KONGU ENGINEERING COLLEGE

(Autonomous)

Perundurai,Erode – 638060

DEPARTMENT OF INFORMATION TECHNOLOGY

FAKE COIN DETECTION USING DIVIDE AND CONQUER ALGORITHM

A MICRO PROJECT REPORT

FOR

DESIGN AND ANALYSIS OF ALGORITHMS (22ITT31)

SUBMITTED BY

SRIVARSHINI R (23ITR157)

KONGU ENGINEERING COLLEGE

(Autonomous)

Perundurai,Erode – 638060

DEPARTMENT OF INFORMATION TECHNOLOGY

FAKE COIN DETECTION USING DIVIDE AND CONQUER ALGORITHM

A MICRO PROJECT REPORT

FOR

DESIGN AND ANALYSIS OF ALGORITHMS(22ITT31)

SUBMITTED BY

SRIVARSHINI R (23ITR157)

KONGU ENGINEERING COLLEGE

(Autonomous)

Perundurai,Erode – 638060

DEPARTMENT OF INFORMATION TECHNOLOGY

BONAFIDE CERTIFICATE

Name	: SRIVARSHINI 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

Efficient sorting of mixed items is essential in industries like food distribution, where speed and accuracy directly impact productivity. This project addresses the challenge faced by a food distributor who receives shipments containing a mix of apples, oranges, and bananas. The objective is to reorder the fruits so that apples come first, followed by oranges, and bananas last.

We propose a linear-time algorithm that solves this problem efficiently using a single pass. Inspired by the Dutch National Flag problem, the solution employs a three-pointer approach that sorts the array in-place. It achieves a time complexity of **O(n)** and a space complexity of **O(1)**, making it suitable for large-scale data processing.

This algorithm is ideal for integration into real-world applications such as automated conveyor belt sorting systems. It eliminates the need for extra storage or multiple passes, which is critical in time-sensitive operations. The simplicity and performance of the method ensure both scalability and reliability.

By adopting this technique, food distributors can significantly improve their sorting speed, reduce manual labor, and enhance overall operational efficiency. This project demonstrates the practical value of algorithmic solutions in solving real-world logistic challenges.

TABLE OF CONTENTS

1.0 INTRODUCTION

In the food distribution industry, the efficient sorting and organization of produce is essential to streamline logistics and improve operational efficiency. This project addresses the problem faced by a food distributor who receives mixed shipments of three types of fruits: apples, oranges, and bananas. To ensure optimal processing and dispatch, the fruits must be separated and arranged in a specific order — apples first, followed by oranges, and bananas last.

We propose a linear-time algorithm that classifies and reorders the mixed shipment using a single pass through the data. This problem is conceptually similar to the Dutch National Flag problem, and our solution leverages a three-pointer approach to partition the array in-place with O(n) time complexity and O(1) space complexity. This ensures that the algorithm is both time-efficient and memory-efficient, making it suitable for large-scale industrial applications.

The algorithm's design makes it highly applicable in real-world scenarios such as conveyor belt sorting systems , where minimal latency and high throughput are critical. By implementing this solution, the distributor can significantly reduce sorting time and overall productivity.

1.1 PURPOSE

- The purpose of this project is to design an efficient algorithm that sorts a mixed shipment of three fruit types — apples, oranges, and bananas — in a specified order with minimal computational resources. The project aims to contribute to automation in food handling systems, where fast and memory-efficient solutions are essential..

1.2 OBJECTIVE

The main objective of this project is to design and implement an efficient algorithm to sort a mixed shipment of three types of fruits — apples, oranges, and bananas — in a specified order using a linear-time, in-place sorting technique. The specific goals include:

- To develop a sorting algorithm that arranges the fruits in the order: apples first, then oranges, and bananas last.
- To implement a single-pass, in-place sorting solution with **O(n)** time complexity and **O(1)** space complexity.
- To adapt the **three-pointer technique** inspired by the Dutch National Flag problem for practical, real-world categorization tasks.
- To promote algorithmic thinking by solving a real-world problem using fundamental computer science concepts.

1.3 METHODOLOGY OVERVIEW

The project implements a structured and efficient approach to sort a mixed collection of apples, oranges, and bananas using a linear-time, in-place algorithm. The sorting logic is inspired by the Dutch National Flag problem and is carried out through the following key steps:

1. **User Input:**
   The user provides the total number of fruits and a randomly mixed list containing apples, oranges, and bananas.

2. **Pointer Initialization:**
   Three pointers are initialized:
   - low to mark the boundary for apples,
   - mid to traverse the list, and
   - high to mark the boundary for bananas.
     This setup enables in-place sorting without additional memory.

3. **Single-Pass Traversal:**
   A loop runs while the mid pointer is less than or equal to high. Each fruit at the mid index is evaluated:
   - If it's an **apple**, it is swapped with the element at low, and both low and mid are incremented.
   - If it's an **orange**, only mid is incremented.
   - If it's a **banana**, it is swapped with the element at high, and high is decremented.

4. **In-Place Rearrangement:**
   The algorithm rearranges the list in-place, meaning no extra space is used beyond the three pointers.

5. **Time and Space Efficiency:**
   The algorithm ensures **O(n)** time complexity with **O(1)** space complexity, making it suitable for large-scale sorting.

6. **Result Output:**
   After the traversal, the fruits are successfully sorted in the required order: apples first, oranges second, bananas last.

2. PROBLEM STATEMENT

In the food distribution industry, shipments often arrive containing a random mixture of different fruit types. One such scenario involves mixed consignments of apples, oranges, and bananas. To streamline operations such as packaging, inventory management, and delivery, the distributor requires a method to organize the fruits in a specific order: **apples first**, followed by **oranges**, and finally **bananas**.

The challenge lies in developing an **efficient algorithm** that can perform this sorting in **linear time** while using **constant space**, ensuring suitability for large-scale, real-time industrial applications. The algorithm must process the list of fruits in a single pass and rearrange them in-place without using additional memory structures.

This project aims to solve this practical problem by designing and implementing a sorting algorithm inspired by the Dutch National Flag problem, ensuring optimal performance and scalability.

3.0 Fruit Sorting Problem Methodology

3.1 Input & Initialization

- Accept the total number of fruits (n) from the user.
- Receive the mixed list of fruits containing apples, oranges, and bananas in random order.
- Initialize three pointers: low, mid, and high.
- low and mid start at the beginning of the list, high starts at the end.

3.2 3 Pointer Technique(Single-Pass Sorting)

Traverse the list using the mid pointer while mid ≤ high:

- If the fruit at mid is an **apple**:
  - Swap it with the fruit at low.
  - Increment both low and mid.
- If the fruit at mid is an **orange**:
  - Leave it in place and increment mid.
- If the fruit at mid is a **banana**:
  - Swap it with the fruit at high.
  - Decrement high without incrementing mid (to check the swapped element).

3.3 In-Place Rearrangement

- Continue the process until mid passes high.
- At this point, the fruits are sorted: apples first, oranges second, bananas last.

3.4 Visualization & Output

- Provide a step-by-step visual or console-based representation of the sorting process, including pointer movements and swaps.
- Display the final sorted list of fruits.

FRUIT SORTING ALGORITHM (Dutch National Flag approach):

```
If length(arr) == 0 or 1:
    return arr   // Already sorted or empty
Initialize pointers:
    low = 0
    mid = 0
    high = length(arr) - 1
While mid <= high:
    If arr[mid] == "apple":
        Swap arr[low] and arr[mid]
        low = low + 1
        mid = mid + 1
    Else if arr[mid] == "orange":
        mid = mid + 1
    Else if arr[mid] == "banana":
        Swap arr[mid] and arr[high]
        high = high - 1

Return arr  // Sorted array: apples, oranges, bananas
```

BRUTE FORCE APPROACH ALGORITHM:

```
ALGORITHM BubbleSortFruit(fruits[])
   n ← length of fruits

   FOR i from 0 to n-1 DO
      FOR j from 0 to n-i-2 DO
         IF order(fruits[j]) > order(fruits[j+1]) THEN
            SWAP fruits[j] and fruits[j+1]
         END IF
      END FOR
   END FOR

   RETURN fruits[]
END ALGORITHM

FUNCTION order(fruit)
   IF fruit == "apple" THEN
      RETURN 1
   ELSE IF fruit == "orange" THEN
      RETURN 2
   ELSE IF fruit == "banana" THEN
      RETURN 3
   END IF
END FUNCTION
```

IMPLEMENTATION :

4.1 Input & Initialization

```
const fruits = ["banana", "apple", "orange", "banana", "apple", "orange"];

console.log("Before sorting:", fruits);
```

4.2 Divide & Compare

```
function sortFruits(arr) {
   let low = 0, mid = 0, high = arr.length - 1;
   function getOrder(fruit) {
      if (fruit === "apple") return 0;
      if (fruit === "orange") return 1;
      if (fruit === "banana") return 2;  }
   while (mid <= high) {
      const order = getOrder(arr[mid]);
      if (order === 0) {
         [arr[low], arr[mid]] = [arr[mid], arr[low]];
         low++; mid++;
      } else if (order === 1) {
         mid++;
      } else {
         [arr[mid], arr[high]] = [arr[high], arr[mid]];
         high--;
      }
   }
}
```

4.3 Three-Way Partitioning (Dutch National Flag Sorting Logic)

```
function getFruitOrder(fruit) {
    if (fruit === "apple") return 0;
    if (fruit === "orange") return 1;
    if (fruit === "banana") return 2;
}

// Dutch National Flag algorithm for fruit sorting
function dutchNationalFlagSort(fruits) {
    let low = 0;
    let mid = 0;
    let high = fruits.length - 1;

    while (mid <= high) {
        const order = getFruitOrder(fruits[mid]);

        if (order === 0) {
            // Apple (lowest priority) goes to the beginning
            [fruits[low], fruits[mid]] = [fruits[mid], fruits[low]];
            low++;
            mid++;
        } else if (order === 1) {
            // Orange (middle priority) stays in the middle
            mid++;
        } else {
```

```
            // Banana (highest priority) goes to the end
            [fruits[mid], fruits[high]] = [fruits[high], fruits[mid]];
            high--;
        }
    }

    return fruits;
}
```

4.4 Visualization & Output

```
            sortFruits(fruits);
            console.log("After sorting:", fruits);
```

DIFFERENCE BETWEEN BRUTEFORCE AND DIVIDE AND CONQUER:

Brute Force(Bubble Sort)

Concept:

- Repeatedly compare adjacent elements and swap them if they are in the wrong order.
- Gradually "bubble" the largest element to the end with each pass.
- Continue until the entire array is sorted.

How it works:

1. Compare adjacent pairs starting from the beginning.
2. Swap if the left element is greater than the right element.
3. After one full pass, the largest element settles at the end.
4. Repeat the process for the remaining unsorted portion.
5. Continue until no swaps are needed.

Time Complexity:

- Worst-case: $O(n^2)$ comparisons and swaps.
- Each element might be compared multiple times.

Pros:

- Simple to implement and understand.
- Good for small or nearly sorted arrays.

Cons:

- Very inefficient for large datasets.
- High number of redundant comparisons.

Dutch National Flag Algorithm (Divide & Conquer / 3-Way Partition):

Concept:

- Partition the array into three groups based on a pivot or categories.
- Use three pointers to maintain boundaries for less than, equal to, and greater than groups.
- Sort the array by rearranging elements in a single pass.

How it works:

1. Maintain three pointers: low, mid, and high.
2. If current element is less than pivot (or category 0), swap it to the low region and advance low and mid.
3. If current element is equal to pivot (or category 1), just move mid.
4. If current element is greater than pivot (or category 2), swap it with the element at high and decrease high.
5. Repeat until mid passes high.

Time Complexity:

- O(n) — single pass through the array.
- Each element is visited once.

Pros:

- Very efficient for sorting arrays with three distinct categories.
- Linear time complexity.
- In-place sorting, no extra memory needed.

Cons:

- Slightly more complex logic than simple bubble sort.
- Specifically designed for three categories, not a general sorting algorithm.

Feature	Brute Force	Divide and Conquer
Strategy	Repeated pairwise swapping	Singlepass 3-way partion
Time Complexity	$O(n^2)$	O(n)
Efficiency	Low	High
Ideal for	Small or nearly sorted arrays	Array(3 distinct categories)
Number of passes	Multiple	One
Logic Complexity	Simple	Moderate

5.0. RESULTS:

Step 1: Start with an unsorted array of fruits

Example:

[2, 0, 2, 1, 1, 0]

where:

- 0 = Apple
- 1 = Orange
- 2 = Banana

Step 2: Initialize pointers

- **low = 0 (start of array)**

- mid = 0 (current index to check)
- high = last index (end of array)

Step 3: Partition fruits into 3 groups while mid <= high

- If fruit at mid is Apple (0):
  Swap with element at low
  Increment both low and mid
- If fruit at mid is Orange (1):
  Just move mid forward
- If fruit at mid is Banana (2):
  Swap with element at high
  Decrement high
- Do not increment mid here because swapped element needs checking

Step 4: Continue until mid passes high

At this point, fruits are sorted as:

- Apples (0) from start to low-1
- Oranges (1) from low to high
- Bananas (2) from high+1 to end

Example: Sort [2, 0, 2, 1, 1, 0]

Step	low	mid	high	Array State	Action
Start	0	0	5	[2, 0, 2, 1, 1, 0]	mid=0 → Banana(2) swap with high(5)
1	0	0	4	[0, 0, 2, 1, 1, 2]	mid=0 → Apple(0) swap with low(0)

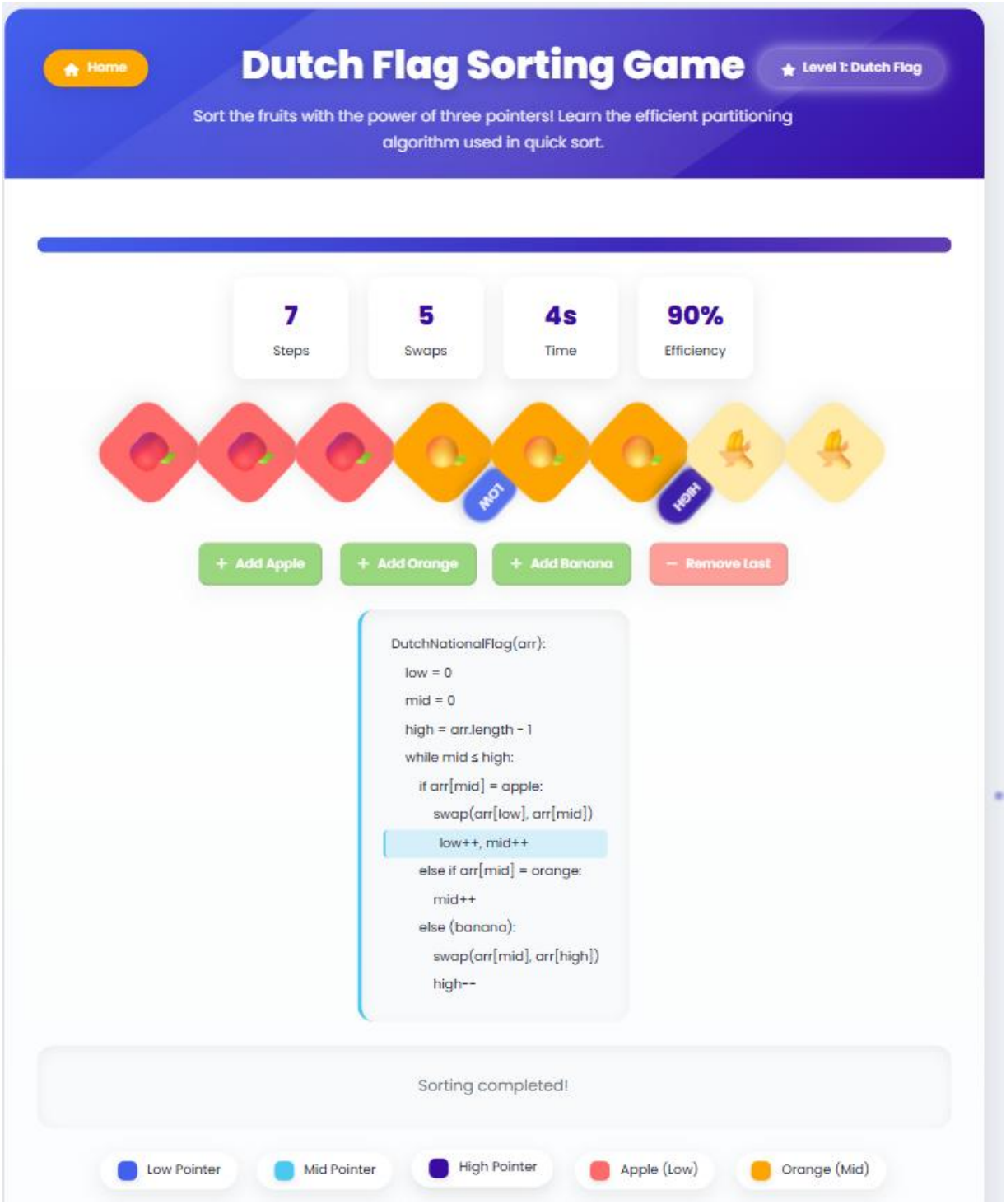
Step	low	mid	high	Array State	Action
2	1	1		[0, 0, 2, 1, 1, 2]	mid=1 → Apple(0) swap with low(1)
3	2	2	4	[0, 0, 2, 1, 1, 2]	mid=2 → Banana(2) swap with high(4)
4	2	2		[0, 0, 1, 1, 2, 2]	mid=2 → Orange(1) move mid forward
5	2	3	3	[0, 0, 1, 1, 2, 2]	mid=3 → Orange(1) move mid forward
End	2	4	3	[0, 0, 1, 1, 2, 2]	mid > high → done

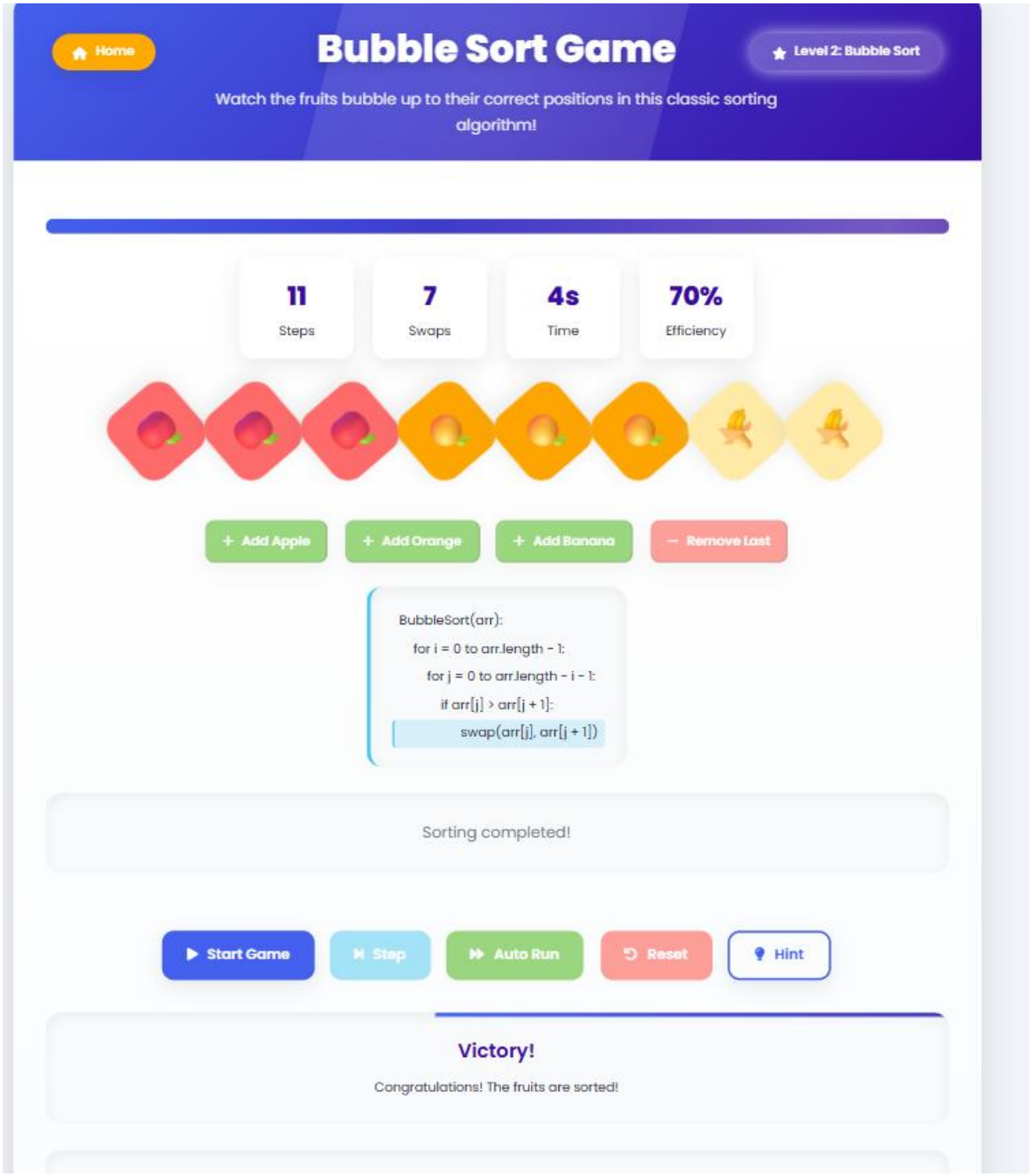
Final Output:

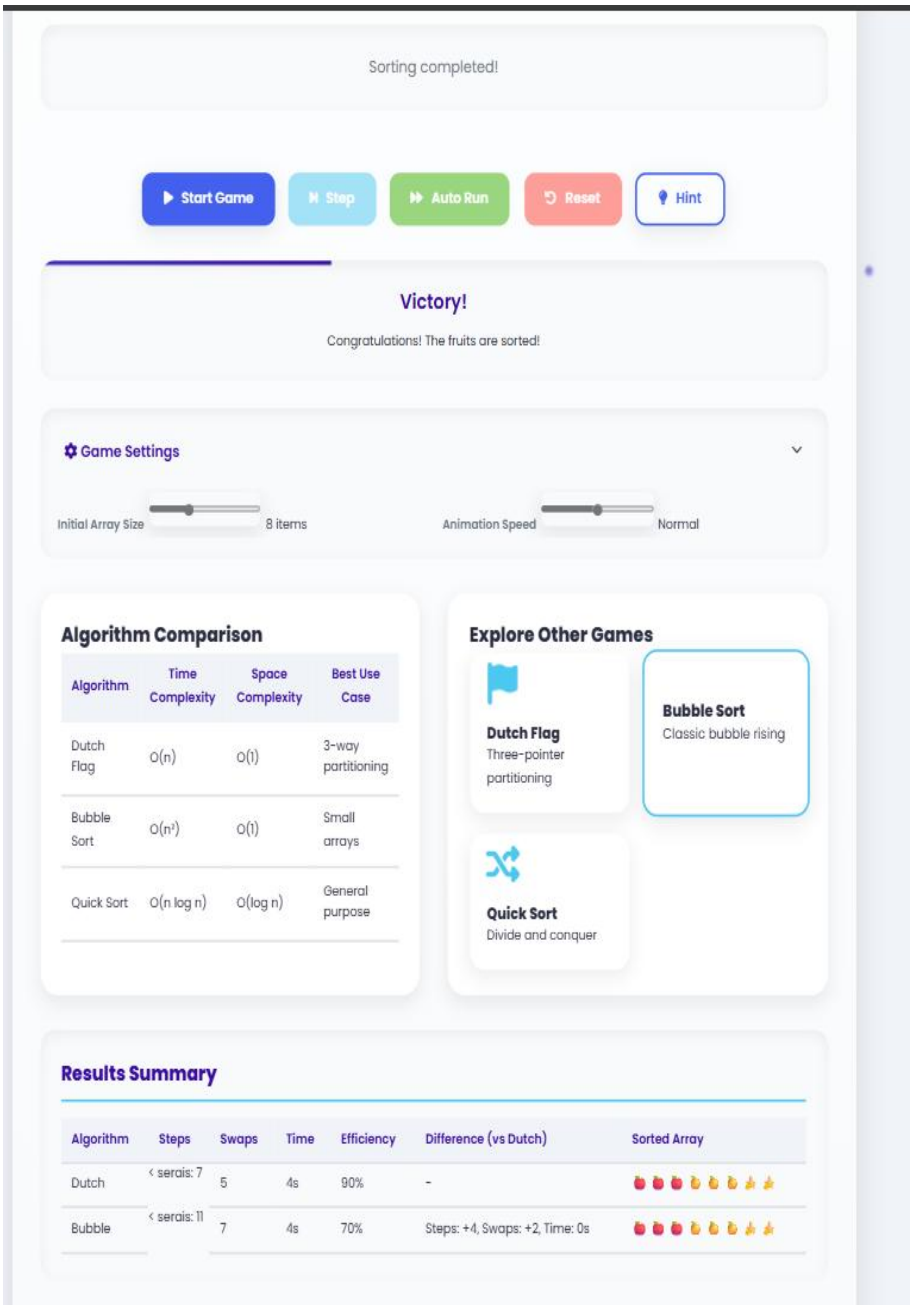
Sorted fruits → [0, 0, 1, 1, 2, 2]

(Apples, Oranges, Bananas)

OUTPUT:


Home
Dutch Flag Sorting Game
Level 1: Dutch Flag
Sort the fruits with the power of three pointers! Learn the efficient partitioning algorithm used in quick sort.
7
Steps
5
Swaps
4s
Time
90%
Efficiency
MID
HIGH
+ Add Apple
+ Add Orange
+ Add Banana
− Remove Last
DutchNationalFlag(arr):
low = 0
mid = 0
high = arr.length - 1
while mid ≤ high:
if arr[mid] = apple:
swap(arr[low], arr[mid])
low++, mid++
else if arr[mid] = orange:
mid++
else (banana):
swap(arr[mid], arr[high])
high--
Sorting completed!
Low Pointer
Mid Pointer
High Pointer
Apple (Low)
Orange (Mid)


Home
Bubble Sort Game
Level 2: Bubble Sort
Watch the fruits bubble up to their correct positions in this classic sorting algorithm!
11
Steps
7
Swaps
4s
Time
70%
Efficiency
+ Add Apple
+ Add Orange
+ Add Banana
− Remove Last
BubbleSort(arr):
for i = 0 to arr.length - 1:
for j = 0 to arr.length - i - 1:
if arr[j] > arr[j + 1]:
swap(arr[j], arr[j + 1])
Sorting completed!
Start Game
Step
Auto Run
Reset
Hint
Victory!
Congratulations! The fruits are sorted!



GITHUB LINK: https://github.com/varchu-sri/DAA