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:

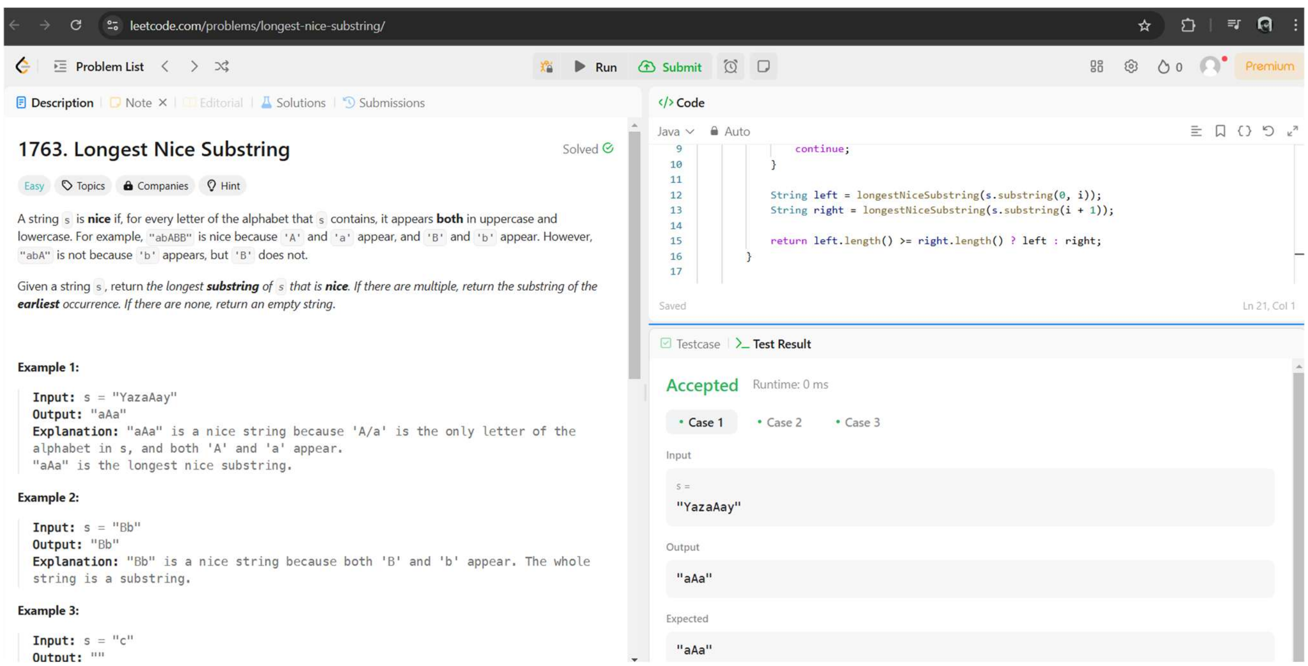
1. Find the longest nice substring where each letter appears in both uppercase and lowercase.
2. Use recursion to split and check substrings for the longest valid one.
3. Return the longest nice substring or an empty string if none exist.

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:



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.

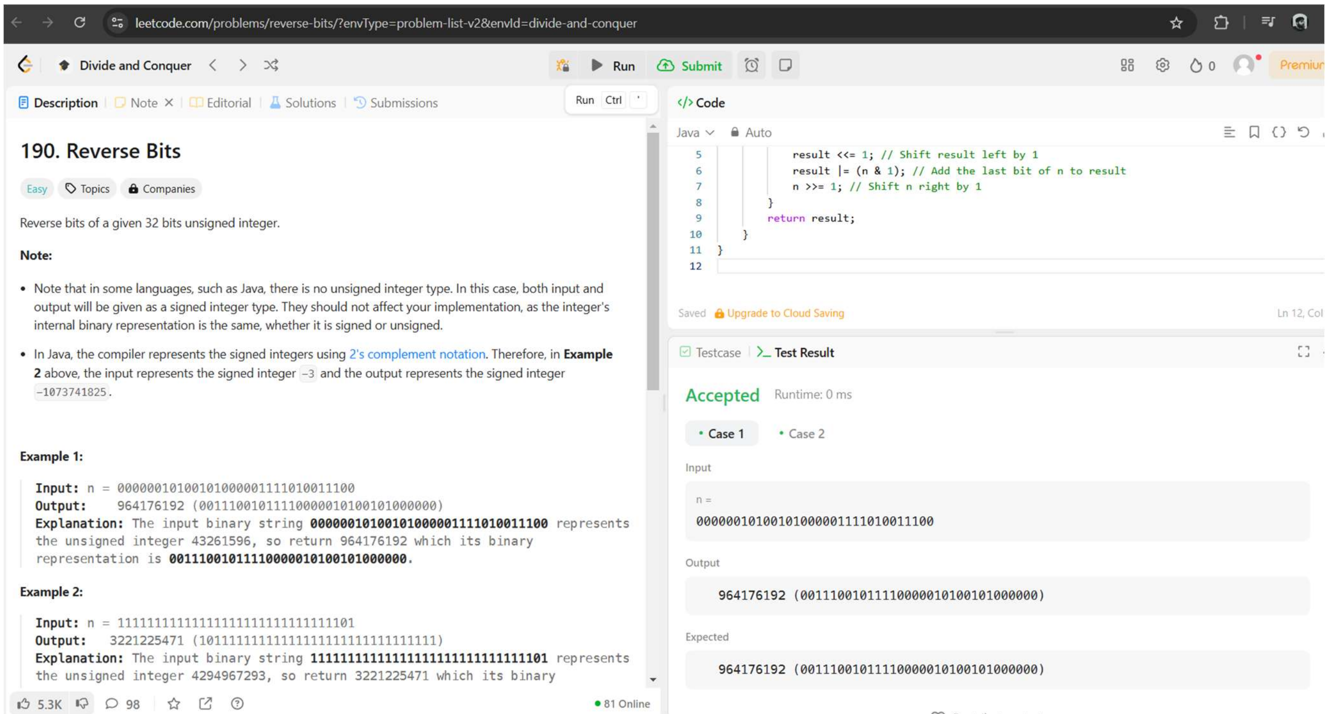
2. Objective:

- 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:



5.Learning Outcomes:

1. Understand how to manipulate bits using bitwise operations like shift (<<, >>) 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.

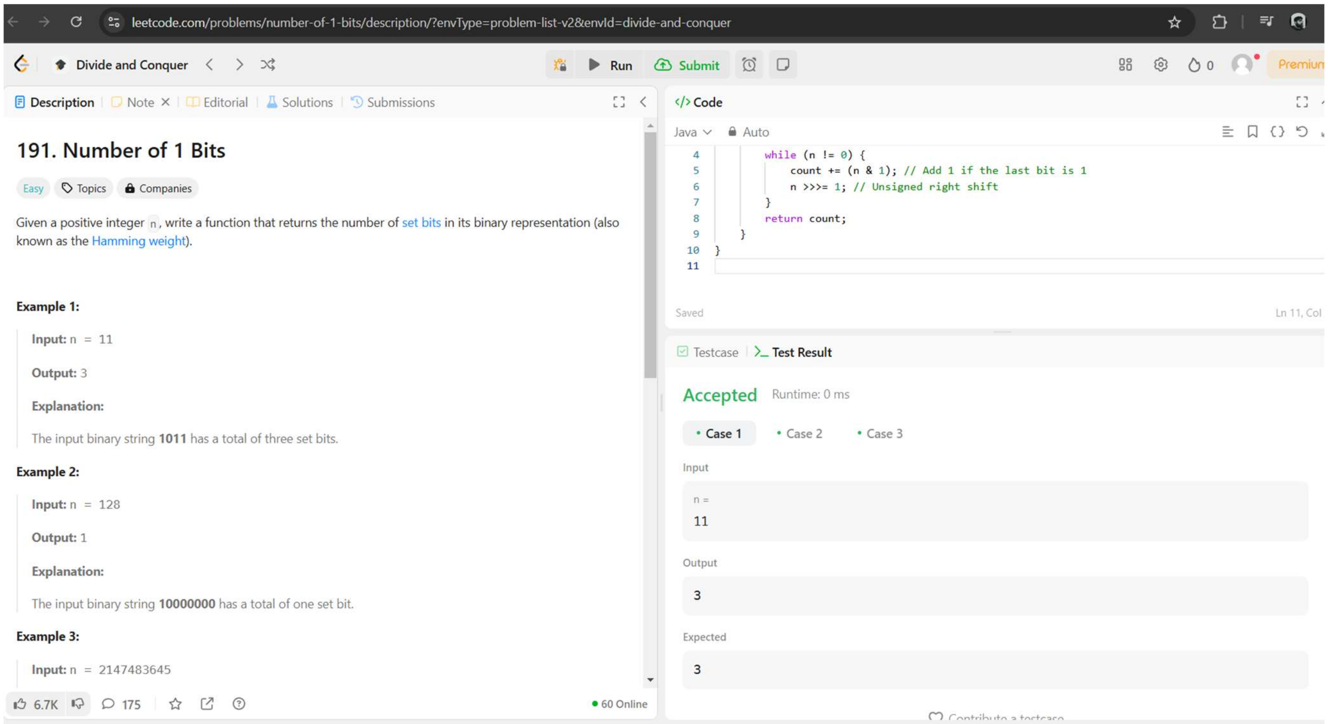
2. Objective:

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:



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.

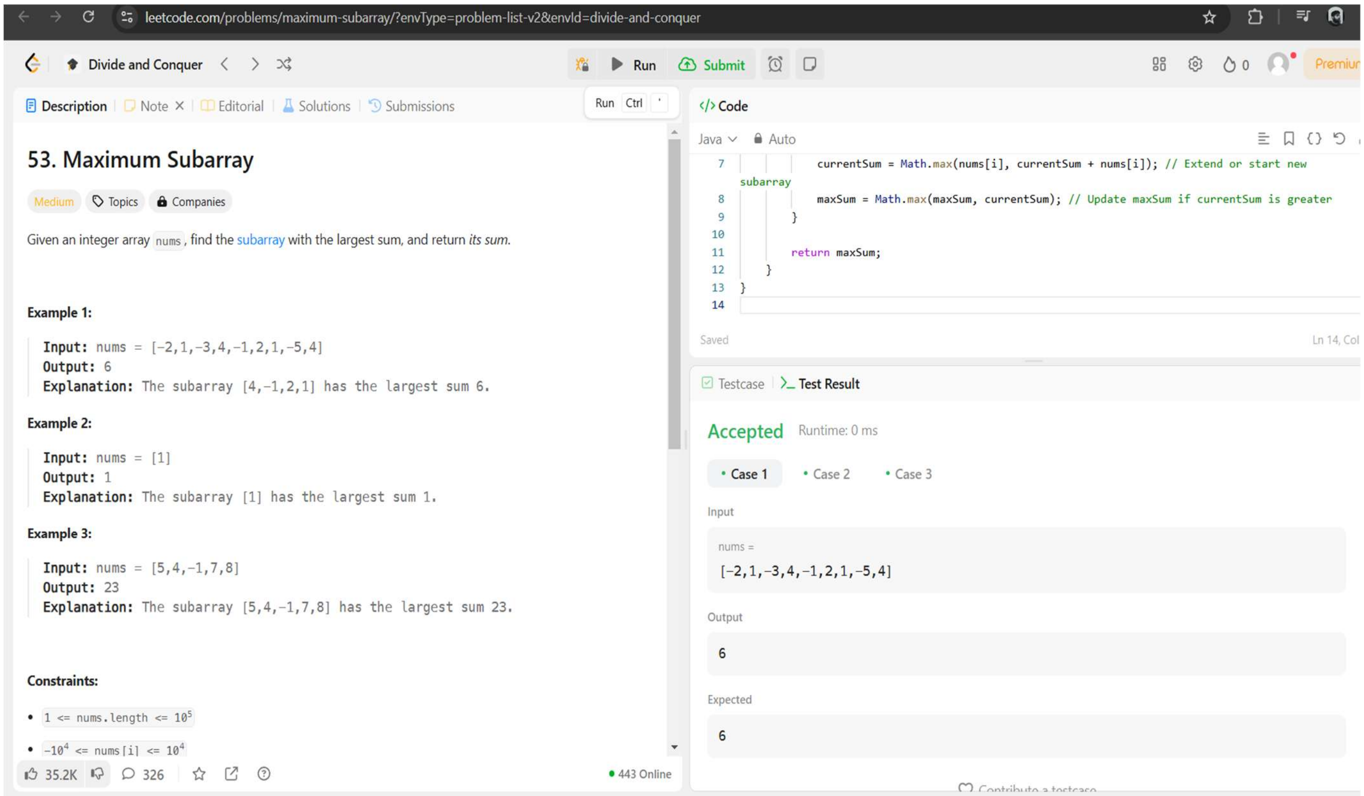
2.Objective:

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:



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 mod 1337 where a is a positive integer and b is a very large positive integer represented as an array of its digits.

2.Objective:

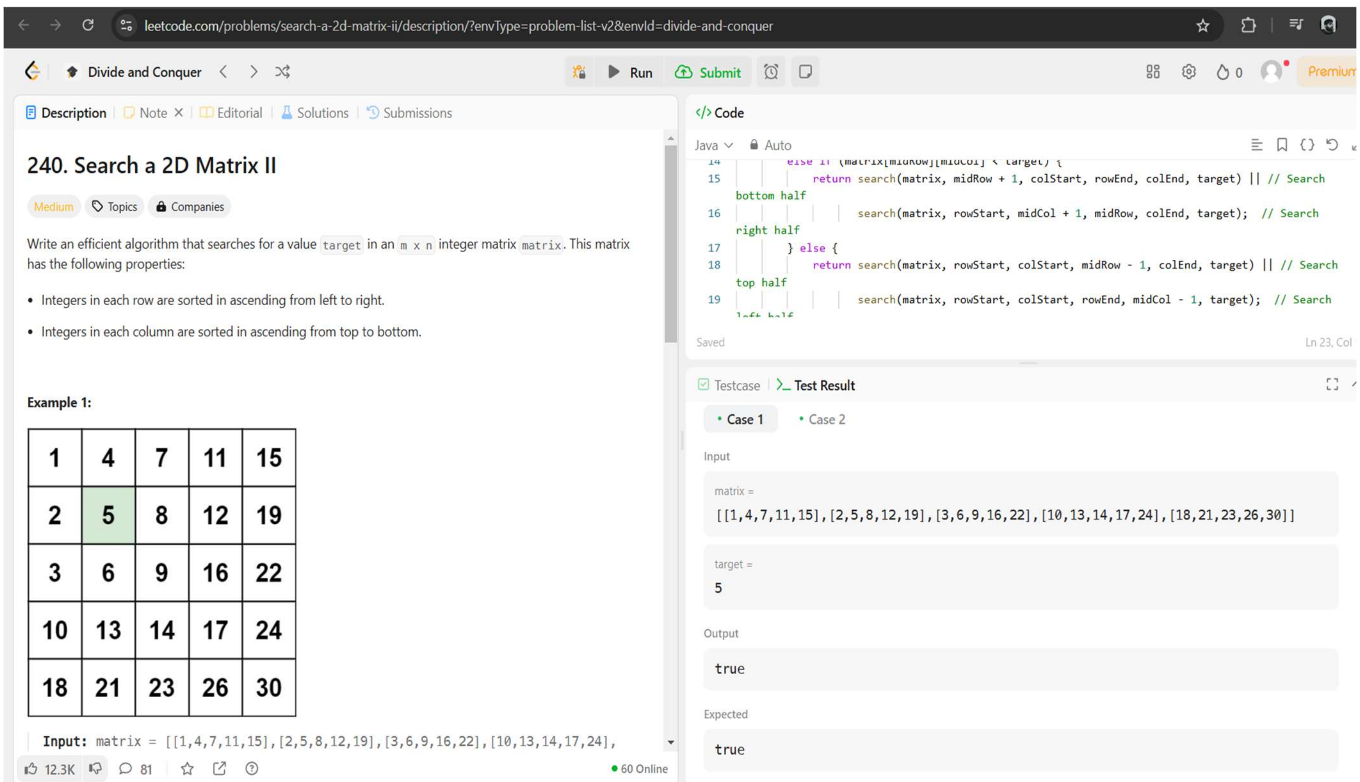
- Understand and implement efficient sorting techniques for singly linked lists.
- Explore merge sort and quick sort for linked list sorting.
- Analyze the time and space complexity of different sorting approaches.
- Develop skills in manipulating linked lists for in-place sorting.

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:



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.