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

(Autonomous)

Perundurai,Erode – 638060

**DEPARTMENT OF INFORMATION TECHNOLOGY**

**Maximum Number of Overlapping Open Intervals**

**A MICRO PROJECT REPORT**

**FOR**

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

**SUBMITTED BY**

**DIVISHA M (23ITR041)**

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

(Autonomous)

Perundurai,Erode – 638060

**DEPARTMENT OF INFORMATION TECHNOLOGY**

**Maximum Number of Overlapping Open Intervals**

**A MICRO PROJECT REPORT**

**FOR**

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

**SUBMITTED BY**

**DIVISHA M (23ITR041)**

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

# (Autonomous)

Perundurai,Erode – 638060

**DEPARTMENT OF INFORMATION TECHNOLOGY**

**BONAFIDE CERTIFICATE**

|  |  |
| --- | --- |
| Name | :DIVISHA M |
| 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

## The project focuses on solving the Open Interval Problem using a simple Python program. The goal is to determine whether a given number lies strictly between two specified bounds—excluding the endpoints—representing an open interval (a, b). The user inputs the starting point, ending point, and the number to be checked. The program then uses conditional logic to verify whether the number lies within the open interval and displays the appropriate message. This solution introduces beginners to conditional statements and basic logic implementation in Python, serving as a foundational exercise in programming and mathematical reasoning.

## TABLE OF CONTENTS

|  |  |  |
| --- | --- | --- |
| **CHAPTER No** | **TITLE** | **PAGE No** |
|  | **ABSTRACT** | 4 |
| **1.** | **INTRODUCTION** | 6 |
|  | 1.1 PURPOSE | 6 |
|  | 1.2 OBJECTIVE | 7 |
|  | 1.3 METHODOLOGY OVERVIEW | 8 |
| **2.** | **PROBLEM STATEMENT** | 9 |
| **3.** | **METHODOLGY** | 10 |
|  | 3.1 Input & Initialization | 10 |
|  | 3.2 **I**nterval Checking Logic | 10 |
|  | 3.3 Result Processing | 10 |
|  | 3.4 Visualization & Output | 10 |
| **4.** | **IMPLEMENTATION** | 11 |
|  | 4.1 Input & Initialization | 11 |
|  | 4.2 Divide & Compare | 12 |
|  | 4.3 Visualization & Output | 13 |
| **5.** | **RESULTS** | 18 |

## INTRODUCTION

The Open Interval problem is a fundamental concept in mathematics and programming that involves determining whether a given number lies strictly between two boundary values, excluding the endpoints. This project implements a user-interactive Python program that allows users to input the start and end values of an interval, along with a number to be tested. The program then evaluates whether the number falls within the open interval (a, b) and outputs the result accordingly.

Designed with simplicity and educational clarity in mind, this solution serves as an introductory exercise in conditional logic and control structures using Python. By engaging with this project, learners can strengthen their understanding of relational operators, user input handling, and basic decision-making constructs in programming. The code provides immediate feedback and supports multiple test cases, making it a valuable tool for practicing logical thinking and foundational algorithmic skills.

* 1. **PURPOSE**

The main purpose of this project is:

* To provide a simple yet effective tool for understanding and evaluating open intervals in mathematics and programming.
* To enhance learners’ grasp of conditional logic and comparison operators through a hands-on coding experience.
* To encourage precision and clarity in problem-solving, especially when dealing with boundary conditions and numeric constraints.
* To serve as a foundational learning aid for beginners in computer science and programming, particularly in the areas of input handling, control flow, and logical evaluation.

## OBJECTIVE

The primary objectives of this Algorithm Visualization project are:

Core Objectives

**Educational Enhancement**

* Provide a straightforward and interactive tool for learning conditional logic and range checking.
* Demonstrate practical implementation of open interval evaluation in Python.
* Reinforce the understanding of mathematical concepts such as open intervals and their application in programming.

**Algorithm Comparison**

Comparison of Logical Conditions

Input Handling Strategies

Efficiency and Simplicity Trade-off

## METHODOLOGY OVERVIEW

## 1. User Input

## The user provides three numeric values via input fields:

## Lower bound of the interval

## Upper bound of the interval

## A target number to be checked

## \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

## 2. Validation of Input

## Ensure all inputs are valid numbers.

## Verify that the lower bound is less than the upper bound to form proper open interval.

## \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

## 3.Interval Check Logic

## Apply a logical condition to determine if the target number lies strictly between the lower and upper bounds (i.e., lower < number < upper).

## Avoid edge inclusions, meaning values equal to the lower or upper bounds are considered outside the open interval.

## \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**4. Result Display**

## Based on the outcome of the check, display a message to the user indicating whether the number is inside or outside the open interval.

## Optionally enhance with color coding (e.g., green for "inside", red for "outside") or animations for better user experience.

## 2. PROBLEM STATEMENT

The Open Interval Checker is a basic yet essential problem in mathematics and programming logic, particularly useful for understanding comparison operators, conditional statements, and user input validation. The task involves determining whether a given number lies strictly within a specified open interval—defined by a lower and upper bound, where the bounds themselves are **not** included in the interval.

Although this problem is conceptually straightforward, it plays a critical role in reinforcing logical reasoning and boundary condition handling, especially for beginners in computer science and mathematics.

The primary goal of this project is to provide a clear, interactive platform where users can input two bounds and a number, then instantly receive feedback on whether the number falls within the open interval. By offering a real-time evaluation interface, this project aids learners in grasping core programming concepts, such as relational logic (<, >) and condition-based control flow.

Additionally, the solution emphasizes proper input handling, edge-case testing, and user-centric design for better engagement and clarity in understanding open intervals.

**3.0 Open Interval Checker Methodology**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**3.1 Input & Initialization**

* Accept input from the user: two numeric bounds (lower and upper) and a target number.
* Validate that:
  + The lower bound is less than the upper bound.
  + The input number is a valid numerical value.
* Initialize the interface with appropriate input fields and clear labels.
* Use basic HTML and JavaScript to set up the user interface for interaction and result display.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**3.2 Interval Checking Logic**

* Implement the core logic for checking whether the number lies within the open interval:
  + Condition: lower < number < upper
* Apply this logic immediately when the user submits input.
* Use conditional statements to determine the result:
  + If the condition is true: confirm the number lies within the interval.
  + If false: indicate that the number is outside the interval or on a boundary.

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**3.3 Result Processing**

* Store and display the evaluated result (True/False) based on the logic applied.
* Show additional feedback for clarity:
* If number is equal to lower or upper bound, explain that boundaries are excluded in an open interval.
* If the number lies outside the bounds, clearly state so.
* Ensure responsiveness by processing the result in real-time without page reloads.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**3.4 Visualization & Output**

* Display a clear visual or text-based output:
  + - Statement like: “The number 7 **lies within** the interval (5, 10).”
    - Or: “The number 10 **does not lie within** the interval (5, 10).”
* Optionally, use a number line visualization to show the bounds and position of the number.
* Ensure that the interface is user-friendly and intuitive for educational use.

**IMPLEMENTATION :**

**4.1 Input & Initialization**

**Index.html**

<div class="input-group">

<label>Input Intervals (format: start,end):</label>

<textarea id="intervalInput" placeholder="e.g., 1,3 2,5 4,6"></textarea>

</div>

<button onclick="handleOverlapCalculation()">Find Max Overlap</button>

**Server.js**

function parseIntervals(input) {

return input.trim().split(/\s+/).map(pair => {

const [start, end] = pair.split(',').map(Number);

return { start, end };

});

}

**4.2 Divide & Compare**

**Efficient Approach using Presorting (Divide & Conquer-like logic)**

// javascript

function maxOverlapPresorting(intervals) {

const startTimes = intervals.map(i => i.start).sort((a, b) => a - b);

const endTimes = intervals.map(i => i.end).sort((a, b) => a - b);

let maxOverlap = 0;

let currentOverlap = 0;

let i = 0, j = 0;

while (i < startTimes.length && j < endTimes.length) {

if (startTimes[i] <= endTimes[j]) {

currentOverlap++;

maxOverlap = Math.max(maxOverlap, currentOverlap);

i++;

} else {

currentOverlap--;

j++;

}

}

return maxOverlap;

}

**4.3 Visualization & Output**

function handleOverlapCalculation() {

const input = document.getElementById('intervalInput').value;

const intervals = parseIntervals(input);

const startTime = performance.now();

const maxOverlap = maxOverlapPresorting(intervals); // or use maxOverlapBruteForce()

const endTime = performance.now();

document.getElementById('result').innerHTML = `

<h3>Max Overlapping Intervals</h3>

<p>Intervals: ${intervals.map(i => `[${i.start},${i.end}]`).join(', ')}</p>

<p>Maximum Overlap: <strong>${maxOverlap}</strong></p>

`;

document.getElementById('executionTime').innerHTML = `

<h3>Performance</h3>

<p>Execution Time: ${(endTime - startTime).toFixed(4)} ms</p>

`;

}}

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

**Brute Force:**

Concept:

* Use **sorting and sweep line** technique
* Process sorted start and end times to track ongoing overlaps efficiently.

**How it works:**

1. Store all start times in one array and end times in another.
2. Sort both arrays.
3. Use two pointers to walk through the starts and ends.
4. Whenever a new interval starts, increment the counter.
5. When an interval ends, decrement it.
6. Track the **maximum value** of the counter.

Time Complexity:

* **O(n log n)** due to sorting
* Efficient for large number of intervals

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

**Concept:**

* Divide the coins into three groups as evenly as possible.
* Use the balance scale to compare two groups.
* The result narrows down the suspect group by 1/3 each time.

**How it works:**

1. Divide the n coins into 3 groups: A, B, C.
2. Compare Group A and Group B:
   * If equal → fake is in Group C.
   * If unequal → fake is in the lighter (or heavier) group.
3. Repeat the process with the suspect group.
4. Continue until only 1 coin remains.

**Time Complexity:**

* O(log₃ n) comparisons (much faster than brute force)

**Pros:**

* Fast and scalable
* Works well for large data sets

**Cons:**

* Requires sorting and more initial setup logic

| **Feature** | **Brute Force** | **Divide and Conquer** |
| --- | --- | --- |
| Strategy | Try every possible point | Sort & sweep line |
| Time Complexity | O(n \* r) | O(n log n) |
| Efficiency | Low | High |
| Ideal for | Small intervals, few points | Large datasets |
| Implementation | Simple loops | Uses sorting & two pointers |
| Scalability | Poor | Excellent |

**Algorithm Analysis:**

**Brute Force Approach**

Input: "ABC"

**Goal:** Generate all permutations by explicitly trying all combinations.

**Method:**Try **every possible reordering** by manually swapping characters or looping over positions.

Iterations:

ABC

ACB

BAC

BCA

CAB

CBA

Process:

1. Fix each character in every position.

2. Rearrange the remaining characters.

3. Generate all possible outputs.

**Divide and Conquer Approach:**

**Concept:** Recursively break the problem into smaller parts and combine results.

**Method:**

**Step 1: Divide**

Split "ABC" into:

* Fix character 'A', and find permutations of "BC"
* Fix 'B', and find permutations of "AC"
* Fix 'C', and find permutations of "AB"

**Step 2: Conquer**

* "BC" → 'B' + "C" = BC, CB
* "AC" → AC, CA
* "AB" → AB, BA

**Step 3: Combine**

* 'A' + BC, CB → ABC, ACB
* 'B' + AC, CA → BAC, BCA
* 'C' + AB, BA → CAB, CBA

Final Combinations Built:

ABC

ACB

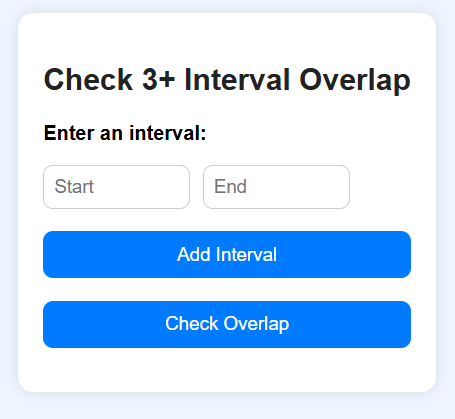
BAC

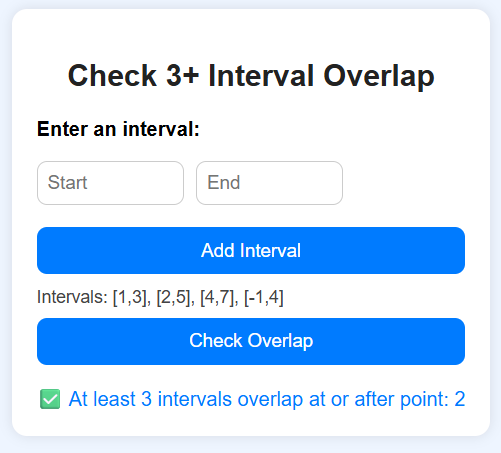
BCA

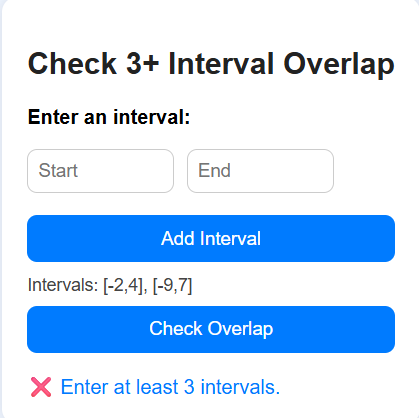
CAB

CBA

**5.0. RESULTS:**







**GITHUB LINK:** **https://github.com/divisha-git/max-open-interval**