



Experiment-4

Student Name: Anshuman Raj

UID: 22BET10081

Branch: BE-IT

Section: 22BET_IOT-702 'A'

Semester: 6th

Date of Performance: 14/02/25

Sub Name: Advanced Programming Lab-2

Subject Code: 22ITP-351

Problem 1

1. Aim:

A string **s** is nice if every letter it contains appears in both uppercase and lowercase. For example, "abABB" is nice, but "abA" is not since 'b' lacks 'B'.

2. Objective:

1. Find the longest nice substring where each letter appears in both uppercase and lowercase.
2. Use recursion to split and check substrings for the longest valid one.
3. Return the longest nice substring or an empty string if none exist.

3. Code:

```
class Solution {
public String longestNiceSubstring(String s) {
    if (s.length() < 2) return "";
    for (int i = 0; i < s.length(); i++) {
        char c = s.charAt(i);
        if (s.contains(Character.toString(Character.toUpperCase(c))) &&
            s.contains(Character.toString(Character.toLowerCase(c)))) {
            continue;
        }
        String left = longestNiceSubstring(s.substring(0, i));
        String right = longestNiceSubstring(s.substring(i + 1));
```

```
return left.length() >= right.length() ? left : right;
}
return s;
}
}
```

4. Output:

Problem List<>RunSubmit

DescriptionNoteEditorialSolutionsSubmissions

1763. Longest Nice Substring

EasyTopicsCompaniesHint

A string `s` is **nice** if, for every letter of the alphabet that `s` contains, it appears **both** in uppercase and lowercase. For example, "abAB" is nice because 'A' and 'a' appear, and 'B' and 'b' appear. However, "abA" is not because 'b' appears, but 'B' does not.

Given a string `s`, return the longest **substring** of `s` that is **nice**. If there are multiple, return the substring of the **earliest** occurrence. If there are none, return an empty string.

Example 1:
Input: `s = "YazaAay"`
Output: `"aAa"`
Explanation: "aAa" is a nice string because 'A/a' is the only letter of the alphabet in s, and both 'A' and 'a' appear.
"aAa" is the longest nice substring.

Example 2:
Input: `s = "Bb"`
Output: `"Bb"`
Explanation: "Bb" is a nice string because both 'B' and 'b' appear. The whole string is a substring.

Example 3:
Input: `s = "c"`
Output: ""

</> Code

JavaAuto

```
9         continue;  
10    }  
11  
12    String left = longestNiceSubstring(s.substring(0, i));  
13    String right = longestNiceSubstring(s.substring(i + 1));  
14  
15    return left.length() >= right.length() ? left : right;  
16  
17 }
```

SavedLn 21, Col 1

Testcase> Test Result

AcceptedRuntime: 0 ms

• Case 1 • Case 2 • Case 3

Input

s =
"YazaAay"

Output

"aAa"

Expected

"aAa"

5. Learning Outcomes:

1. Understand the concept of a nice substring, where each letter appears in both uppercase and lowercase.
2. Learn how to use recursion to solve string-based problems efficiently.
3. Improve problem-solving skills by applying divide and conquer techniques.
4. Gain experience in string manipulation and character checking in Java.



Problem 2

1. Aim:

To develop a program that reverse the bits of a given 32-bit unsigned integer and return the result as an integer.

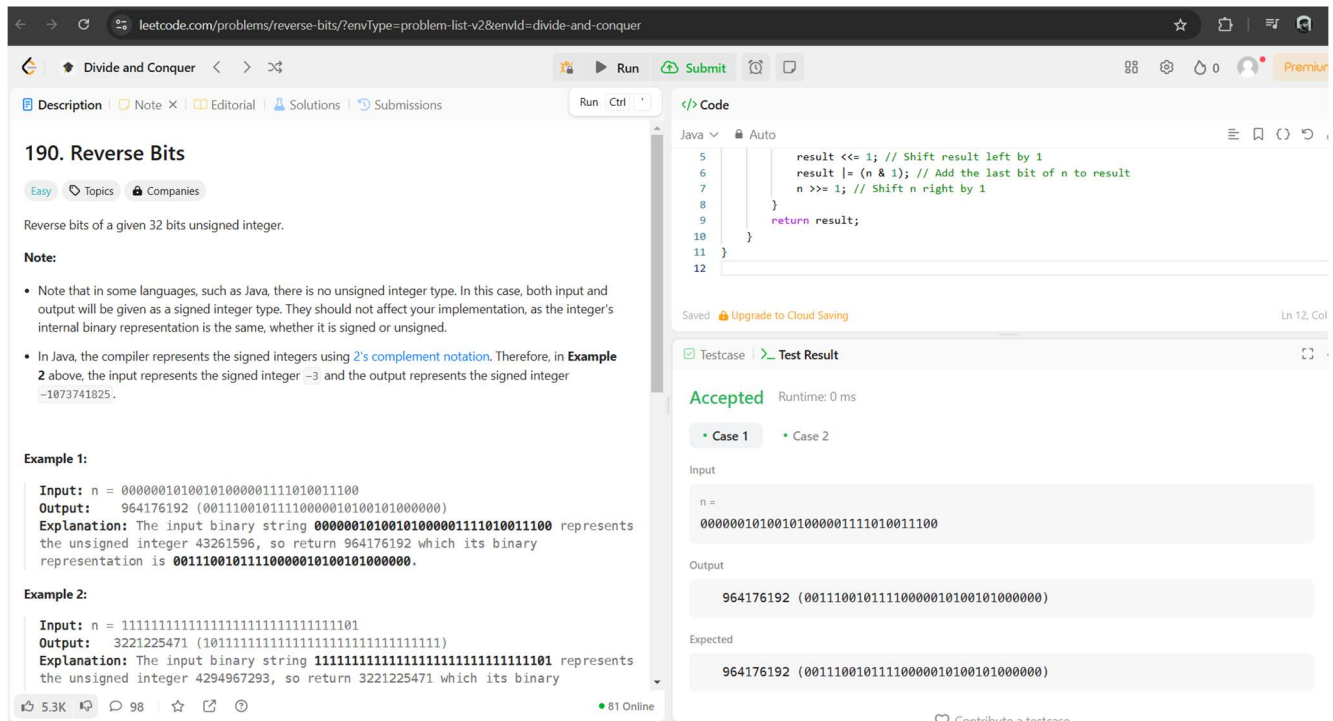
2. Objective:

- Gain proficiency in traversing and manipulating a singly linked list using pointers.
- Understand and implement in-place reversal of a specific sublist within a linked list.

3.Code:

```
public class Solution {  
  
    public int reverseBits(int n) {  
  
        int result = 0;  
  
        for (int i = 0; i < 32; i++) {  
  
            result <<= 1;  
  
            result |= (n & 1);  
  
            n >>= 1;  
  
        }  
  
        return result;  
  
    }  
}
```

4.Output:



190. Reverse Bits

Reverse bits of a given 32 bits unsigned integer.

Note:

- Note that in some languages, such as Java, there is no unsigned integer type. In this case, both input and output will be given as a signed integer type. They should not affect your implementation, as the integer's internal binary representation is the same, whether it is signed or unsigned.
- In Java, the compiler represents the signed integers using 2's complement notation. Therefore, in **Example 2** above, the input represents the signed integer -3 and the output represents the signed integer -1073741825 .

Example 1:

Input: $n = 00000010100101000001111010011100$
Output: 964176192 ($0011100101110000010100101000000$)
Explanation: The input binary string $00000010100101000001111010011100$ represents the unsigned integer 43261596 , so return 964176192 which its binary representation is $0011100101110000010100101000000$.

Example 2:

Input: $n = 11111111111111111111111111111101$
Output: 3221225471 ($10111111111111111111111111111111$)
Explanation: The input binary string $11111111111111111111111111111101$ represents the unsigned integer 4294967293 , so return 3221225471 which its binary representation is $10111111111111111111111111111111$.

```

5  result <<= 1; // Shift result left by 1
6  result |= (n & 1); // Add the last bit of n to result
7  n >>= 1; // Shift n right by 1
8  }
9  return result;
10 }
11 }
12

```

Accepted Runtime: 0 ms

Case 1 Case 2

Input

$n = 00000010100101000001111010011100$

Output

964176192 ($0011100101110000010100101000000$)

Expected

964176192 ($0011100101110000010100101000000$)

5.Learning Outcomes:

1. Understand how to manipulate bits using bitwise operations like shift (\ll , \gg) and bit masking ($\&$).
2. Learn an efficient $O(1)$ approach to reverse bits in a 32-bit integer.
3. Gain experience in loop-based bit manipulation techniques for optimizing low-level operations.
4. Improve problem-solving skills by working with binary representation and bitwise transformations in Java.

Problem 3

1.Aim:

Given a positive integer n , write a function to count and return the number of set bits (1s) in its binary representation, also known as the Hamming weight.

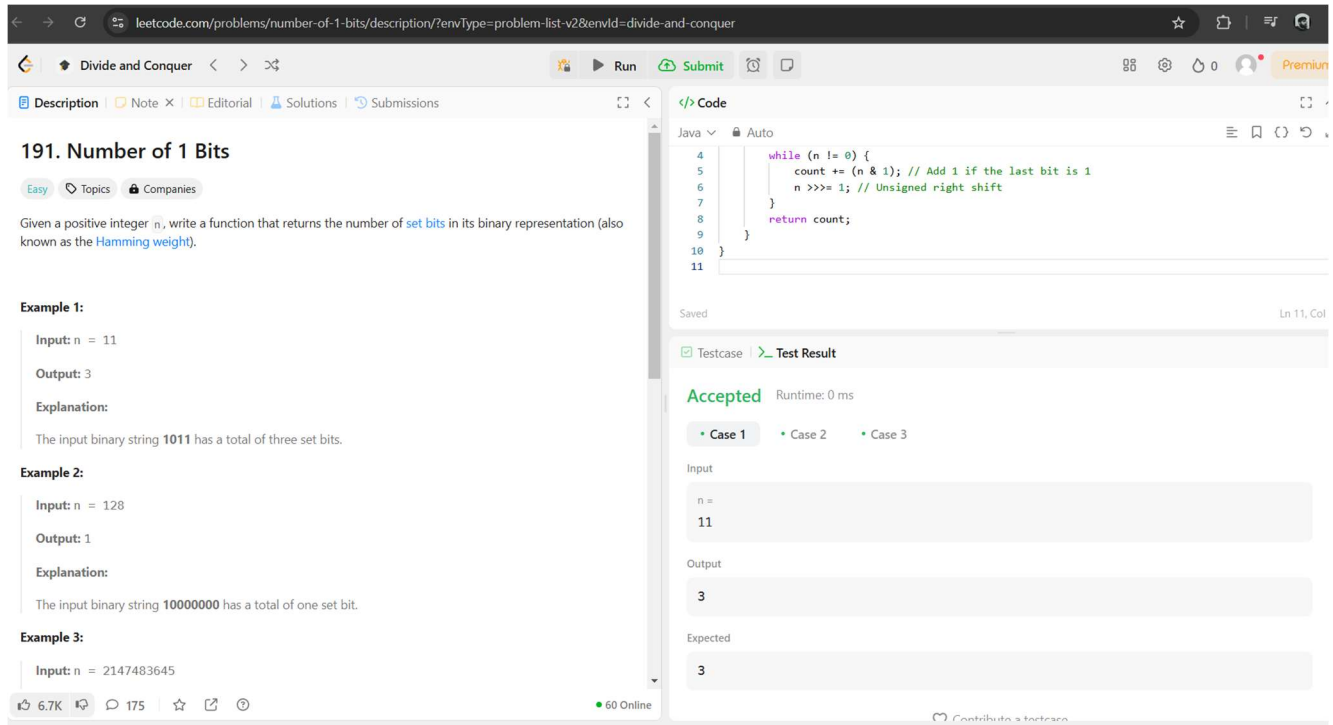
2. Objective:

1. Count the number of set bits (1s) in the binary representation of a given positive integer.
2. Use bitwise operations to efficiently determine the Hamming weight.
3. Implement an optimal algorithm with minimal time complexity.

3.Code:

```
public class Solution {  
  
    public int hammingWeight(int n) {  
  
        int count = 0;  
  
        while (n != 0) {  
  
            count += (n & 1); // Add 1 if the last bit is 1  
  
            n >>= 1; // Unsigned right shift  
  
        }  
  
        return count;  
  
    }  
}
```

4.Output:



191. Number of 1 Bits

Easy Topics Companies

Given a positive integer n , write a function that returns the number of **set bits** in its binary representation (also known as the **Hamming weight**).

Example 1:

Input: $n = 11$

Output: 3

Explanation:

The input binary string 1011 has a total of three set bits.

Example 2:

Input: $n = 128$

Output: 1

Explanation:

The input binary string 10000000 has a total of one set bit.

Example 3:

Input: $n = 2147483645$

6.7K 175 60 Online

```
Java
while (n != 0) {
    count += (n & 1); // Add 1 if the last bit is 1
    n >>= 1; // Unsigned right shift
}
return count;
```

Accepted Runtime: 0 ms

Case 1 Case 2 Case 3

Input: $n = 11$

Output: 3

Expected: 3

5.Learning Outcomes:

1. Understand the concept of Hamming weight and how to count set bits in a binary number.
2. Learn to use bitwise operations like AND (&) and right shift (>>>) for efficient computation.
3. Improve problem-solving skills by implementing an $O(1)$ time complexity approach for a fixed 32-bit integer.
4. Gain hands-on experience with binary representation and low-level bit manipulation in Java.



Problem 4

1.Aim:

Given an integer array nums, find the contiguous subarray with the maximum sum and return that sum.

2.Objective:

1. Efficiently determine the contiguous subarray with the maximum sum from a given array of integers.
2. Implement Kadane's Algorithm to solve the problem in $O(n)$ time complexity using a greedy approach.
3. Explore the Divide and Conquer approach to understand its recursive structure and $O(n \log n)$ complexity.
4. Compare both approaches in terms of performance, scalability, and real-world applicability.

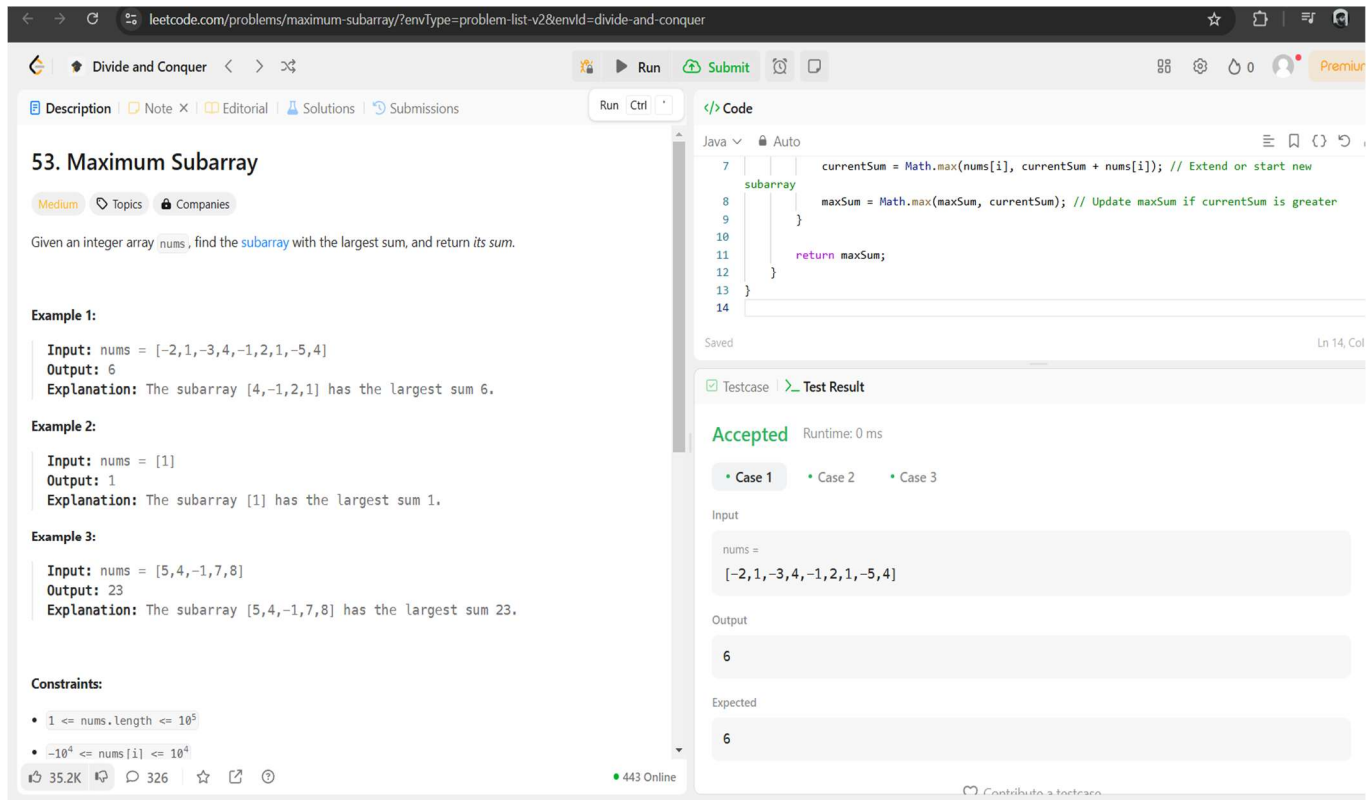
3.Code:

```
class Solution {
public int maxSubArray(int[] nums) {
    int maxSum = nums[0]; // Initialize max sum as the first element
    int currentSum = nums[0]; // Initialize current sum as the first element

    for (int i = 1; i < nums.length; i++) {
        currentSum = Math.max(nums[i], currentSum + nums[i]); // Extend or start new subarray
        maxSum = Math.max(maxSum, currentSum); // Update maxSum if currentSum is greater
    }

    return maxSum;
}
```

4.Output:



The screenshot shows the LeetCode interface for problem 53, "Maximum Subarray". The problem description states: "Given an integer array `nums`, find the `subarray` with the largest sum, and return `its sum`." It includes three examples: Example 1 with input `nums = [-2,1,-3,4,-1,2,1,-5,4]` and output 6; Example 2 with input `nums = [1]` and output 1; and Example 3 with input `nums = [5,4,-1,7,8]` and output 23. The constraints are: $1 \leq \text{nums.length} \leq 10^5$ and $-10^4 \leq \text{nums}[i] \leq 10^4$. The code editor shows a Java solution using Kadane's Algorithm. The test results show "Accepted" with a runtime of 0 ms for Case 1, where the input is `nums = [-2,1,-3,4,-1,2,1,-5,4]` and the output is 6.

53. Maximum Subarray

Medium Topics Companies

Given an integer array `nums`, find the `subarray` with the largest sum, and return `its sum`.

Example 1:

Input: `nums = [-2,1,-3,4,-1,2,1,-5,4]`
Output: 6
Explanation: The subarray `[4,-1,2,1]` has the largest sum 6.

Example 2:

Input: `nums = [1]`
Output: 1
Explanation: The subarray `[1]` has the largest sum 1.

Example 3:

Input: `nums = [5,4,-1,7,8]`
Output: 23
Explanation: The subarray `[5,4,-1,7,8]` has the largest sum 23.

Constraints:

- $1 \leq \text{nums.length} \leq 10^5$
- $-10^4 \leq \text{nums}[i] \leq 10^4$

35.2K 326 443 Online

```
Java Auto
7   currentSum = Math.max(nums[i], currentSum + nums[i]); // Extend or start new
8   subarray
9   }
10  maxSum = Math.max(maxSum, currentSum); // Update maxSum if currentSum is greater
11  }
12  return maxSum;
13  }
14  }
```

Saved Ln 14, Col

Testcase Test Result

Accepted Runtime: 0 ms

Case 1 Case 2 Case 3

Input

`nums =`
`[-2,1,-3,4,-1,2,1,-5,4]`

Output

6

Expected

6

Contribute a test case

5. Learning Outcomes:

1. Understanding Kadane's Algorithm – Learn how to efficiently find the maximum sum of a contiguous subarray in $O(n)$ time complexity using a greedy approach.
2. Applying Divide and Conquer – Understand how to break the problem into subproblems and solve it recursively in $O(n \log n)$ complexity.
3. Comparing Algorithmic Approaches – Analyze the trade-offs between Kadane's Algorithm (greedy) and the Divide and Conquer method in terms of time complexity and practical use cases.
4. Enhancing Problem-Solving Skills – Develop a deeper understanding of dynamic programming, recursion, and optimization techniques for solving array-based problems.



Problem 5

1.Aim:

Your task is to compute $a^b \bmod 1337$ where a is a positive integer and b is a very large positive integer represented as an array of its digits.

2.Objective:

- Understand and implement efficient sorting techniques for singly linked lists.
- Explore merge sort and quick sort for linked list sorting.
- Analyze the time and space complexity of different sorting approaches.
- Develop skills in manipulating linked lists for in-place sorting.

3.Code:

```
class Solution {
public boolean searchMatrix(int[][] matrix, int target) {
    if (matrix == null || matrix.length == 0 || matrix[0].length == 0) return false;
    return search(matrix, 0, 0, matrix.length - 1, matrix[0].length - 1, target);
}

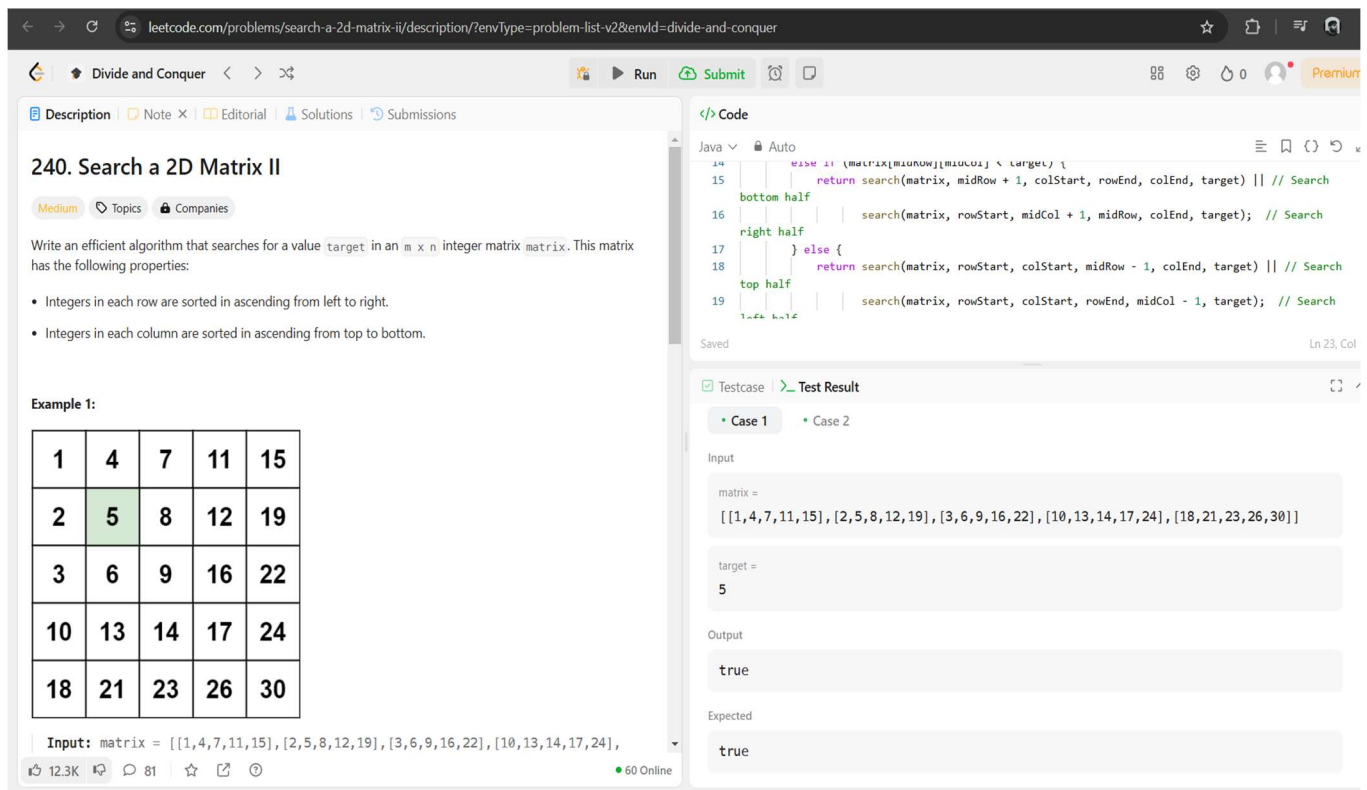
private boolean search(int[][] matrix, int rowStart, int colStart, int rowEnd, int colEnd, int target) {
    if (rowStart > rowEnd || colStart > colEnd) return false;

    int midRow = rowStart + (rowEnd - rowStart) / 2;
    int midCol = colStart + (colEnd - colStart) / 2;

    if (matrix[midRow][midCol] == target) return true;
    else if (matrix[midRow][midCol] < target) {
        return search(matrix, midRow + 1, colStart, rowEnd, colEnd, target) ||
            search(matrix, rowStart, midCol + 1, midRow, colEnd, target);
    } else {
        return search(matrix, rowStart, colStart, midRow - 1, colEnd, target) ||
            search(matrix, rowStart, colStart, rowEnd, midCol - 1, target);
    }
}
```

```
}
}
}
```

4. Output:



240. Search a 2D Matrix II

Medium

Write an efficient algorithm that searches for a value `target` in an `m x n` integer matrix `matrix`. This matrix has the following properties:

- Integers in each row are sorted in ascending from left to right.
- Integers in each column are sorted in ascending from top to bottom.

Example 1:

1	4	7	11	15
2	5	8	12	19
3	6	9	16	22
10	13	14	17	24
18	21	23	26	30

Input: matrix = [[1,4,7,11,15], [2,5,8,12,19], [3,6,9,16,22], [10,13,14,17,24], [18,21,23,26,30]], target = 5

Output: true

Expected: true

5. Learning Outcomes:

1. Understanding Matrix Properties – Learn how sorted 2D matrices enable efficient search strategies beyond brute force.
2. Mastering Optimized Search ($O(m + n)$) – Utilize the top-right or bottom-left approach to efficiently search in a sorted matrix with linear time complexity.
3. Exploring Divide and Conquer ($O(\log m! * \log n!)$) – Understand how recursive partitioning can solve search problems but may not always be the most efficient approach.
4. Comparing Algorithmic Strategies – Analyze the trade-offs between greedy search and recursive approaches in terms of time complexity and practical applications.