Experiment: - 10

Student Name: Sanjusha Singh **UID:**22BCS13130
Branch: CSE **Section/Group:** 22BCS IOT-614/B
Semester: 6th **Date of Performance:** 17/04/2025
Subject Name: Advanced Programming Lab-2 **Subject Code:** 22CSP-351

Problem -1

1. Aim: Pascal's triangle.

Objective:

- **Algorithm design**: Practice generating the triangle using loops or recursion.
- **Data structure use**: Learn how to use arrays, lists, or matrices to store and manipulate triangle data.
- **Problem solving**: Apply Pascal's Triangle to solve coding problems (e.g., in LeetCode or competitive programming).

2. Implementation/Code:

```
class Solution {
public:
    vector<vector<int>> generate(int numRows) {
        if (numRows == 0) return {};
        if (numRows == 1) return {{1}};

        vector<vector<int>> prevRows = generate(numRows - 1);
        vector<int> newRow(numRows, 1);

        for (int i = 1; i < numRows - 1; i++) {
            newRow[i] = prevRows.back()[i - 1] + prevRows.back()[i];
        }

        prevRows.push_back(newRow);
        return prevRows;
    }
};
```

4. Output

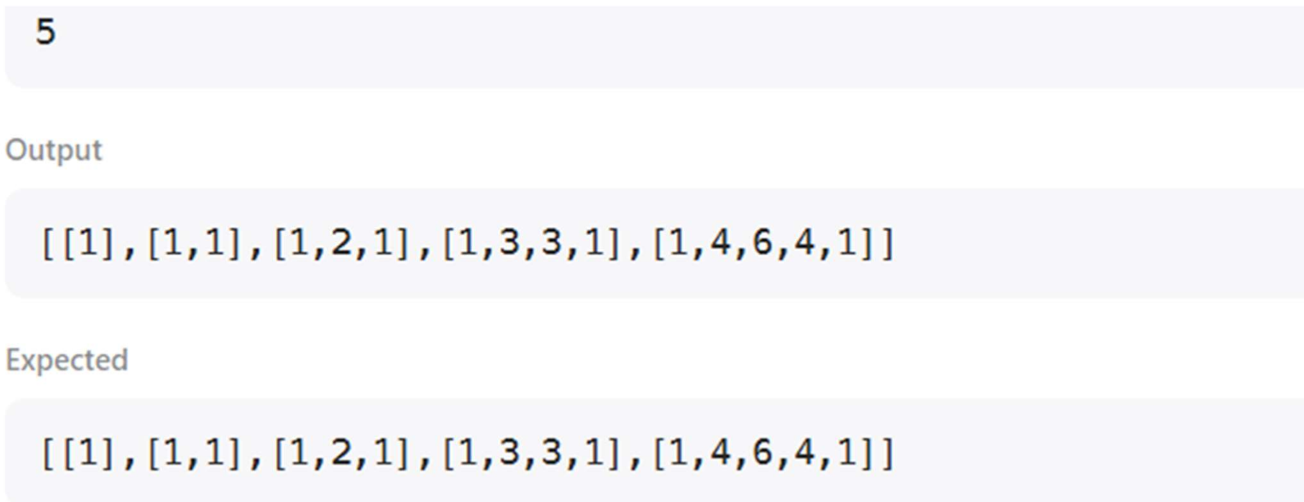

Figure 1

5. Learning Outcomes:

- **Improve logical thinking**: Through pattern recognition and rule-following (each entry = sum of two above).
- **Link algebra and geometry**: Make connections between abstract math and visual representations.
- **Build mathematical curiosity**: Use the triangle to spark exploration in number theory.

<u>Problem-2</u>

1. **Aim:** Task Scheduler
2. **Objectives:**

- **Understand process scheduling** in operating systems.
- **Learn different scheduling algorithms** like FCFS, Round Robin, SJF, Priority Scheduling.
- **Manage resources efficiently** through task scheduling strategies.
- **Implement real-time and non-real-time task scheduling**.
- **Develop simulations or software models** of scheduling systems.

3. Implementation/Code:

```
class Solution {
public:
    int leastInterval(vector<char>& tasks, int n) {
        // Building frequency map
        int freq[26] = {0};
        for (char &ch : tasks) {
```

```
            freq[ch - 'A']++;
        }

        // Max heap to store frequencies
        priority_queue<int> pq;
        for (int i = 0; i < 26; i++) {
            if (freq[i] > 0) {
                pq.push(freq[i]);
            }
        }

        int time = 0;
        // Process tasks until the heap is empty
        while (!pq.empty()) {
            int cycle = n + 1;
            vector<int> store;
            int taskCount = 0;
            // Execute tasks in each cycle
            while (cycle-- && !pq.empty()) {
                if (pq.top() > 1) {
                    store.push_back(pq.top() - 1);
                }
                pq.pop();
                taskCount++;
            }
            // Restore updated frequencies to the heap
            for (int &x : store) {
                pq.push(x);
            }
            // Add time for the completed cycle
            time += (pq.empty() ? taskCount : n + 1);
        }
        return time;
    }
};
```

\

Output:

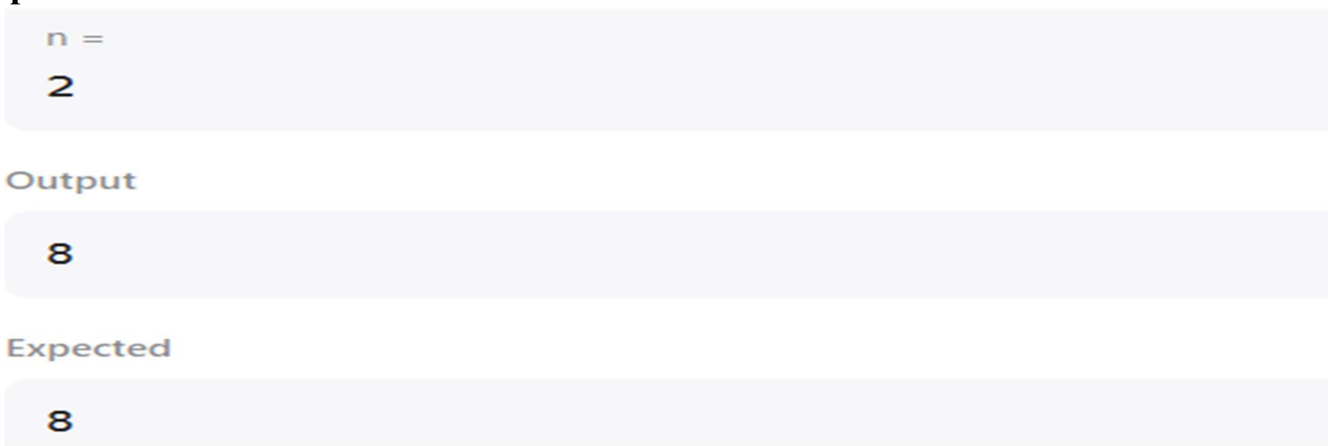

Figure 2

4. Learning Outcomes:

- Explain the need for task scheduling in multi-tasking systems or operating systems.
- Differentiate between various scheduling algorithms and their use cases.
- Implement basic scheduling algorithms in code (e.g., using C, Java, or Python).
- Analyze the performance of scheduling strategies using metrics like turnaround time, waiting time, and throughput.
- Simulate a scheduler that handles tasks with constraints like deadlines or priorities.
- Understand the role of schedulers in real-time systems, embedded devices, and cloud environments.

Problem: - 3

1. **Aim:** Number of 1 bit
2. **Objectives:**

- **Understand bitwise operations** in programming.
- **Learn how binary representation of numbers works**.
- **Implement an efficient method** to count the number of 1s in a binary number.
- **Explore multiple approaches** (looping, bit masking, built-in functions, etc.).
- **Improve understanding of low-level computation** and memory efficiency.

3. **Implementation/Code:**

```
class Solution {
public:
    int hammingWeight(uint32_t n) {
```

```
        int res = 0;
        for (int i = 0; i < 32; i++) {
            if ((n >> i) & 1) {
                res += 1;
            }
        }
        return res;
    }
};
```

4. Output:

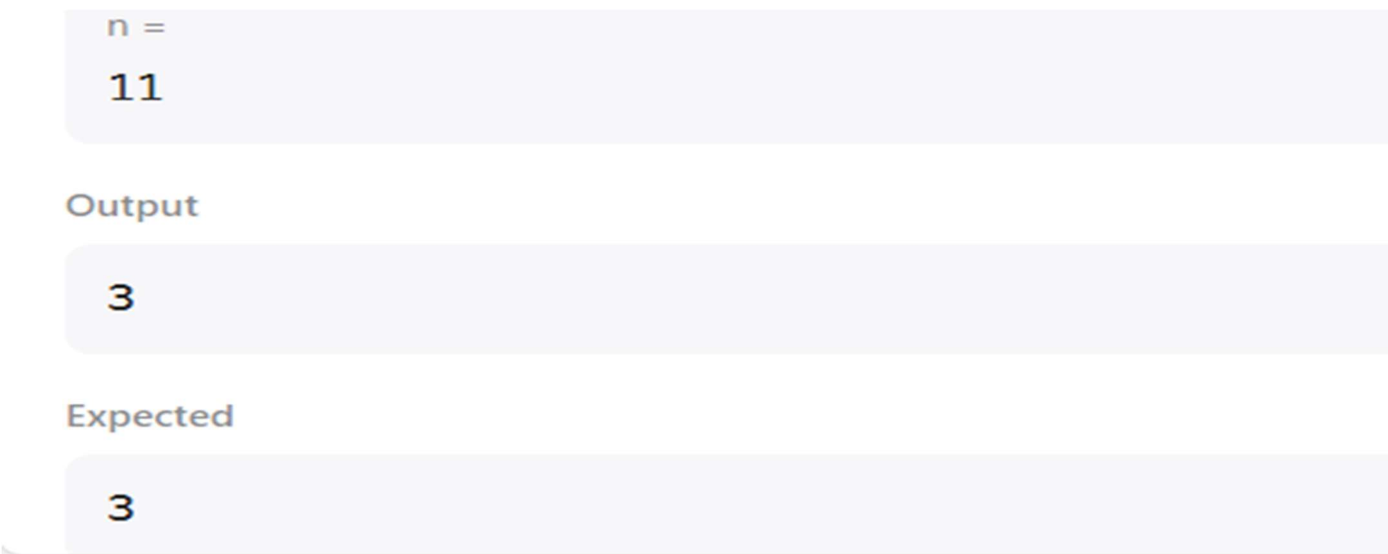

Figure 3

5. Learning Outcomes:

- You will be able to find the lowest common ancestor of two given nodes in a binary tree. This will help in solving hierarchical tree problems.
- You will understand how recursion helps in solving complex tree-based problems. This will improve your ability to write efficient recursive functions.
- You will learn to apply depth-first search (DFS) to navigate through trees. This will make it easier to find specific nodes and their ancestors.
- You will gain confidence in handling base cases and edge cases in recursive solutions. This will ensure your code runs correctly for all scenarios.
- You will be able to write clear and optimized C++ code for tree problems. This will strengthen your programming skills and logical thinking.

Problem 4

1.Aim: Divide two Integer

2.Objective:

- To understand how integer division works at a low level without using built-in arithmetic operators.
- To develop an algorithm that performs division using alternative techniques such as bit manipulation or subtraction.
- To handle edge cases in integer division, including overflow, negative numbers, and zero handling.
- To improve problem-solving and logical thinking by implementing mathematical operations manually.
- To analyze time and space complexity of custom arithmetic algorithms.

3.Code Implementation:

```
class Solution {
public:
    int divide(int dividend, int divisor) {
        if (dividend == divisor) return 1;
        if (dividend == INT_MIN && divisor == -1) return INT_MAX;
        if (divisor == 1) return dividend;

        int sign = (dividend < 0) ^ (divisor < 0) ? -1 : 1;

        long long n = abs((long long)dividend);
        long long d = abs((long long)divisor);
        int ans = 0;

        while (n >= d) {
            int p = 0;
            while (n >= (d << p)) p++;
            p--;
            n -= (d << p);
            ans += (1 << p);
        }
```

```
        return sign * ans;
    }
};
```

5.Output:

divisor =

3

Output

3

Expected

3

6.Learning Outcome:

- Explain how integer division can be simulated using subtraction and bit shifts.
- Implement an efficient function to divide two integers without using multiplication, division, or modulus operators.
- Handle special cases such as division by zero and integer overflow.
- Demonstrate a deeper understanding of binary arithmetic and two's complement representation.
- Optimize the algorithm for better performance and understand its time complexity.