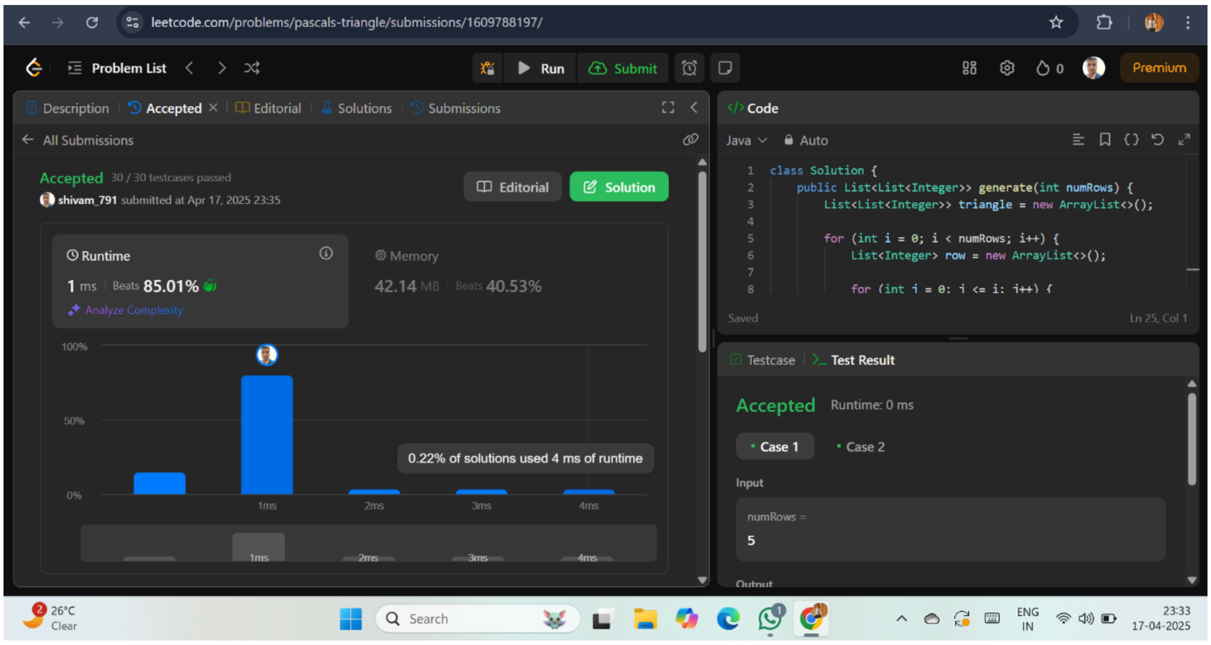
# AP Assignment – 10

**Name –** Shivam Gautam **UID –** 22BCS12184

**Q1** Given an integer numRows, return the first numRows of Pascal's triangle. In Pascal's triangle, each number is the sum of the two numbers.

**Code –**

```
 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<>();
        for (int j = 0; j <= i; j++) {
            if (j == 0 || j == i) {
                row.add(1);
            } else {
                    int sum = triangle.get(i - 1).get(j - 1) +          triangle.get(i - 1).get(j);
                row.add(sum);
            }
        }
        triangle.add(row);
    }
    return triangle;
}
}
```



Accepted 30 / 30 testcases passed
shivam_791 submitted at Apr 17, 2025 23:35

Runtime
1 ms | Beats 85.01%

Memory
42.14 MB | Beats 40.53%

0.22% of solutions used 4 ms of runtime

```
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<>();

            for (int j = 0; j <= i; j++) {
```

Saved    Ln 25, Col 1

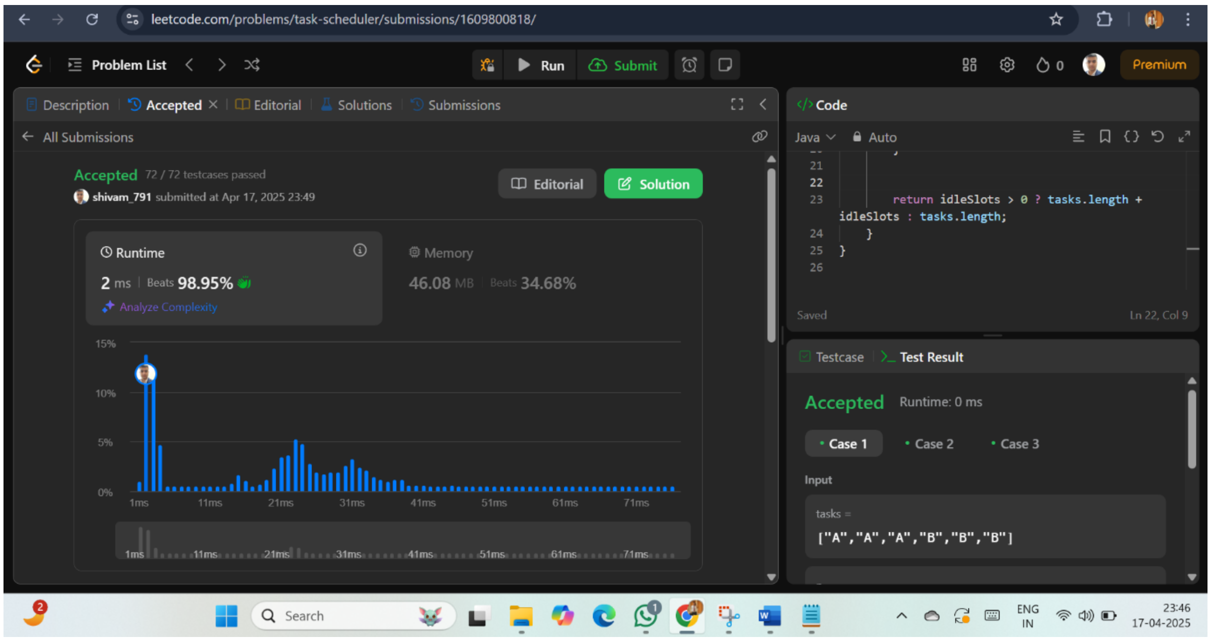
Accepted Runtime: 0 ms

Input
numRows =
5

**Q2** You are given an array of CPU tasks, each labeled with a letter from A to Z, and a number n. Each CPU interval can be idle or allow the completion of one task. Tasks can be completed in any order, but there's a constraint: there has to be a gap of at least n intervals between two tasks with the same label. Return the minimum number of CPU intervals required to complete all tasks.

## Code –

```
import java.util.*;
class Solution {
    public int leastInterval(char[] tasks, int n) {
        int[] freq = new int[26];
        for (char task : tasks) {
            freq[task - 'A']++;
        }
        Arrays.sort(freq);
        int maxFreq = freq[25] - 1;
        int idleSlots = maxFreq * n;
        for (int i = 24; i >= 0 && freq[i] > 0; i--) {
            idleSlots -= Math.min(freq[i], maxFreq);
        }
        return idleSlots > 0 ? tasks.length + idleSlots : tasks.length;
    }
}
```



Accepted 72 / 72 testcases passed

shivam_791 submitted at Apr 17, 2025 23:49

Runtime 2 ms | Beats 98.95%

Memory 46.08 MB | Beats 34.68%

```
        return idleSlots > 0 ? tasks.length +
idleSlots : tasks.length;
    }
}
```

Accepted Runtime: 0 ms

Case 1 · Case 2 · Case 3

Input

tasks =
["A","A","A","B","B","B"]

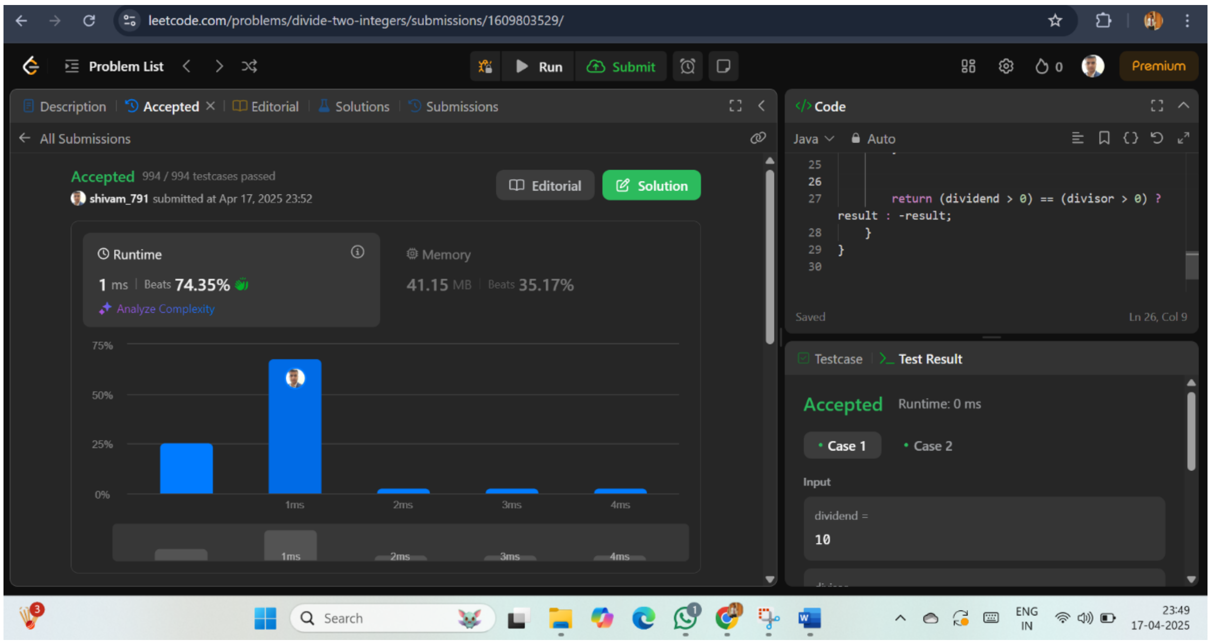
**Q3** Given two integers dividend and divisor, divide two integers without using multiplication, division, and mod operator. The integer division should truncate toward zero, which means losing its fractional part. For example, 8.345 would be truncated to 8, and -2.7335 would be truncated to -2.

Return *the quotient after dividing* dividend *by* divisor.

Note: Assume we are dealing with an environment that could only store integers within the 32-bit signed integer range: $[-2^{31}, 2^{31} - 1]$. For this problem, if the quotient is strictly greater than $2^{31} - 1$, then return $2^{31} - 1$, and if the quotient is strictly less than $-2^{31}$, then return $-2^{31}$**.**

## Code –

```
class Solution {
    public int divide(int dividend, int divisor) {
        if (dividend == Integer.MIN_VALUE && divisor == -1) {
            return Integer.MAX_VALUE;     }
        long lDividend = Math.abs((long) dividend);
        long lDivisor = Math.abs((long) divisor);
        int result = 0;
        while (lDividend >= lDivisor) {
            long temp = lDivisor, multiple = 1;
            while (lDividend >= (temp << 1)) {
                temp <<= 1;
                multiple <<= 1;
            }
            lDividend -= temp;
            result += multiple;
        }
        return (dividend > 0) == (divisor > 0) ? result : -result;
    }
}
```



Accepted 994 / 994 testcases passed

shivam_791 submitted at Apr 17, 2025 23:52

Runtime
1 ms | Beats 74.35%

Memory
41.15 MB | Beats 35.17%

```
        return (dividend > 0) == (divisor > 0) ?
result : -result;
    }
}
```

Accepted Runtime: 0 ms

dividend =
10

**Q4** You have n tasks and m workers. Each task has a strength requirement stored in a 0-indexed integer array tasks, with the $i^{th}$ task requiring tasks[i] strength to complete. The strength of each worker is stored in a 0-indexed integer array workers, with the $j^{th}$ worker having workers[j] strength. Each worker can only be assigned to a single task and must have a strength greater than or equal to the task's strength requirement (i.e., workers[j] >= tasks[i]).

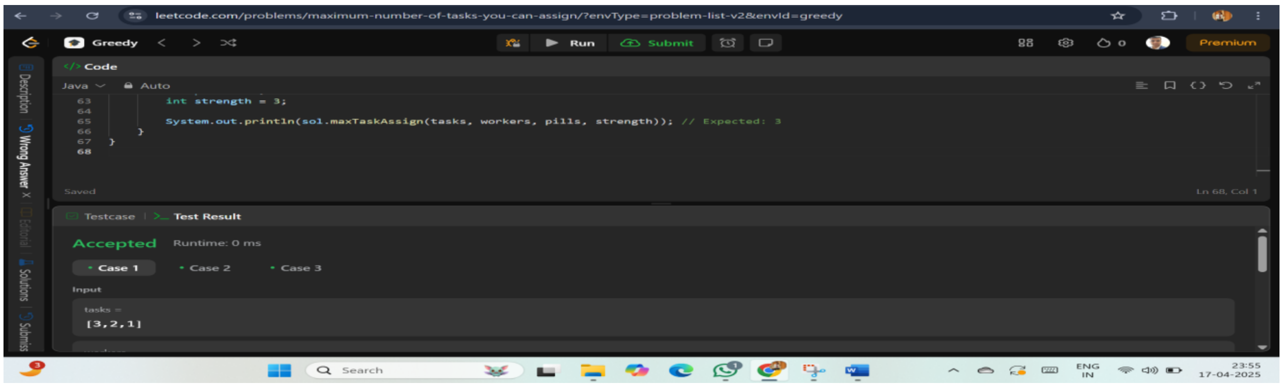
Additionally, you have pills magical pills that will increase a worker's strength by strength. You can decide which workers receive the magical pills, however, you may only give each worker at most one magical pill.

Given the 0-indexed integer arrays tasks and workers and the integers pills and strength, return *the maximum number of tasks that can be completed.*

## Code –

```
import java.util.*;

class Solution {

  public int maxTaskAssign(int[] tasks, int[] workers, int pills, int strength) {

    Arrays.sort(tasks);

    Arrays.sort(workers);

    int low = 0, high = Math.min(tasks.length, workers.length), result = 0;

    while (low <= high) {

      int mid = (low + high) / 2;

      if (canAssign(mid, tasks, workers, pills, strength)) {

        result = mid;

        low = mid + 1;

      } else {

        high = mid - 1;

      }

    }

    return result;

  }
```

```
    private boolean canAssign(int k, int[] tasks, int[] workers, int pills, int strength) {
        Deque<Integer> deque = new ArrayDeque<>();
        for (int i = workers.length - k; i < workers.length; i++) {
            deque.addLast(workers[i]);
        }
        int pillsLeft = pills;
        for (int i = k - 1; i >= 0; i--) {
            int taskStrength = tasks[i];
            if (!deque.isEmpty() && deque.peekLast() >= taskStrength) {
                deque.pollLast();
            }
            else if (!deque.isEmpty() && pillsLeft > 0 && deque.peekFirst() + strength >=
taskStrength) {
                deque.pollFirst();
                pillsLeft--;
            } else {
                return false;
            }
        }
        return true;
    }
}
```


leetcode.com/problems/maximum-number-of-tasks-you-can-assign/?envType=problem-list-v2&envId=greedy

```
        int strength = 3;

        System.out.println(sol.maxTaskAssign(tasks, workers, pills, strength)); // Expected: 3
    }
}
```

Accepted Runtime: 0 ms

Input

tasks =
[3,2,1]