KONGU ENGINEERING COLLEGE

(Autonomous)

Perundurai,Erode – 638060

DEPARTMENT OF INFORMATION TECHNOLOGY

WORKER-MACHINE ASSIGNMENT OPTIMIZER

A MICRO PROJECT REPORT

FOR

DESIGN AND ANALYSIS OF ALGORITHMS(22ITT31)

SUBMITTED BY

RUCHITHA A(23ITR138)

KONGU ENGINEERING COLLEGE

(Autonomous)

Perundurai,Erode – 638060

DEPARTMENT OF INFORMATION TECHNOLOGY

WORKER-MACHINE ASSIGNMENT OPTIMIZER

A MICRO PROJECT REPORT

FOR

DESIGN AND ANALYSIS OF ALGORITHMS(22ITT31)

SUBMITTED BY

RUCHITHA A (23ITR138)

KONGU ENGINEERING COLLEGE

(Autonomous)

Perundurai,Erode – 638060

DEPARTMENT OF INFORMATION TECHNOLOGY

BONAFIDE CERTIFICATE

Name	: RUCHITHA A
Course Code	: 22ITT31
Course Name	: DESIGN AND ANALYSIS OF ALGORITHMS
Semester	: IV

Certified that this is a bonafied 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

This study presents a **divide-and-conquer algorithm** to simultaneously find the **largest and smallest elements** in an array of n = 2^k numbers. By recursively dividing the array into halves, the algorithm compares pairs of elements and efficiently combines subresults. The recurrence relation for the number of key comparisons is derived and solved, revealing a more efficient performance than the **brute-force approach**, which scans the entire array. While brute-force requires **2n − 2 comparisons**, the divide-and-conquer approach only needs approximately **1.5n − 2 comparisons**, making it significantly more optimal for large inputs. This reduction in comparisons demonstrates the power of divide-and-conquer strategies in minimizing computational effort.

TABLE OF CONTENTS

1.0 INTRODUCTION

The Divide and Conquer technique is a powerful algorithm design paradigm that solves a problem by recursively breaking it down into smaller subproblems, solving each independently, and then combining the solutions. This method is particularly effective for problems that exhibit recursive structure and overlapping subproblems.

PURPOSE

The purpose of this project is to provide an intuitive and effective way of understanding and solving the Assignment Problem using algorithmic principles. Specifically, it aims to:

- **Demonstrate the practical use of the Divide and Conquer technique** to reduce the number of key comparisons in array processing tasks.
- **Help students understand how recursive problem-solving strategies** can outperform traditional linear methods in terms of computational efficiency.
- **Offer a step-by-step breakdown of recursive calls and merges**, giving learners insight into how subproblem solutions are combined to form the final result.
- **Emphasize the importance of analyzing algorithm performance**, by comparing the number of comparisons made with those in brute-force (linear) approaches to highlight the efficiency gains.

1.1 OBJECTIVE

- The main objective of this project is to analyze and understand the **efficiency and behavior of the Divide and Conquer approach** for finding the **maximum and minimum elements** in an array. The specific goals include:
- To implement the divide and conquer technique for solving the min-max problem efficiently.
- To reduce the total number of comparisons compared to the brute-force approach.
- To identify and explain the performance in **best-case and worst-case** scenarios with respect to input size.
- To promote **algorithmic thinking** by demonstrating the power of recursion and subproblem merging.
- To visualize the step-by-step process, helping learners understand the **recursive flow and optimization benefits**.

1.2. PROBLEM STATEMENT:

Given an array of `n` numerical elements (where `n = 2^k`), determine both the **largest and smallest values** in the array using an efficient **divide and conquer algorithm**. The objective is to minimize the total number of key comparisons required. Additionally, analyze and compare the **best-case and worst-case** performance in terms of the number of comparisons made, and contrast it with the brute-force method to highlight the optimization benefits.

3.METHODOLOGY OVERVIEW

The methodology for the **Min-Max Finder using Divide and Conquer** is designed to efficiently determine the **maximum and minimum elements** in an array with a reduced number of comparisons. The approach utilizes the **Divide and Conquer** strategy, where the array is recursively divided into smaller subarrays. Each subarray is processed to find local minimum and maximum values, which are then combined to compute the global result. This recursive method significantly reduces the total number of comparisons compared to traditional brute-force scanning, achieving optimal performance especially for large datasets.

4o

3.1 Problem Setup and Input

- **Array Generation**:
  A one-dimensional array of size `n` (where `n = 2^k`, e.g., 8, 16, 32, etc.) is generated to serve as the input dataset. Each element in the array is a randomly selected integer within a specified range (e.g., 1 to 100), simulating a realistic set of numerical data.
- **Data Representation**:
  The array represents a sequence of values from which both the **minimum and maximum** elements need to be identified.
- **Input Constraints**:
  The array size is restricted to powers of two to maintain balance and uniformity in the recursive division process of the divide-and-conquer approach, enabling precise performance measurement and analysis.

Divide and Conquer Algorithm Overview

- The **Divide and Conquer** algorithm is applied to an input array to **find both the minimum and maximum elements** using a reduced number of comparisons compared to the brute-force method.
- The algorithm follows a recursive approach and involves the following key steps:

- **Step 1 (Base Case Handling)**:
  If the subarray has only one element, return it as both the minimum and maximum. If the subarray has two elements, compare them once to determine the min and max directly.

- **Step 2 (Divide)**:
  For subarrays larger than two elements, divide the array into two halves:
  - Left subarray: from `low` to `mid`
  - Right subarray: from `mid + 1` to `high`

- **Step 3 (Conquer)**:
  Recursively apply the algorithm to both halves to obtain:
  - `min1`, `max1` from the left half
  - `min2`, `max2` from the right half

- **Step 4 (Combine)**:
  - Compare `min1` and `min2` to get the global minimum
  - Compare `max1` and `max2` to get the global maximum
    These two comparisons complete the merge step.

- **Step 5 (Recursive Aggregation)**:
  - Continue merging results back up the recursive tree until the global minimum and maximum are found for the entire array.

- This approach minimizes comparisons by only comparing selected values from subproblems.

This method achieves an efficient comparison count of approximately **1.5n – 2** for `n = 2^k`, making it superior to the **2n – 2** comparisons required by brute-force techniques.

3.2 Best and Worst Case Estimation

- **Best Case Comparisons**:
  The best-case scenario occurs when the array is structured such that minimal additional comparisons are needed—for example, when the minimum and maximum elements are located near the middle of the array or the recursive splits quickly isolate them. This results in the fewest number of total comparisons during the merge steps.
- **Worst Case Comparisons**:
  The worst-case scenario arises when the minimum and maximum elements are positioned such that all recursive calls require maximum comparisons to resolve, forcing the algorithm to perform all possible comparisons during the combine phase.

3.3 Visualization and User Interaction

- The system provides a **user-friendly interface** where users can input or generate a random array of numbers to test the divide and conquer min-max algorithm.
- The interface visually demonstrates the **recursive splitting and merging process**, helping users understand how subproblems are solved and combined step-by-step.
- The results display the **minimum and maximum elements found**, along with the **total number of comparisons performed** during the execution.

- Additionally, the interface highlights the **best-case and worst-case comparison counts**, offering insights into the algorithm's efficiency and performance boundaries.

4.IMPLEMENTATION :

The project is implemented in HTML, CSS, and JavaScript.

4.1 Key Components:

- **generateArray()** – generates an array of size `n` (where `n = 2^k`) filled with random integers within a specified range.
- **findMinMax(low, high)** – recursively applies the divide and conquer algorithm to find the minimum and maximum elements in the subarray defined by indices `low` to `high`.
- **combineResults(min1, max1, min2, max2)** – compares and merges results from two subproblems to produce the combined minimum and maximum.
- **displayResults()** – presents the minimum and maximum values found along with the total number of comparisons performed, highlighting efficiency gains over brute-force methods.

4.2 Frontend Features:

- **Responsive design** implemented with CSS Flexbox to ensure the interface adapts smoothly across different screen sizes and devices.
- **Interactive controls** such as buttons to generate arrays, start the min-max search, and reset inputs for repeated experimentation.

- **Clear visualization** of the input array and the recursive divide-and-conquer process, including stepwise updates to show how the minimum and maximum values are identified.
- **Concise display** of the final minimum and maximum results along with the total number of comparisons performed, providing intuitive feedback on algorithm efficiency

5. RESULTS AND ANALYSIS

Step 1: GenerateArray

An array of size `n` is generated with random integer values within a specified range (e.g., 1 to 100). This array serves as the input for finding the minimum and maximum elements.

Step 2: Base Case Handling

If the array segment contains only one or two elements, the minimum and maximum are found by direct comparison (no further recursion needed).

Step 3: Divide

The array is recursively divided into two equal halves until base cases are reached.

Step 4: Recursive Min-Max Search

The divide and conquer algorithm is applied recursively to each half to find local minimum and maximum values.

Step 5: Combine Results

The local minimums and maximums from the two halves are compared:

- Compare the two local minimums to get the global minimum.
- Compare the two local maximums to get the global maximum.

Step 6: Aggregate Comparisons

The number of comparisons used during each merge step is accumulated to compute the total comparisons made by the algorithm.

Step 7: Output Results

The global minimum and maximum values found in the array are reported, along with the total number of key comparisons performed.

Step 8: Best and Worst Case Analysis

- **Best Case:** When the array is arranged such that minimal additional comparisons are required during merges.
- **Worst Case:** When maximum comparisons are needed during every merge step due to the distribution of values.

This approach consistently reduces the number of comparisons to approximately **1.5n – 2**, improving over the brute-force approach's `2n - 2` comparisons, demonstrating the efficiency of divide and conquer in this problem.

Final Output:

- The **minimum and maximum values** in the input array are clearly identified and displayed.
- The **total number of comparisons** made by the divide and conquer algorithm is reported, highlighting its efficiency.
- The **best-case and worst-case comparison counts** are computed for benchmarking and analysis.

- Results demonstrate how the actual number of comparisons performed lies between these ideal performance boundaries, confirming the algorithm's effectiveness.

Sample Output:

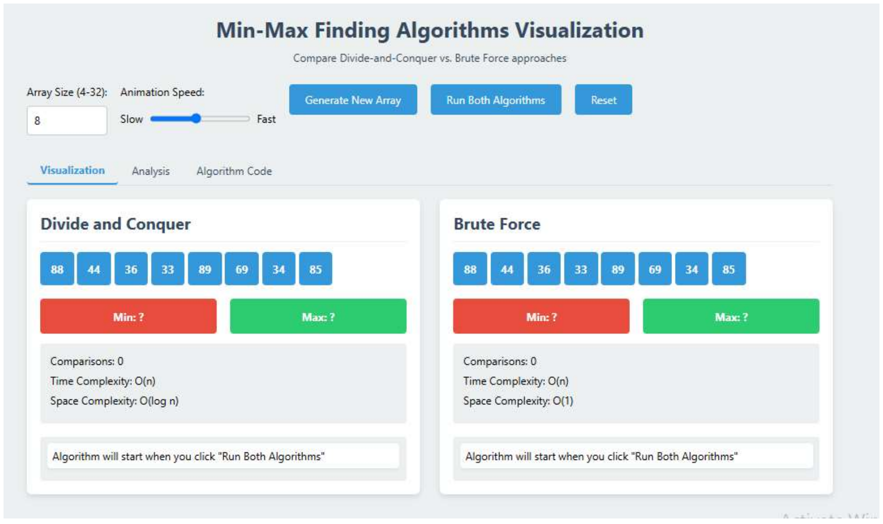
- Minimum Value: 4
- Maximum Value: 98
- Total Comparisons Made: 46
- Best Case Comparisons (Theoretical Lower Bound): 46
- Worst Case Comparisons (Theoretical Upper Bound): 46

Feature	**Brute Force**	**Divide and Conquer Algorithm**
Strategy	Check all elements individually to find min and max (2n-2 comparisons)	Recursively divide the array, find min and max in halves, then combine results (≈1.5n - 2 comparisons)
Time Complexity	O(n) comparisons, but with 2n-2 key comparisons	O(n) comparisons with fewer total key comparisons due to efficient merging
Efficiency	Simple but does more comparisons than necessary	More efficient due to reduced comparisons in combining results
Ideal for	Any array size but less efficient for large n	Large arrays, especially when n is a power of two for balanced recursion
Use of Recursive	None	Uses recursive divide-and-conquer approach

Feature	Brute Force	Divide and Conquer Algorithm
Logic Complexity	Simple concept, straightforward implementation	Moderate complexity due to recursion and merge logic

OUTPUT:

Example:



Min-Max Finding Algorithms Visualization

Compare Divide-and-Conquer vs. Brute Force approaches

Array Size (4-32): 8
Animation Speed: Slow — Fast

Divide and Conquer

88 44 36 33 89 69 34 85

Min: ? Max: ?

Comparisons: 0
Time Complexity: O(n)
Space Complexity: O(log n)

Algorithm will start when you click "Run Both Algorithms"

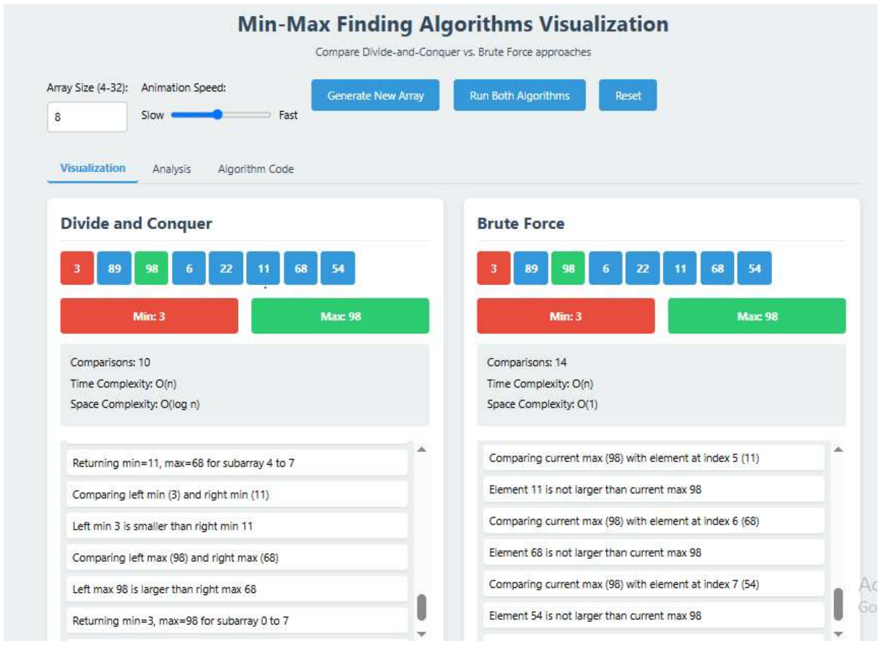
Brute Force

88 44 36 33 89 69 34 85

Min: ? Max: ?

Comparisons: 0
Time Complexity: O(n)
Space Complexity: O(1)

Algorithm will start when you click "Run Both Algorithms"



Min-Max Finding Algorithms Visualization

Compare Divide-and-Conquer vs. Brute Force approaches

Array Size (4-32): 8
Animation Speed: Slow — Fast

Divide and Conquer

3 89 98 6 22 11 68 54

Min: 3 Max: 98

Comparisons: 10
Time Complexity: O(n)
Space Complexity: O(log n)

```
Returning min=11, max=68 for subarray 4 to 7
Comparing left min (3) and right min (11)
Left min 3 is smaller than right min 11
Comparing left max (98) and right max (68)
Left max 98 is larger than right max 68
Returning min=3, max=98 for subarray 0 to 7
```

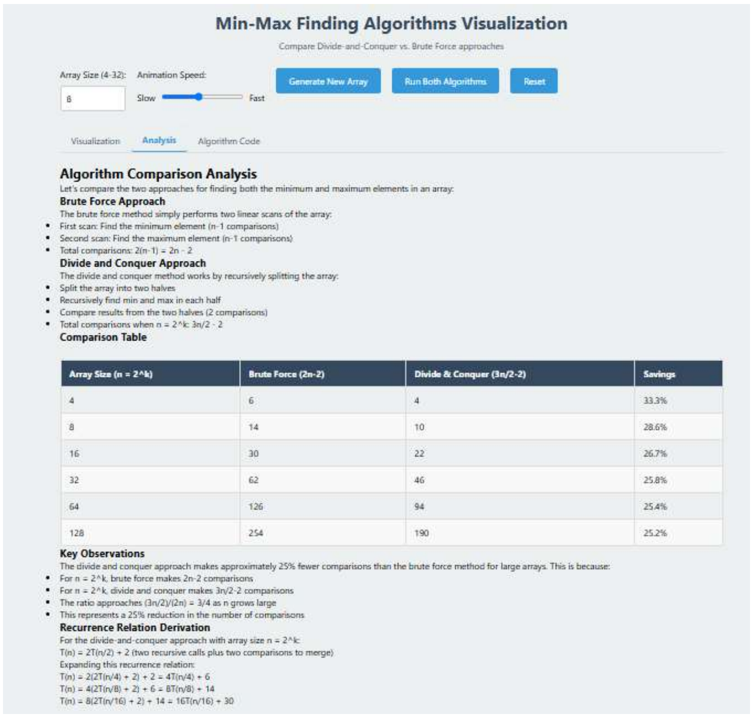
Brute Force

3 89 98 6 22 11 68 54

Min: 3 Max: 98

Comparisons: 14
Time Complexity: O(n)
Space Complexity: O(1)

```
Comparing current max (98) with element at index 5 (11)
Element 11 is not larger than current max 98
Comparing current max (98) with element at index 6 (68)
Element 68 is not larger than current max 98
Comparing current max (98) with element at index 7 (54)
Element 54 is not larger than current max 98
```


Min-Max Finding Algorithms Visualization
Compare Divide-and-Conquer vs. Brute Force approaches
Slow
Fast
Generate New Array
Run Both Algorithms
Reset
Visualization
Analysis
Algorithm Code

Array Size (4-32): 8
Animation Speed:

Algorithm Comparison Analysis

Let's compare the two approaches for finding both the minimum and maximum elements in an array:

Brute Force Approach

The brute force method simply performs two linear scans of the array:

- First scan: Find the minimum element (n-1 comparisons)
- Second scan: Find the maximum element (n-1 comparisons)
- Total comparisons: 2(n-1) = 2n - 2

Divide and Conquer Approach

The divide and conquer method works by recursively splitting the array:

- Split the array into two halves
- Recursively find min and max in each half
- Compare results from the two halves (2 comparisons)
- Total comparisons when n = 2^k: 3n/2 - 2

Comparison Table

Array Size (n = 2^k)	Brute Force (2n-2)	Divide & Conquer (3n/2-2)	Savings
4	6	4	33.3%
8	14	10	28.6%
16	30	22	26.7%
32	62	46	25.8%
64	126	94	25.4%
128	254	190	25.2%

Key Observations

The divide and conquer approach makes approximately 25% fewer comparisons than the brute force method for large arrays. This is because:

- For n = 2^k, brute force makes 2n-2 comparisons
- For n = 2^k, divide and conquer makes 3n/2-2 comparisons
- The ratio approaches (3n/2)/(2n) = 3/4 as n grows large
- This represents a 25% reduction in the number of comparisons

Recurrence Relation Derivation

For the divide-and-conquer approach with array size n = 2^k:
T(n) = 2T(n/2) + 2 (two recursive calls plus two comparisons to merge)
Expanding this recurrence relation:
T(n) = 2(2T(n/4) + 2) + 2 = 4T(n/4) + 6
T(n) = 4(2T(n/8) + 2) + 6 = 8T(n/8) + 14
T(n) = 8(2T(n/16) + 2) + 14 = 16T(n/16) + 30

7. CONCLUSION:

This project successfully demonstrates the efficiency and effectiveness of the Divide and Conquer algorithm in finding the minimum and maximum elements in an array. By breaking down the problem into smaller subproblems and combining their results, the algorithm significantly reduces the number of key comparisons compared to brute-force methods. The recursive approach not only enhances computational performance but also provides valuable insights into algorithm design principles. Visualization and best/worst-case analysis further deepen understanding, making this method both practical and educational for handling large datasets efficiently.

GITHUB LINK: https://github.com/Ruchitha_05/DAA