KONGU ENGINEERING COLLEGE

(Autonomous)
Perundurai,Erode – 638060

DEPARTMENT OF INFORMATION TECHNOLOGY

ALTERNATING DISKS SORTING PUZZLE USING BRUTE FORCE
A MICRO PROJECT REPORT

FOR

DESIGN AND ANALYSIS OF ALGORITHMS (22ITT31)

SUBMITTED BY
SHALINI G R (23ITR150)

KONGU ENGINEERING COLLEGE

(Autonomous)

Perundurai,Erode – 638060

DEPARTMENT OF INFORMATION TECHNOLOGY

ALTERNATING DISKS SORTING PUZZLE USING BRUTE FORCE
A MICRO PROJECT REPORT

FOR

DESIGN AND ANALYSIS OF ALGORITHMS (22ITT31)

SUBMITTED BY
SHALINI G R (23ITR150)

KONGU ENGINEERING COLLEGE

(Autonomous)

Perundurai,Erode – 638060

DEPARTMENT OF INFORMATION TECHNOLOGY

BONAFIDE CERTIFICATE

Name	: SHALINI G 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 Alternating Disks Problem is a classical puzzle that involves sorting a sequence of 2n disks—n dark and n light—arranged in an alternating pattern (dark, light, dark, light, ...). The objective is to segregate the disks such that all light disks are positioned on the left side and all dark disks are on the right, using only adjacent swaps. This problem illustrates fundamental concepts in algorithm design and combinatorial optimization. We propose a simple and effective algorithm based on iterative sweeps, where we perform multiple passes through the sequence: in each pass, we alternate between comparing and swapping odd-indexed and even-indexed pairs. During each pass, adjacent disks are compared and swapped if they are out of the desired order (i.e., a dark disk appears before a light disk). This continues for n iterations, where each iteration brings the configuration closer to the goal state. The algorithm guarantees convergence and completes in exactly n^2 moves, making it both deterministic and efficient. This solution serves as a pedagogical example for understanding sorting under restricted operations and has practical implications in robotics and automated rearrangement tasks where only local actions are permitted.

TABLE OF CONTENTS

1. INTRODUCTION

Sorting algorithms are fundamental to computer science, and visualizing their behavior can greatly enhance understanding of their inner workings. The disk sorting problem provides a compelling and intuitive platform for this purpose. It involves a sequence of alternating dark and light disks arranged in a line (e.g., DLDLDL...), with the objective of sorting them such that all light disks move to the left and all dark disks to the right (LLLLDDD...). The unique constraint is that only adjacent disks can be swapped, mimicking realistic physical limitations.

This project presents an interactive, web-based visualization tool that demonstrates the step-by-step execution of three sorting algorithms—**Bubble Sort**, **Quick Sort**, and **Merge Sort**—as applied to the disk arrangement problem. Each algorithm is adapted to adhere to the adjacent-swap constraint, and their operations are animated for better comprehension. By observing the movement of disks over time, users can compare algorithm efficiency, number of swaps, and behavioral patterns. The tool serves both as an educational aid for learners and a comparative study for researchers, offering insights into the strengths and limitations of each algorithm in a constrained sorting environment.

1.1 PURPOSE

- This project aims to provide a visual and interactive demonstration of various sorting algorithms applied specifically to the disk sorting problem, where disks of two types (dark and light) are arranged in an alternating sequence.

- The primary goal is to visually showcase how different sorting algorithms behave in real-time when constrained to only adjacent swaps.

- This tool serves an educational purpose by allowing users to understand and compare algorithm performance and behavior, making abstract concepts more tangible.

1.2 OBJECTIVE

- The main objective of this project is to compare the efficiency and behavior of Bubble Sort, Quick Sort, and Merge Sort when adapted to the disk sorting problem.

- By implementing visual, step-by-step execution of each algorithm, users can observe how each approach handles the constraints of the problem and how performance varies in terms of steps taken and time consumed.

- The project is intended to help students and educators grasp the nuances of algorithm design, performance trade-offs, and optimization.

1.3 METHODOLOGY OVERVIEW

The project is implemented as a web-based application that utilizes interactive visualizations to animate each step of the sorting process. Key metrics such as the number of swaps and time taken are tracked to evaluate the efficiency of each algorithm. A comparative analysis is then conducted to provide insights into which algorithms are better suited for this specific constrained sorting task.

1. UserInput:
The user chooses how the initial disk configuration should be created:

- Option 1: Automatically alternate 'D' (Dark) and 'L' (Light) disks.
- Option 2: Manually enter a custom configuration (e.g., DLDLLD).
  The user is also asked whether they want to see an animation of the sorting process**.**

2. DiskInitialization:
Depending on the user's input:

- A list of alternating dark and light disks is generated using generate_disks(n).
- Or, a custom list is built by parsing the user’s input.
  This initial list is then used as the input to the sorting algorithms.

3. SortingAlgorithms:
Three disk sorting algorithms are implemented:

- **Bubble Sort:** Selects the minimum element in each pass and swaps it with the correct position.

- **Quick Sort:** Recursively partitions the list around a pivot, sorting elements smaller to the left and larger to the right.

- **Merge Sort:** Recursively divides the list into halves, sorts each half, and merges them to produce a sorted list.

4. PerformanceAnalysis:

Each algorithm is executed and timed using analyze_time(), which returns:

- The step-by-step states for visualization.
- The total time taken to complete the sort.
  This helps in comparing the time efficiency of the different approaches.

5. Visualization:

If animation is enabled:

- A matplotlib animation is shown, plotting the disk configurations for each algorithm over time.
- Each subplot displays how disks are sorted step-by-step for a specific algorithm.

6. ResultDisplay:

Once sorting and animation (if enabled) are complete:

- The execution time for each sorting algorithm is printed to the console. This allows the user to analyze and compare the computational performance of all three methods.

2. PROBLEM STATEMENT

The problem begins with an array of 2n disks, consisting of n dark disks and n light disks arranged in an alternating pattern: Dark, Light, Dark, Light, and so on. The goal is to rearrange the disks such that all light disks are moved to the left side of the array and all dark disks to the right side. However, the movement is restricted: only adjacent disks can be swapped. The performance of any sorting algorithm is measured by the number of such adjacent swaps required to reach the final configuration (LLLL...DDDD). The problem provides a simplified model for sorting under constraints and has real-world applications in robotic motion planning and restricted rearrangement scenarios.

3. METHODOLOGY

3.1 ALGORITHM SELECTION

Three well-known sorting algorithms were selected:

- Bubble Sort: A simple, comparison-based algorithm that repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order.

- Quick Sort: A divide-and-conquer algorithm that partitions the array around a pivot and recursively sorts subarrays. It requires significant adaptation for adjacent-swap constraints.

- Merge Sort: Another divide-and-conquer approach that merges sorted subarrays. Like Quick Sort, it needs to be modified to operate under the adjacent-swap restriction.

3.2 IMPLEMENTATION APPROACH

Each algorithm is adapted to only use adjacent swaps while sorting disks. The implementation includes step-by-step tracking of swaps and intermediate states. These steps are recorded for visualization and performance analysis. Performance metrics include the total number of adjacent swaps, time complexity approximations, and execution time on different configurations.

3.3 VISUALIZATION STRATEGY

Disks are represented using DOM elements (HTML divs or canvas shapes), with distinct colors or labels for dark and light disks. Animation features include highlighting current comparisons, showing swaps, and pausing between steps. The user interface allows for configuration changes such as the number of disks, animation speed, and choice of algorithm.

4. IMPLEMENTATION :

1. Initialization and Utility Functions

```
import matplotlib.pyplot as plt
import matplotlib.animation as animation
import time

def generate_disks(n):
    disks = []
    for i in range(n):
        disks.append('D')
        disks.append('L')
    return disks

def plot_disks(ax, disks, title):
    ax.clear()
    colors = ['black' if d == 'D' else 'white' for d in disks]
    ax.set_title(title)
    ax.set_xticks([])
    ax.set_yticks([])
    ax.set_xlim(-1, len(disks))
    ax.set_ylim(0, 1)
    for i, color in enumerate(colors):
        circle = plt.Circle((i, 0.5), 0.4, color=color, ec='black')
        ax.add_patch(circle)
```

2. Sorting Algorithm Implementations

Bubble Sort

```
def bubble_sort(disks):
    disks = disks.copy()
    steps = [disks.copy()]
    n = len(disks)
    for i in range(n):
```

```
 for j in range(n - 1 - i):
            if disks[j] == 'D' and disks[j+1] == 'L':
                disks[j], disks[j+1] = disks[j+1], disks[j]
                steps.append(disks.copy())
    return steps
```

Quick Sort using adjacent swaps (partition simulation):

python

CopyEdit

```
def quick_sort(disks):
    steps = [disks.copy()]

    def partition(disks, low, high):
        pivot = disks[high]
        i = low
        for j in range(low, high):
            if (pivot == 'L' and disks[j] == 'L') or (pivot == 'D' and disks[j] == 'L'):
                while j > i:
                    disks[j], disks[j-1] = disks[j-1], disks[j]
                    j -= 1
                    steps.append(disks.copy())
                i += 1
        while high > i:
            disks[high], disks[high-1] = disks[high-1], disks[high]
            high -= 1
            steps.append(disks.copy())
        return i

    def quick_sort_recursive(disks, low, high):
        if low < high:
            pi = partition(disks, low, high)
            quick_sort_recursive(disks, low, pi - 1)
            quick_sort_recursive(disks, pi + 1, high)

    disks = disks.copy()
    quick_sort_recursive(disks, 0, len(disks) - 1)
    return steps
```

```
def merge_sort(disks):
    steps = [disks.copy()]

    def merge(disks, l, m, r):
        left = disks[l:m+1]
        right = disks[m+1:r+1]
        i = 0
        j = 0
        k = l
        while i < len(left) and j < len(right):
            if left[i] == 'L':
                disks[k] = left[i]
                i += 1
            else:
                if right[j] == 'L':
                    disks[k] = right[j]
                    j += 1
                else:
                    disks[k] = left[i]
                    i += 1
            k += 1
            steps.append(disks.copy())
        while i < len(left):
            disks[k] = left[i]
            i += 1
            k += 1
            steps.append(disks.copy())
        while j < len(right):
            disks[k] = right[j]
            j += 1
            k += 1
            steps.append(disks.copy())

    def merge_sort_recursive(disks, l, r):
        if l < r:
            m = (l + r) // 2
            merge_sort_recursive(disks, l, m)
```

```
 merge_sort_recursive(disks, m + 1, r)
            merge(disks, l, m, r)

    disks = disks.copy()
    merge_sort_recursive(disks, 0, len(disks) - 1)
    return steps
```

3. Performance Analysis and User Input

```
def analyze_time(algorithm, disks):
    start = time.time()
    steps = algorithm(disks)
    end = time.time()
    return steps, end - start

if __name__ == "__main__":
    print("Choose disk initialization method:")
    print("1. Alternate 'D' and 'L' automatically")
    print("2. Enter custom disk configuration manually (e.g., DLDLLD)")
    choice = input("Enter 1 or 2: ").strip()

    if choice == "1":
        n_disks = int(input("Enter number of dark (and light) disks (e.g., 4): "))
        initial_disks = generate_disks(n_disks)
    elif choice == "2":
        while True:
            user_input = input("Enter sequence using only 'D' and 'L' (e.g., DLDLLD):
").strip().upper()
            if all(c in ('D', 'L') for c in user_input) and len(user_input) >= 2:
                initial_disks = list(user_input)
                break
            else:
                print("Invalid input. Please enter a valid sequence using only 'D' and 'L'.")
    else:
        print("Invalid choice. Exiting.")
        exit()
```

4. Animation and Result Display

```
animate_choice = input("Do you want to see the animation? (yes/no): ").strip().lower()
show_animation = animate_choice in ('yes', 'y')

bubble_steps, bubble_time = analyze_time(bubble_sort, initial_disks)
quick_steps, quick_time = analyze_time(quick_sort, initial_disks)
merge_steps, merge_time = analyze_time(merge_sort, initial_disks)

if show_animation:
    fig, axes = plt.subplots(3, 1, figsize=(10, 6))

    def update_all(i):
        if i < len(bubble_steps):
            plot_disks(axes[0], bubble_steps[i], f"Bubble Sort - Step
{i+1}/{len(bubble_steps)}")
        if i < len(quick_steps):
            plot_disks(axes[1], quick_steps[i], f"Quick Sort - Step {i+1}/{len(quick_steps)}")
        if i < len(merge_steps):
            plot_disks(axes[2], merge_steps[i], f"Merge Sort - Step
{i+1}/{len(merge_steps)}")

    ani = animation.FuncAnimation(
        fig,
        update_all,
        frames=max(len(bubble_steps), len(quick_steps), len(merge_steps)),
        interval=500,
        repeat=False
    )

    plt.tight_layout()
    plt.show()

print("\nTime Complexity Analysis:")
print(f"Bubble Sort Time: {bubble_time:.6f} seconds")
print(f"Quick Sort Time: {quick_time:.6f} seconds")
print(f"Merge Sort Time: {merge_time:.6f} seconds")
```

5. DIFFERENCE BETWEEN DIVIDE AND CONQUER & BRUTE FORCE

1. Concept

Feature	Divide and Conquer	Brute Force
Idea	Break the problem into smaller sub-problems, solve them recursively, and combine the results.	Try all possible solutions and pick the correct/best one.
Approach	Top-down: Divide → Solve → Combine	Exhaustive: Try everything
Optimization	Efficient and avoids redundant work	Inefficient, especially for large inputs

2. How It Works

Step	Divide and Conquer	Brute Force
Divide	Split input into smaller parts	No division – tackle the problem as a whole
Conquer	Solve each part recursively	Generate and test every possible solution
Combine	Merge the partial solutions to get the final result	Select the best/correct one among all

3. Time Complexity

Algorithm/Problem	Divide and Conquer Time	Brute Force Time
Merge Sort	O(n log n)	N/A (Brute force doesn't apply here)
Quick Sort (Average)	O(n log n)	$O(n^2)$ if done badly (like brute swap)

5.1 Sample Input:

1. Bubble Sort Style Disk Sort

Initial Configuration:
The starting arrangement is ['D', 'L', 'D', 'L'], with alternating dark (D) and light (L) disks. The goal is to move all light disks to the left and dark disks to the right.

Step 1:
We start from the left. At index 0 and 1, we find 'D' followed by 'L', which violates the desired order. So, we swap them. The configuration becomes ['L', 'D', 'D', 'L'].

Step 2:
Next, at index 1 and 2, the order 'D' and 'D' is fine—no swap. At index 2 and 3, we again see 'D' and 'L', so we swap them. The configuration updates to ['L', 'D', 'L', 'D'].

Step 3:
In the next pass, index 0 and 1 is 'L' and 'D'—fine. At index 1 and 2, we have 'D' and 'L', so we swap again. Now the disks are ['L', 'L', 'D', 'D'].

Step 4:
A final pass shows everything in order, so no more swaps are needed.

Final Result: ['L', 'L', 'D', 'D']

Algorithm:

```
BUBBLE_SORT(disks):
    repeat (n - 1) times:
        for i from 0 to n - 2:
            if disks[i] == 'D' and disks[i+1] == 'L':
                swap(disks[i], disks[i+1])
```

2. Quick Sort Style Disk Sort (Modified for 'L' < 'D')

Initial Configuration:
The starting configuration is ['D', 'L', 'D', 'L'].

Step 1 (Choose Pivot):
We select the last element 'L' as the pivot. Our goal is to place all 'L' (light disks) to the left

and all 'D' (dark disks) to the right.

Step 2 (Partitioning):
We compare from left to right. At index 0, 'D' > 'L' → no swap. At index 1, 'L' <= 'L' → swap with index 0 → ['L', 'D', 'D', 'L']. Then at index 2, 'D' > 'L' → no swap. Now, we swap the pivot ('L' at index 3) with the element at index 1 → ['L', 'L', 'D', 'D'].

Step 3 (Recurse Left and Right):
The left sub-array is ['L', 'L'], already sorted. The right sub-array is ['D', 'D'], already sorted.

Final Result: ['L', 'L', 'D', 'D']

Algorithm:

```
QUICK_SORT(disks, low, high):
    if low < high:
        pivot_index = PARTITION(disks, low, high)
        QUICK_SORT(disks, low, pivot_index - 1)
        QUICK_SORT(disks, pivot_index + 1, high)


PARTITION(disks, low, high):
    pivot = disks[high]
    i = low - 1
    for j from low to high - 1:
        if disks[j] <= pivot:  // L < D
            i = i + 1
            swap(disks[i], disks[j])
    swap(disks[i + 1], disks[high])
    return i + 1
```

3. Merge Sort Style Disk Sort

Initial Configuration:
We start with ['D', 'L', 'D', 'L'].

Step 1 (Divide):
We split the list into two halves: Left = ['D', 'L'], Right = ['D', 'L'].

Step 2 (Further Divide):
Each half is split again:

- Left → ['D'], ['L']
- Right → ['D'], ['L']

Step 3 (Merge Each Half):
We begin merging:

- ['D'] and ['L'] → compare: 'L' < 'D' → result is ['L', 'D']
- Same for the right half → result is ['L', 'D']

Step 4 (Final Merge):
We merge ['L', 'D'] and ['L', 'D'].
Compare L vs L → both fine.
Then compare D vs L → L comes before D. Final merge result becomes: ['L', 'L', 'D', 'D']

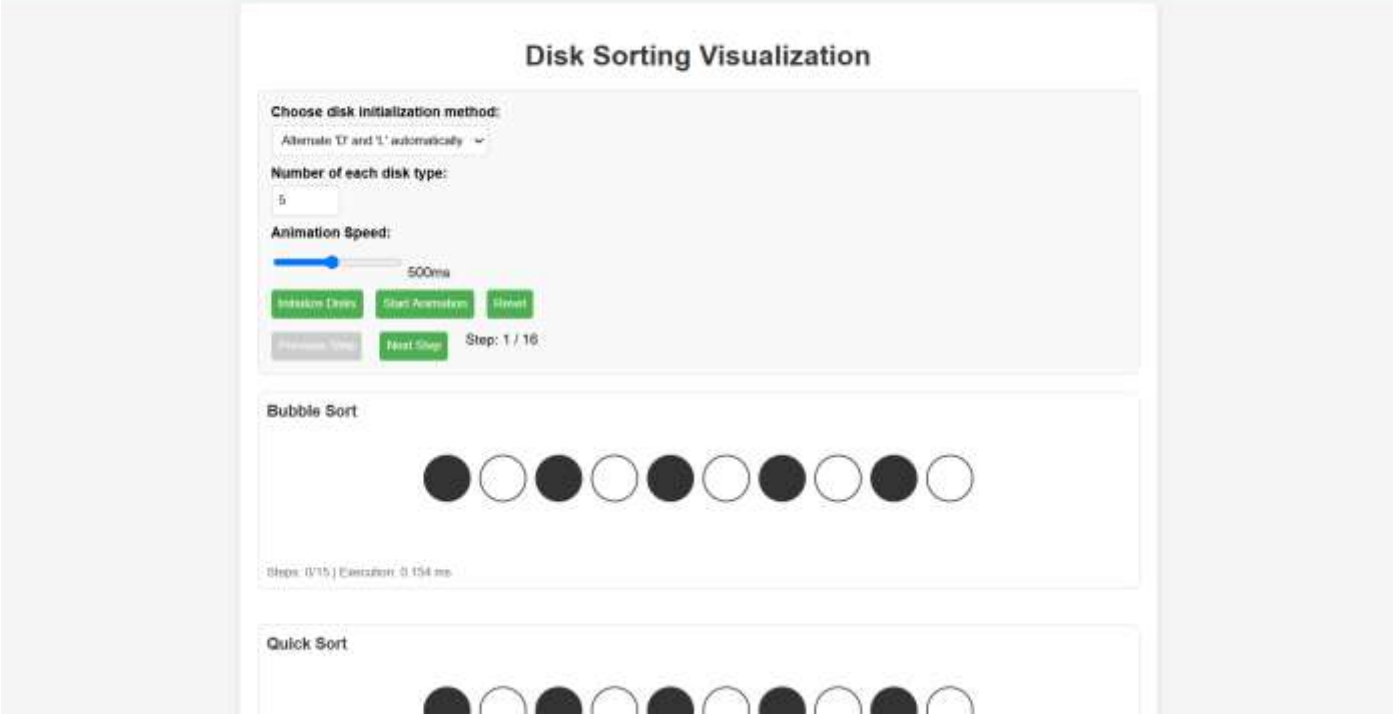
Final Result: ['L', 'L', 'D', 'D']

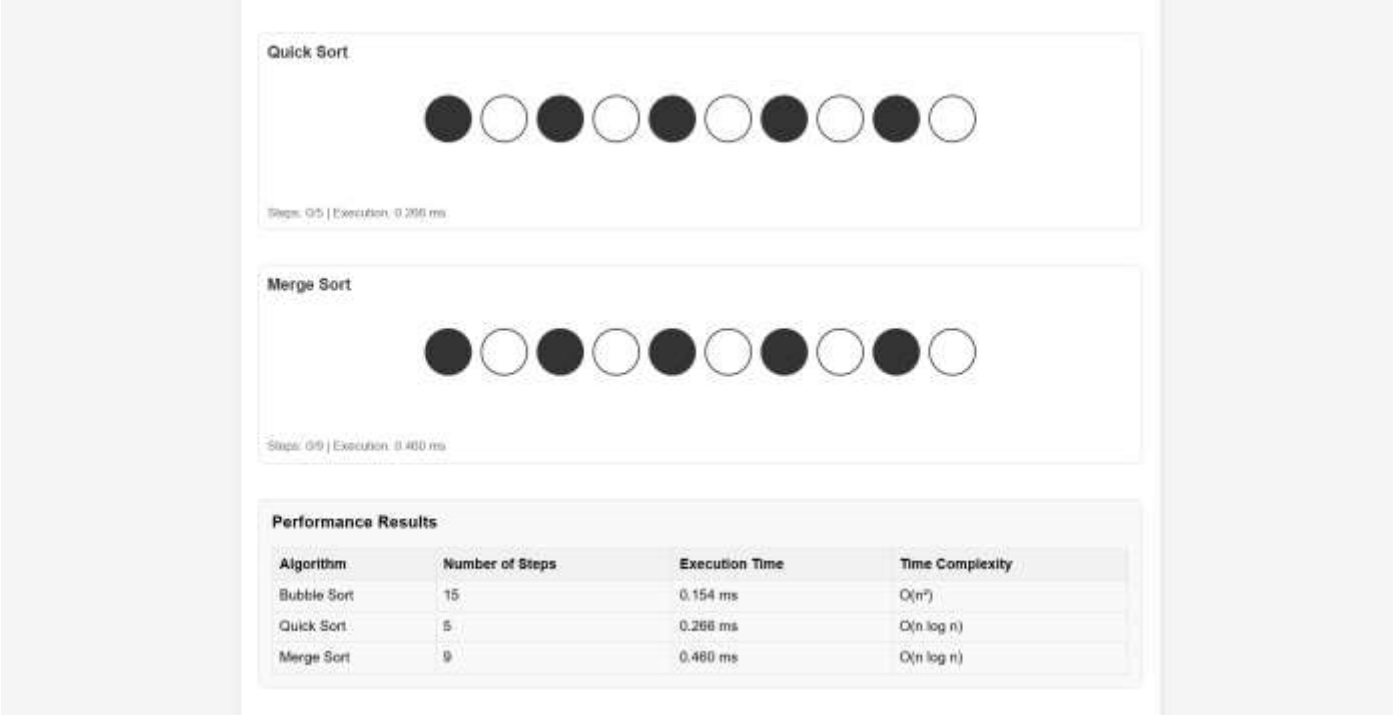
Algorithm:

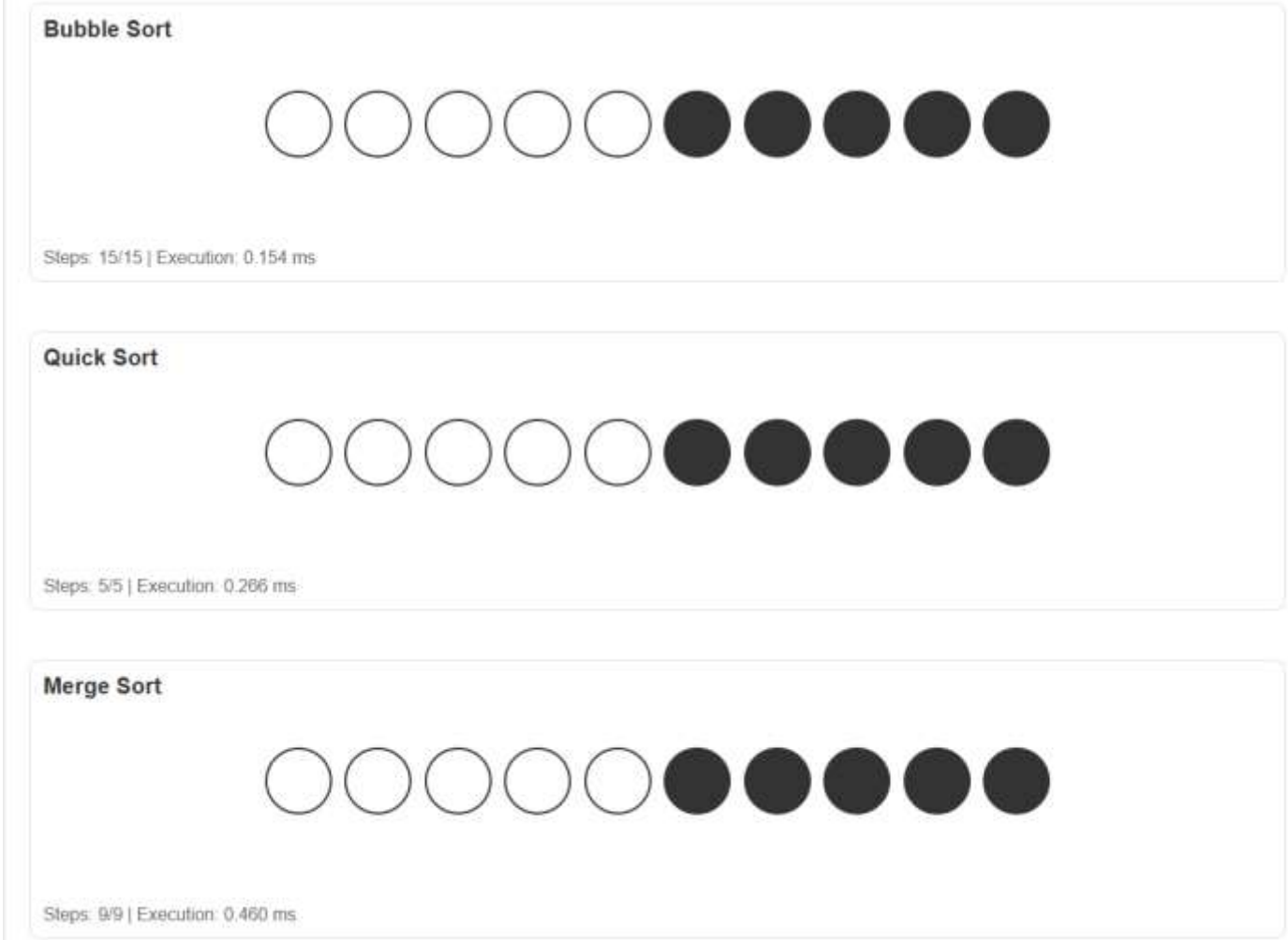
```
MERGE_SORT(disks):
    if length of disks > 1:
        mid = length // 2
        left = disks[0 to mid - 1]
        right = disks[mid to end]

        MERGE_SORT(left)
        MERGE_SORT(right)

        MERGE(left, right, disks)

MERGE(left, right, disks):
```

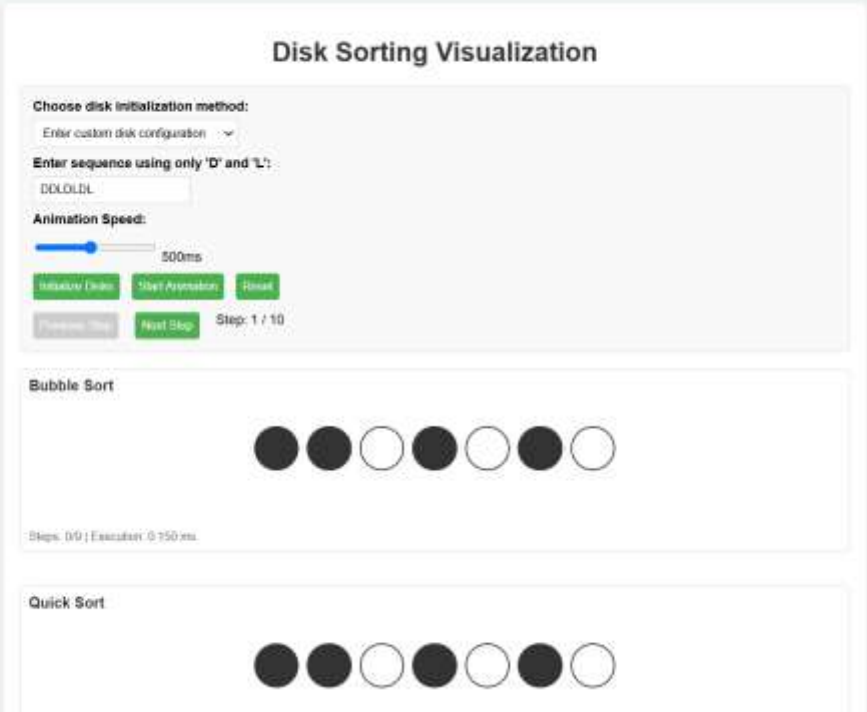
```
i = j = k = 0
  while i < len(left) and j < len(right):
      if left[i] <= right[j]:  // L < D
          disks[k] = left[i]
          i += 1
      else:
          disks[k] = right[j]
          j += 1
      k += 1

  // Copy remaining elements
  while i < len(left):
      disks[k] = left[i]
      i += 1
      k += 1
  while j < len(right):
      disks[k] = right[j]
      j += 1
      k += 1
```

6. RESULTS:


Disk Sorting Visualization
Choose disk initialization method:
Alternate 'D' and 'L' automatically
Number of each disk type:
5
Animation Speed:
500ms
Initialize Disks
Start Animation
Reset
Previous Step
Next Step
Step: 1 / 16
Bubble Sort
Steps: 0/15 | Execution: 0.154 ms
Quick Sort


Quick Sort
Steps: 0/5 | Execution: 0.266 ms
Merge Sort
Steps: 0/9 | Execution: 0.460 ms
Performance Results
Algorithm
Number of Steps
Execution Time
Time Complexity
Bubble Sort
15
0.154 ms
O(n²)
Quick Sort
5
0.266 ms
O(n log n)
Merge Sort
9
0.460 ms
O(n log n)


Bubble Sort
Steps: 15/15 | Execution: 0.154 ms
Quick Sort
Steps: 5/5 | Execution: 0.266 ms
Merge Sort
Steps: 9/9 | Execution: 0.460 ms


Disk Sorting Visualization
Choose disk initialization method:
Enter custom disk configuration
Enter sequence using only 'D' and 'L':
DDLDLDL
Animation Speed:
500ms
Initialize Disks
Start Animation
Reset
Previous Step
Next Step
Step: 1 / 10
Bubble Sort
Steps: 0/0 | Execution: 0.150 ms
Quick Sort

Steps: 0/3 | Execution: 0.254 ms

Merge Sort

Steps: 0/6 | Execution: 0.458 ms

Performance Results

Algorithm	Number of Steps	Execution Time	Time Complexity
Bubble Sort	9	0.150 ms	$O(n^2)$
Quick Sort	3	0.254 ms	O(n log n)
Merge Sort	6	0.458 ms	O(n log n)

Python Implementation

Bubble Sort

Steps: 9/9 | Execution: 0.150 ms

Quick Sort

Steps: 3/3 | Execution: 0.254 ms

Merge Sort

Steps: 6/6 | Execution: 0.458 ms

GITHUB LINK: https://github.com/ShaliniGR05/DAA_Micro_Project