIS OF ALGORITHMS |
| Semester | : IV |

Certified that this is a bonafide record of work for application project done by the above student for 22ITT31-DESIGN AND ANALYSIS OF ALGORITHMS during the academic year 2024-2025.

Submitted for the Viva Voice Examination held on \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Faculty Incharge Head of the Department

## ABSTRACT

## The Quicksort algorithm is a fundamental example that highlights the power and limitations of the divide and conquer approach in sorting problems. This project presents an analysis of Quicksort’s performance under different input scenarios, including arrays with all equal elements and strictly decreasing sequences. By simulating these cases, the project demonstrates how input arrangement affects the time complexity—ranging from the efficient O(n log n) in the best case to the slower O(n²) in the worst case. The algorithm works by selecting a pivot, partitioning the array, and recursively sorting the subarrays. Through this analysis, the project fosters a deeper understanding of algorithm efficiency, input sensitivity, and design strategies. It also lays a foundation for exploring optimization techniques and serves as a valuable learning tool for algorithm analysis.

## TABLE OF CONTENTS

|  |  |  |
| --- | --- | --- |
| **CHAPTER NO** | **TITLE** | **PAGE NO** |
|  | **ABSTRACT** | **1** |
| **1.** | **INTRODUCTION** | 3 |
|  | 1.1 PURPOSE | 4 |
|  | 1.2 OBJECTIVE | 4 |
|  | 1.3 METHODOLOGY OVERVIEW | 5 |
| **2.** | **PROBLEM STATEMENT** | 6 |
| **3.** | **METHODOLGY** | 7 |
|  | 3.1 Input & Initialization | 7 |
|  | 3.2 **Divide & Compare** | 7 |
|  | 3.3 Recursive Detection | 7 |
|  | 3.4 Visualization & Output | 8 |
|  | 3.5 Algorithm | 8 |
| **4.** | **IMPLEMENTATION** | 9 |
|  | 4.1 Input & Initialization | 9 |
|  | 4.2 Divide & Compare | 9 |
|  | 4.3 Recursive Detection | 10 |
|  | 4.4 Visualization & Output | 10 |
| **5.** | **RESULTS** | 13 |

## INTRODUCTION

Sorting algorithms play a vital role in computer science, with applications spanning data processing, searching, and optimization tasks. Among them, Quicksort stands out as a highly efficient and widely used algorithm, particularly valued for its average-case performance.

In this project, we analyze the behavior of the Quicksort algorithm under different input conditions to understand how input structure influences performance. Using the divide-and-conquer strategy, Quicksort selects a pivot element, partitions the array into subarrays of smaller and greater elements, and recursively sorts them. While the best and average cases offer optimal performance (O(n log n)), the algorithm can degrade to O(n²) in the worst-case scenario—such as when the input is already sorted in reverse.

This project includes simulations of sorting arrays with specific patterns (e.g., all elements equal, strictly decreasing order) to observe and compare Quicksort’s efficiency. The goal is to help learners visualize how algorithmic decisions impact runtime and to promote a deeper understanding of algorithm design and analysis.

* 1. **PURPOSE**

The main purpose of this project is to:

* Make the concept of sorting algorithms, particularly Quicksort, more accessible and easy to understand for students and beginners.
* Illustrate the power and limitations of the Divide and Conquer technique in algorithm design.
* Demonstrate how input data structure affects the performance of an algorithm.
* Support academic learning by providing clear examples of best, worst, and average case scenarios in sorting.

## OBJECTIVE

The main objective of this project is to analyze and understand the performance of the Quicksort algorithm under different input conditions using the Divide and Conquer strategy. The specific goals include:

* To simulate Quicksort behavior on arrays with different patterns (e.g., equal elements, decreasing order).
* To identify and explain the best, worst, and average case complexities of Quicksort.
* To promote algorithmic thinking by exploring how pivot selection and input structure impact performance.
* To present the results in a simplified and educational manner that enhances conceptual clarity for learners.

## METHODOLOGY OVERVIEW

## The project follows a systematic approach to analyze the performance of the Quicksort algorithm using the Divide and Conquer strategy. The key steps in the implementation are:

## User Input:

## The user provides an input array of real numbers. Different input patterns such as all equal values, strictly decreasing order, and random values are tested to observe performance variations.

## Pivot Selection:

## A pivot element is chosen from the array (commonly the last or middle element). This pivot is used to divide the array into subarrays of elements less than and greater than the pivot.

## Partitioning Process:

## The array is partitioned based on comparisons with the pivot. Elements smaller than the pivot move to the left; larger ones move to the right. This is the core step of Quicksort’s divide and conquer logic.

## Recursive Sorting:

## The left and right subarrays formed after partitioning are recursively sorted using the same Quicksort procedure.

## Base Case:

## The recursion terminates when subarrays have zero or one element, which are inherently sorted.

## Performance Observation:

## The algorithm is executed on different input types, and its behavior (best, average, and worst cases) is observed and analyzed in terms of time complexity and number of recursive calls.

## 2. PROBLEM STATEMENT

The **Quicksort Performance Analysis Problem** explores how the input structure of an array affects the efficiency of the Quicksort algorithm—a classic divide and conquer sorting technique. Given an array A[0..n−1] of n real numbers, Quicksort recursively selects a pivot, partitions the array into elements less than and greater than the pivot, and sorts the subarrays.

This project investigates how specific input patterns—such as arrays with all equal elements or strictly decreasing sequences—impact the algorithm's performance. In particular, it examines whether these patterns result in the **best case**, **worst case**, or fall under **neither** based on the number of comparisons and recursive calls.

The analysis is carried out by applying Quicksort to different types of inputs and observing how the algorithm behaves in each scenario. The goal is to enhance understanding of algorithmic behavior, efficiency classes, and the significance of input arrangement in divide and conquer strategies.

**3.0 Quicksort Performance Analysis Methodology**

**3.1 Input & Initialization**

* Accept an array A[0..n−1] of n real numbers from the user.
* Prepare different input scenarios:
* All elements equal (e.g., [5, 5, 5, 5])
* Strictly decreasing order (e.g., [9, 7, 5, 3, 1])
* Random order
* Choose a pivot selection strategy (e.g., last element as pivot).
* Initialize variables to track number of comparisons and recursive calls.

**3.2 Divide & Compare**

* Select a pivot element from the array.
* Partition the array into two groups:
* Elements less than or equal to the pivot
* Elements greater than the pivot
* Rearrange elements around the pivot to ensure correct relative ordering.
* Compare elements with the pivot and count the number of operations.

**3.3 Recursive Detection**

* Apply the same divide-and-partition logic recursively on the left and right subarrays.
* Continue recursion until subarrays have one or zero elements (base case).
* Keep track of recursion depth and number of steps.
* Analyze how input type affects the recursion structure and efficiency.
  1. **Visualization & Output**
* Provide a step-by-step visualization of how the array is partitioned and sorted.
* After sorting, display:
* The sorted array
* Total number of comparisons and recursive calls
* Time complexity classification (Best, Worst, or Average Case)
* Explanation of why a particular input resulted in that case
  1. **Algorithm**

**QuickSort**

function quickSort(arr, low, high):

if low < high:

pivotIndex = partition(arr, low, high)

quickSort(arr, low, pivotIndex - 1)

quickSort(arr, pivotIndex + 1, high)

**Partition**

function partition(arr, low, high):

pivot = arr[low]

i = low + 1

j = high

while true:

while i <= j and arr[i] <= pivot:

i += 1

while j >= i and arr[j] > pivot:

j -= 1

if i < j:

swap(arr[i], arr[j])

else:

break

swap(arr[low], arr[j])

return j

**IMPLEMENTATION :**

**4.1 Input & Initialization**

const arr = [9, 7, 5, 3, 1];

let comparisons = 0;

let recursiveCalls = 0;

**4.2 Divide & Compare**

function partition(array, low, high) {

const pivot = array[high];

let i = low - 1;

for (let j = low; j < high; j++) {

comparisons++;

if (array[j] <= pivot) {

i++;

[array[i], array[j]] = [array[j], array[i]];

}

}

[array[i + 1], array[high]] = [array[high], array[i + 1]];

return i + 1;

**4.3 Recursive Detection**

function quicksort(array, low, high) {

if (low < high) {

recursiveCalls++;

const pi = partition(array, low, high);

quicksort(array, low, pi - 1);

quicksort(array, pi + 1, high);

}

}

**4.4 Visualization & Output**

quicksort(arr, 0, arr.length - 1);

console.log("Quicksort Complete!");

console.log("Sorted array:", arr);

console.log(`Total comparisons: ${comparisons}`);

console.log(`Total recursive calls: ${recursiveCalls}`);

**DIFFERENCE BETWEEN BRUTEFORCE AND DIVIDE AND CONQUER:**

**Brute Force:**

**Concept:**

* Compare every pair of elements and swap if out of order.
* Check all possible pairs until the entire array is sorted.

**How it works:**

1. Start from the first element and compare it with the next.
2. Swap if they are in the wrong order.
3. Repeat this process for all pairs, multiple passes until no swaps are needed.
4. Continue until the array is fully sorted.

**Time Complexity:**

* Worst-case: O(n²) comparisons and swaps.
* Inefficient for large arrays due to repeated comparisons.

**Divide and Conquer Approach (Divide into Three Groups)**

**Concept:**

* Select a pivot element.
* Partition the array into two parts: elements less than pivot and elements greater than pivot.
* Recursively sort the subarrays.

**How it works:**

* Pick a pivot element from the array.
* Rearrange the array so that elements less than the pivot come before it, and elements greater come after.
* Recursively apply the same steps to the left and right subarrays.
* Continue until the subarrays have only one or zero elements (already sorted).

**Time Complexity:**

* Best/Average Case: O(n log n) comparisons.
* Worst Case (e.g., sorted or reverse sorted array with bad pivot): O(n²) comparisons.

**Pros:**

* Efficient on average, especially for large datasets.
* Uses fewer comparisons than brute force methods.
* Recursive structure makes problem easier to solve by breaking it down.

**Cons:**

* Performance depends on pivot choice.
* Worst-case can degrade to O(n²), but this can be minimized by good pivot strategies.

|  |  |  |
| --- | --- | --- |
| **Feature** | **Brute Force** | **Divide and Conquer** |
| **Strategy** | Pairwise comparisons and swaps | Recursive splitting and sorting |
| **Time Complexity** | O(n²) | O(n log n) on average |
| **Efficiency** | Low | High |
| **Ideal for** | Very small or sorted arrays | Large & Random arrays |
| **Use of Recursive Calls** | None | Extensive recursion |
| **Logic Complexity** | Simple | Moderate |

**5.0. RESULTS:**

**Step 1: Choose a Pivot**

The algorithm starts by selecting a pivot element:

* This can be the first, last, middle, or a random element in the array.
* In many implementations, the last element is chosen as the pivot.

**Step 2: Partition the Array**

Rearrange the array so that:

* All elements less than the pivot come before it.
* All elements greater than or equal to the pivot come after it.
* The pivot is placed at its correct sorted position.

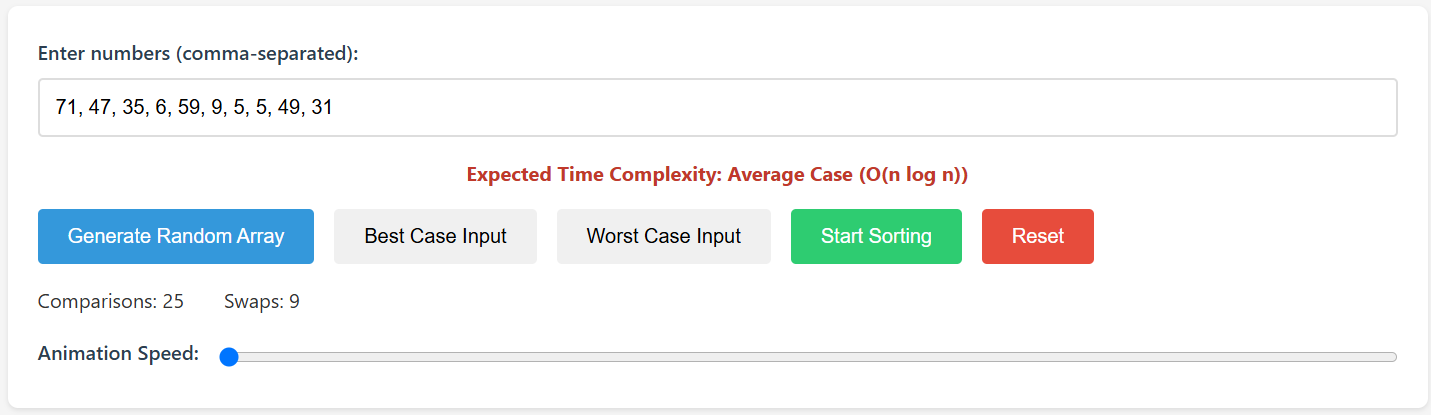
**Step 3: Recursively Sort Subarrays**

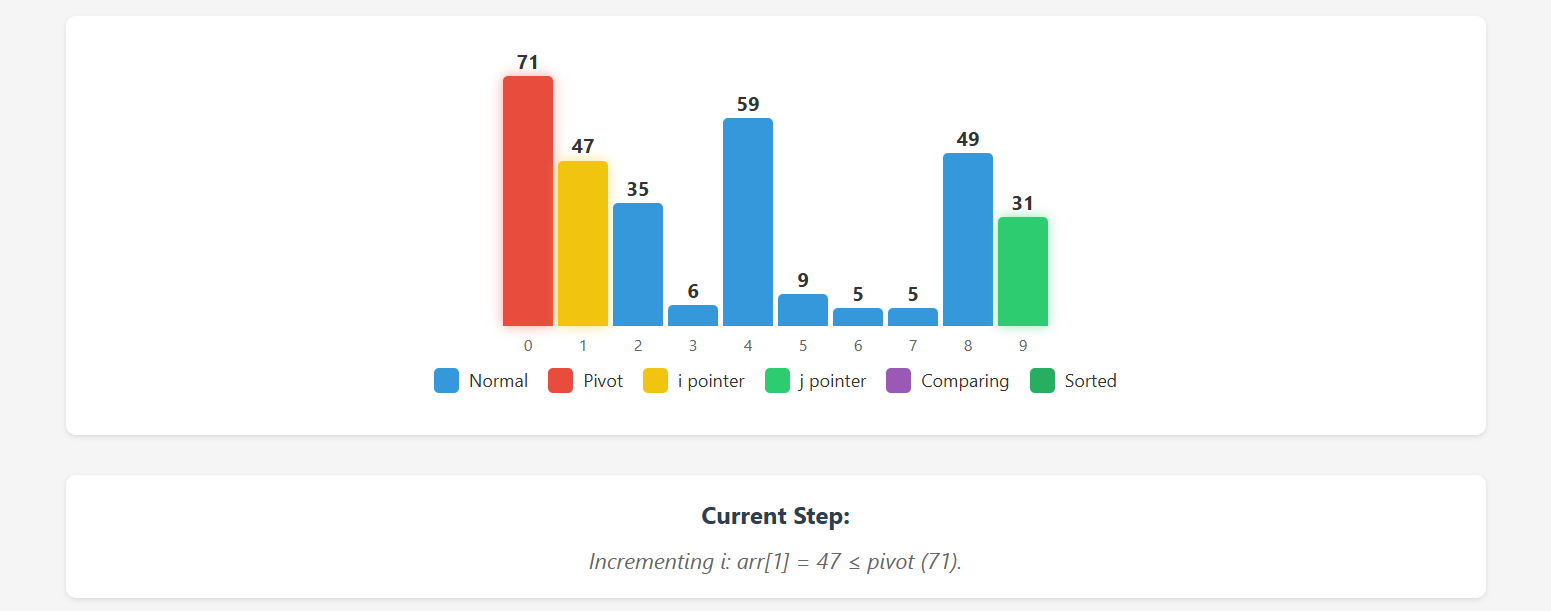
Apply the same quicksort algorithm to:

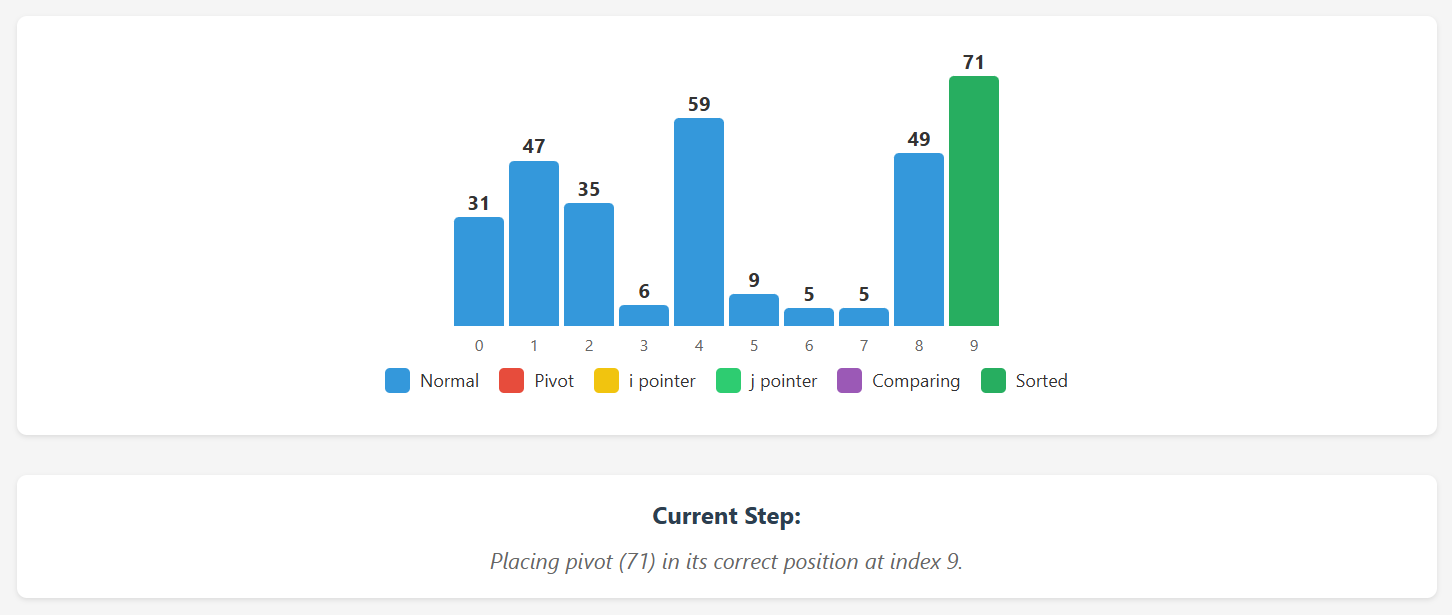
1. The subarray left of the pivot.
2. The subarray right of the pivot.

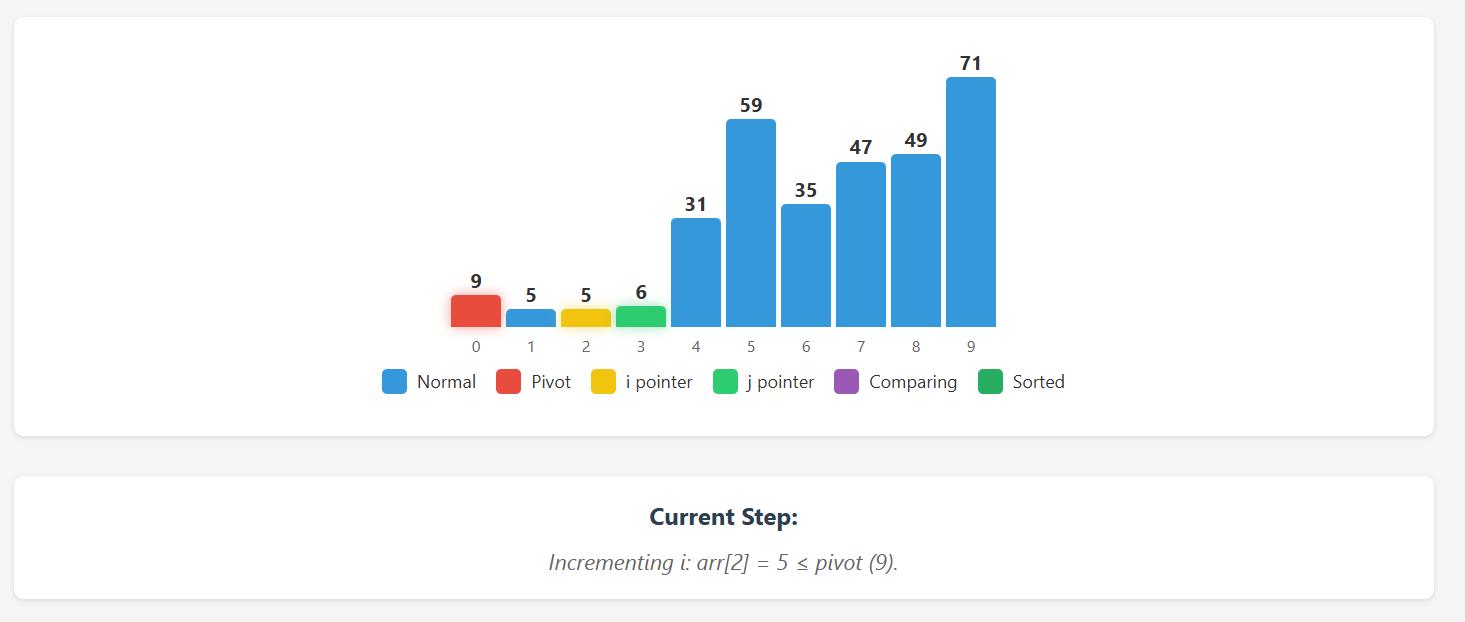
**Step 4: Combine the Results**

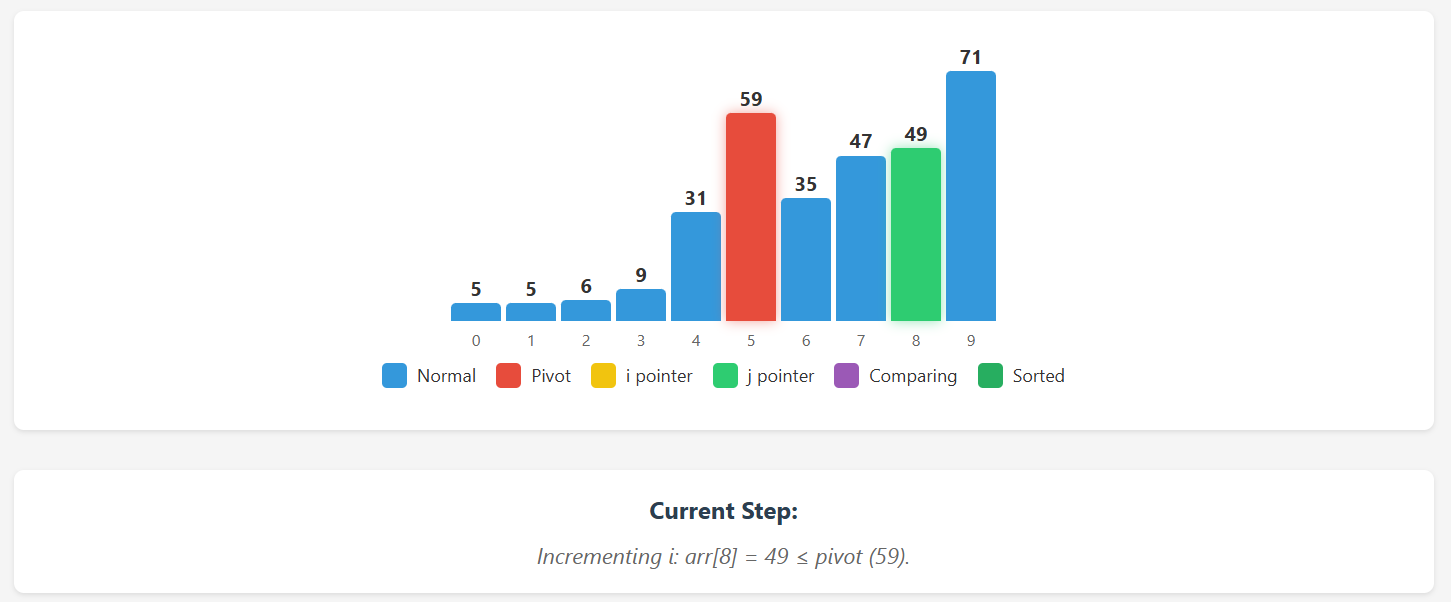
* After the recursive calls, the subarrays and the pivot are all in correct order.
* The fully sorted array is now obtained.

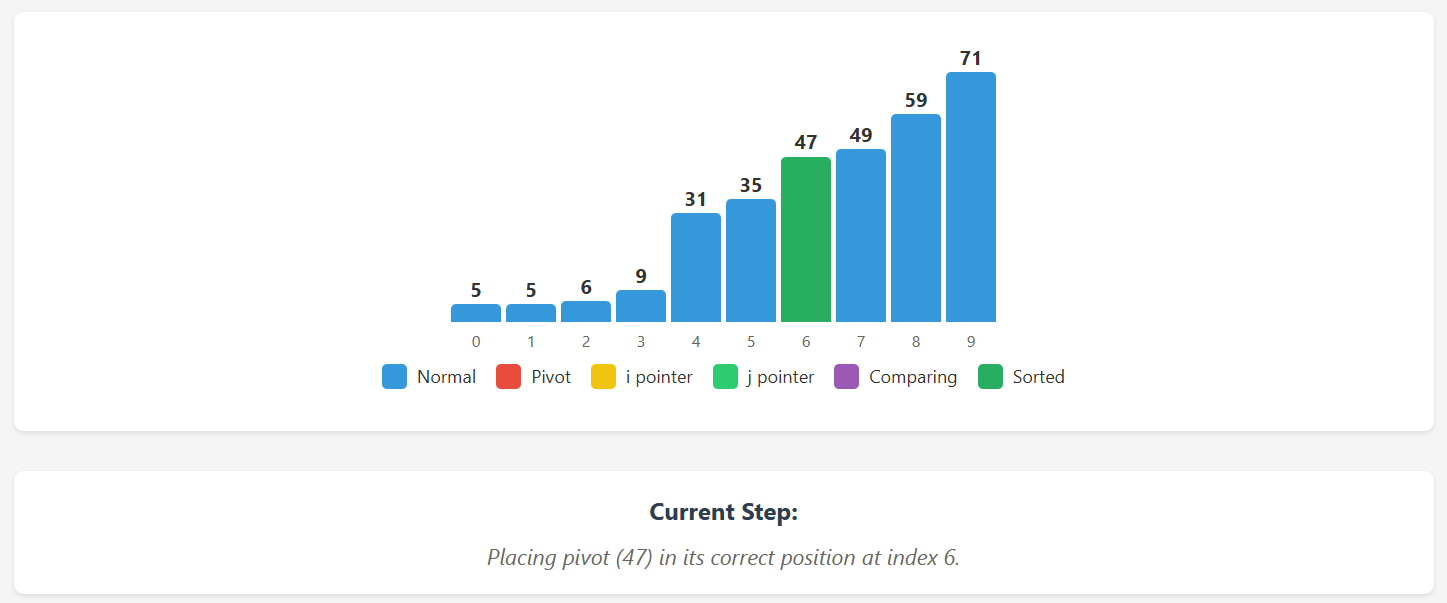
****

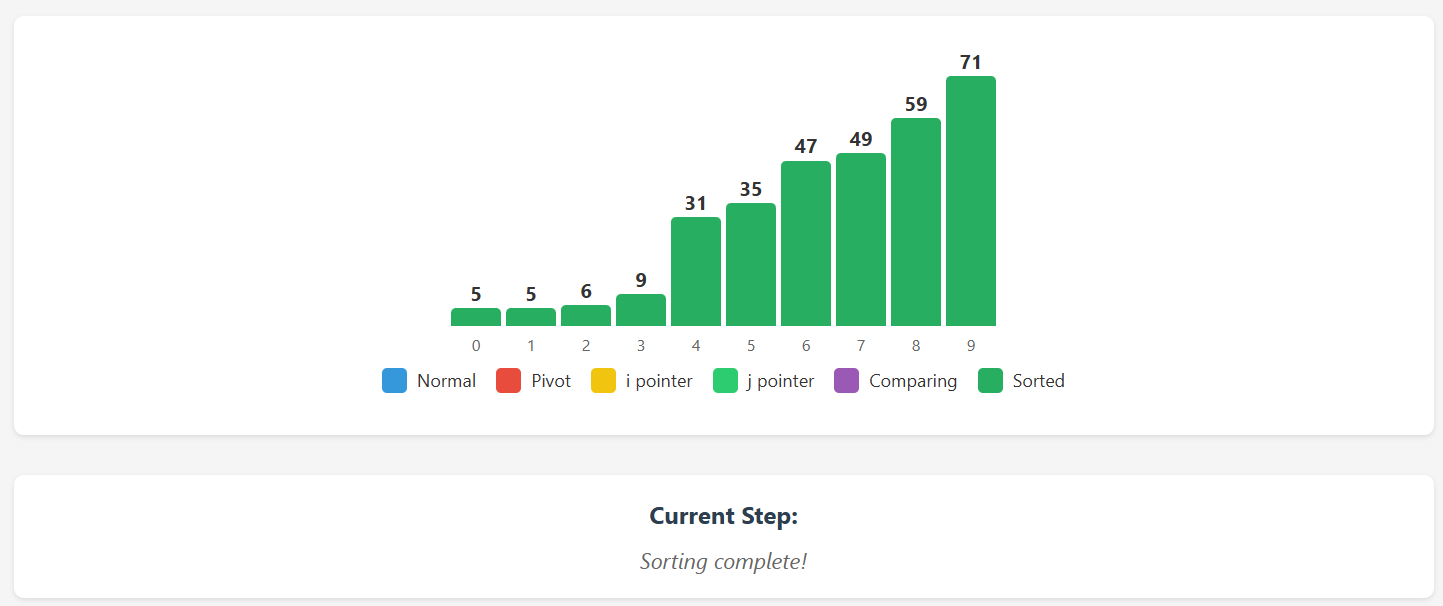
****

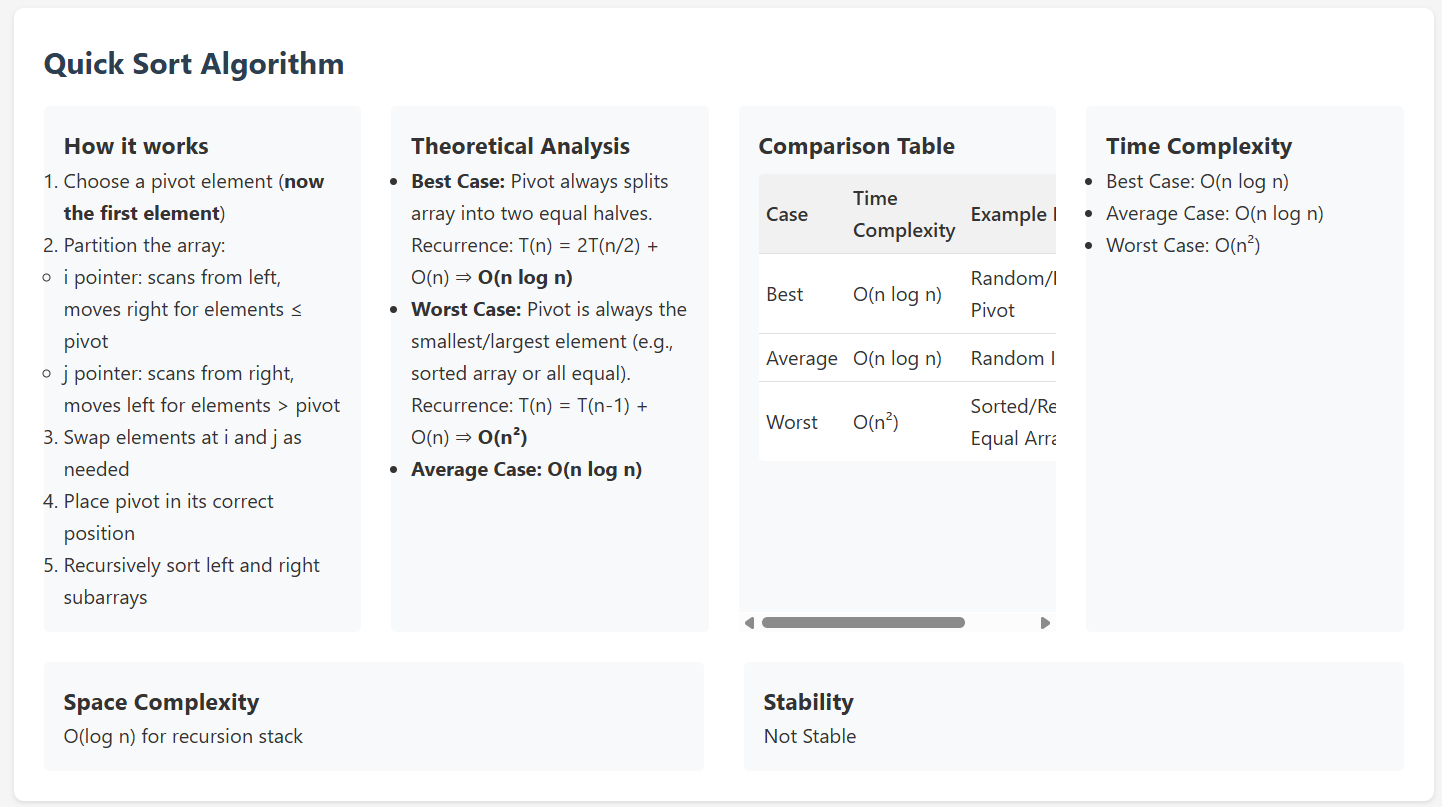
****

****

****

****

****



**GITHUB LINK: https://github.com/yokarajanr/DAA\_Microproject**