Experiment-9(A)

Student Name: Ayush Pathania **UID:** 22BCS16023
Branch: CSE **Section/Group:** NTPP_602-A
Semester: 6 **Date of Performance:** 15-03-25
Subject Name: Advanced Programming Lab-2 **Subject Code:** 22CSH-359

1. Title: Miscellaneous (Hamming Distance)

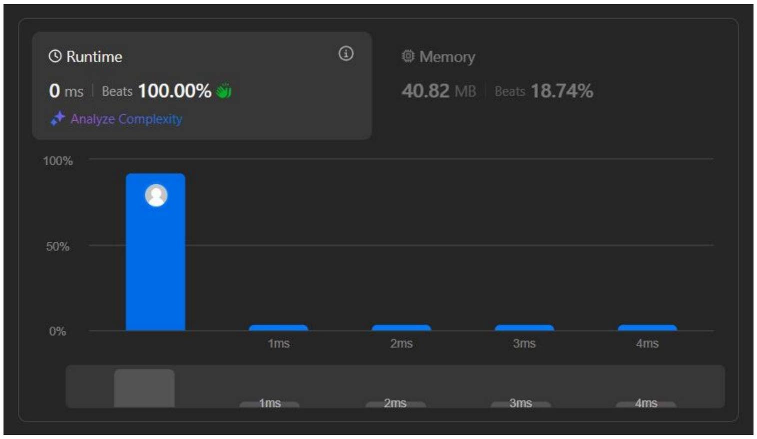
2. Objective: To calculate the number of differing bits between two integers.

3. **Algorithm:**
   - **Input:** Two integers `x` and `y`.
   - **XOR Operation:**
     - XOR the two numbers to identify differing bits.
   - **Bit Count:**
     - Count the number of `1`s in the XOR result.

4. **Implementation/Code:**

```
class Solution {
        public int hammingDistance(int x, int y) { return
                Integer.bitCount(x^y);
        }
}
```

5. **Output:**



Runtime

0 ms | Beats 100.00%

Analyze Complexity

Memory

40.82 MB | Beats 18.74%

6. **Time Complexity:** O (1) 7. **Space Complexity:** O(1)

Experiment 9(B)

1. **Title:** Divide Two Integers

2. **Objective:** To perform integer division without using multiplication, division, or modulo operators.

3. **Algorithm:**

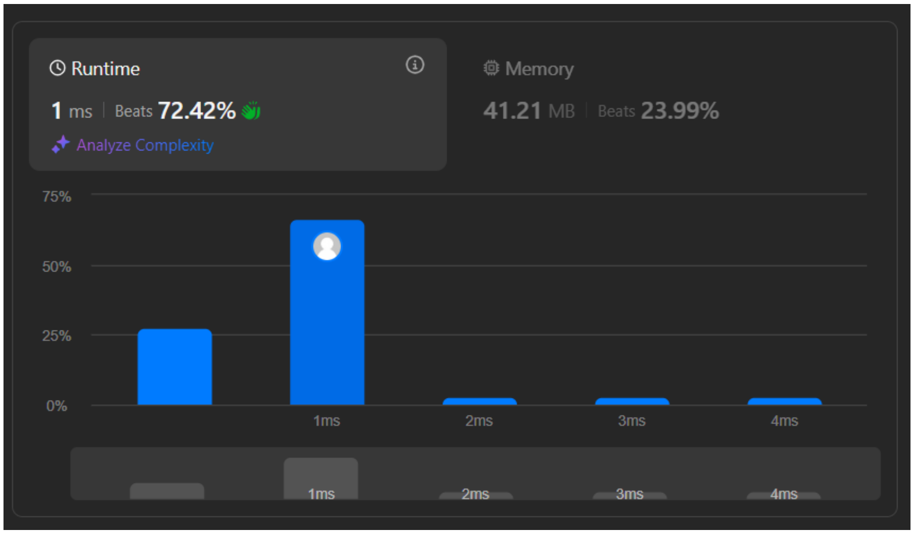
- **Input:** Two integers `dividend` and `divisor`.
- **Handle Edge Cases:**
    - If `dividend = Integer.MIN_VALUE` and `divisor = -1`, return `Integer.MAX_VALUE`.
- **Sign Calculation:**
    - Calculate the sign using XOR: `(dividend < 0) ^ (divisor < 0)`.
- **Convert to Positive:**
    - Take absolute values of `dividend` and `divisor`.
- **Repeated Subtraction (Bitwise Shift):**
    - Iterate while `dividend >= divisor`.
    - Continuously shift the divisor left by 1 and subtract to accumulate the result.
- **Output:** Return the result with the calculated sign.

4. **Implementation/Code:**

```
class Solution {
        public int divide(int dividend, int divisor) {
                if (dividend == Integer.MIN_VALUE && divisor == -1) return
                        Integer.MAX_VALUE;

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

                Math.abs((long) dividend);
                long ldivisor = Math.abs((long) divisor);

                int result = 0;
                while (ldividend >= ldivisor) {
                        long temp = ldivisor, multiple = 1;
```

```
                    while (dividend >= (temp << 1)) { temp <<= 1;
                        multiple <<= 1;
                    }

                    ldividend -= temp; result +=
                    multiple;
                }

                return sign * result;
            }
        }
```

5. Output:



Runtime

1 ms | Beats 72.42%

Analyze Complexity

Memory

41.21 MB | Beats 23.99%

6. Time Complexity: O(log n)

7. Space Complexity: O(1)

8. Learning Outcome:

- Learned efficient bitwise operations for arithmetic calculations.
- Mastered handling of integer limits and edge cases.

Experiment 9(C)

1. **Title:** Pascal's Triangle

2. **Objective:** To generate the first numRows of Pascal's Triangle.

3. Algorithm:

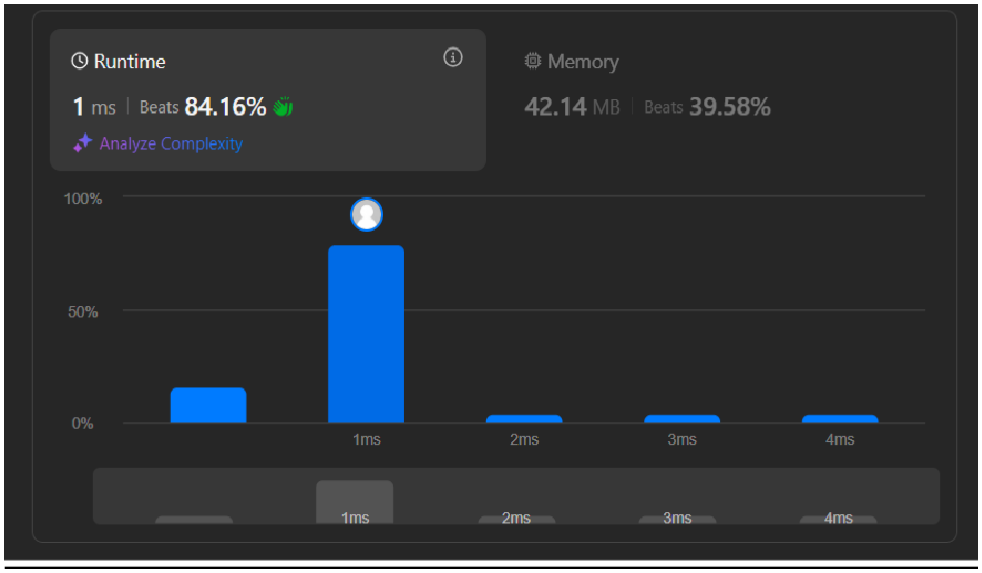
- **Input:** An integer numRows.
- **Initialization:** Create an empty list triangle to store the rows.
- **Iteration:**

- For each row i:
    - Create a list with i + 1 elements, initialized to 1.
    - For each element j from index 1 to i - 1:
        - Set row[j] = triangle[i-1][j-1] + triangle[i-1][j].
    - Append this row to triangle.
- **Output:** Return the triangle.

4. Implementation/Code:

```
import java.util.*;

class Solution {
        public List<List<Integer>> generate(int numRows) { List<List<Integer>> triangle =
                new ArrayList<>();

                for (int i = 0; i < numRows; i++) {
                        List<Integer> row = new ArrayList<>(Collections.nCopies(i + 1,
1));

                        for (int j = 1; j < i; j++) {
                                row.set(j, triangle.get(i - 1).get(j - 1) + triangle.get(i -
1).get(j));
                        }

                        triangle.add(row);
                }

                return triangle;
        }
}
```

5. Output:



Runtime

1 ms | Beats 84.16%

Memory

42.14 MB | Beats 39.58%

8. Time Complexity:_O(n^2)

9. Space Complexity: O(n^2)

10. LearningOutcomes:

- Learned efficient bit manipulation techniques.
- Understood the XOR operation for identifying differing bits.
- Gained a better understanding of combinatorial mathematics.
- Practiced 2D array manipulation in Java.