ASSIGNMENT 10

Student Name: Ramit Chaturvedi **UID:22BCS15597**
Branch: CSE **Section/Group:614-B**
Semester: 6 **Date of Performance:17-04-25**
Subject Name: AP LAB-II **Subject Code: 22CSP-351**

Q 1) **Pascal's Triangle**

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 directly above it as shown:

Code)

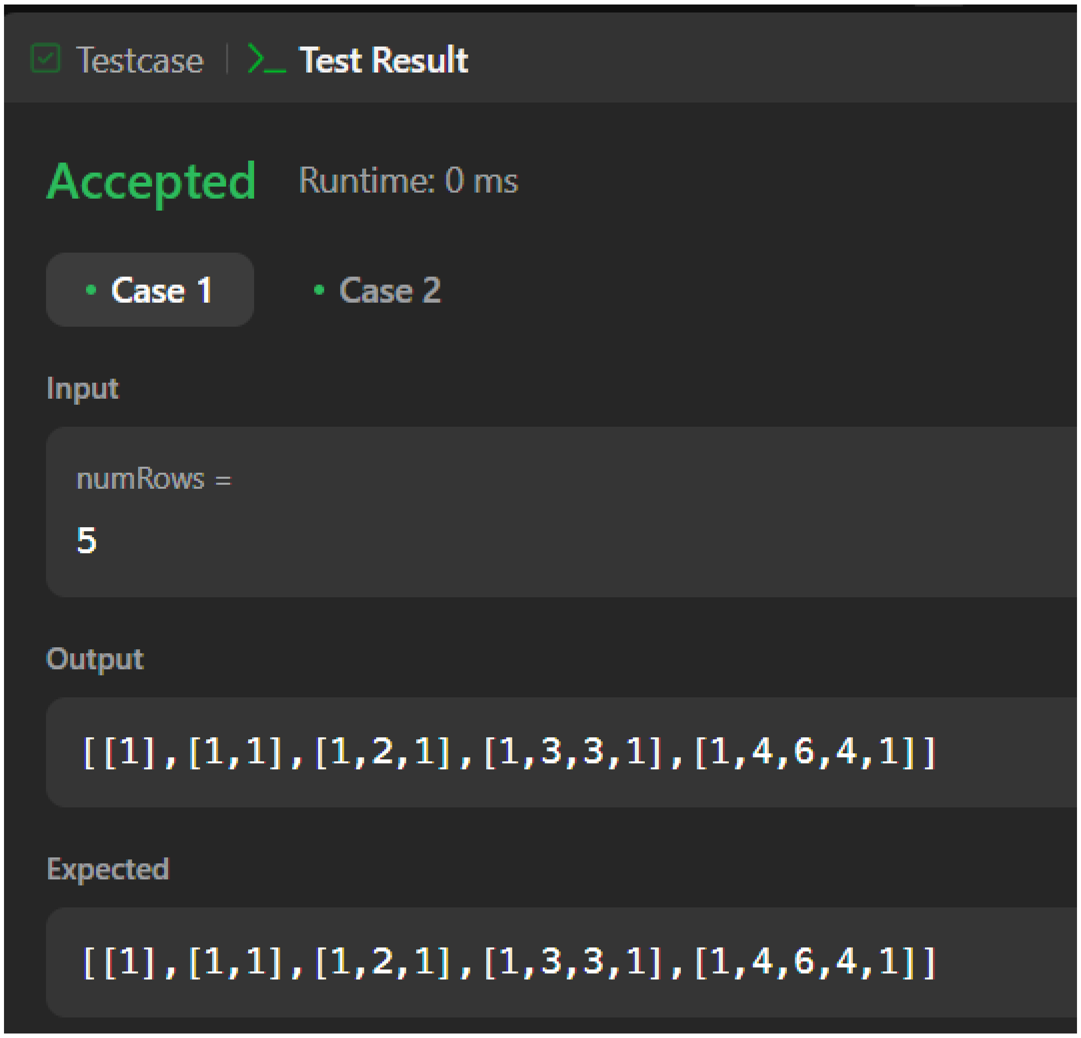
```
class Solution {
 public:
  vector<vector<int>> generate(int numRows) {
    vector<vector<int>> ans;

    for (int i = 0; i < numRows; ++i)
      ans.push_back(vector<int>(i + 1, 1));

    for (int i = 2; i < numRows; ++i)
      for (int j = 1; j < ans[i].size() - 1; ++j)
        ans[i][j] = ans[i - 1][j - 1] + ans[i - 1][j];

    return ans;
  }
};
```

Output)



Accepted Runtime: 0 ms

Input

numRows =
5

Output

```
[[1],[1,1],[1,2,1],[1,3,3,1],[1,4,6,4,1]]
```

Expected

```
[[1],[1,1],[1,2,1],[1,3,3,1],[1,4,6,4,1]]
```

Q 2) Hamming Distance

The Hamming distance between two integers is the number of positions at which the corresponding bits are different.

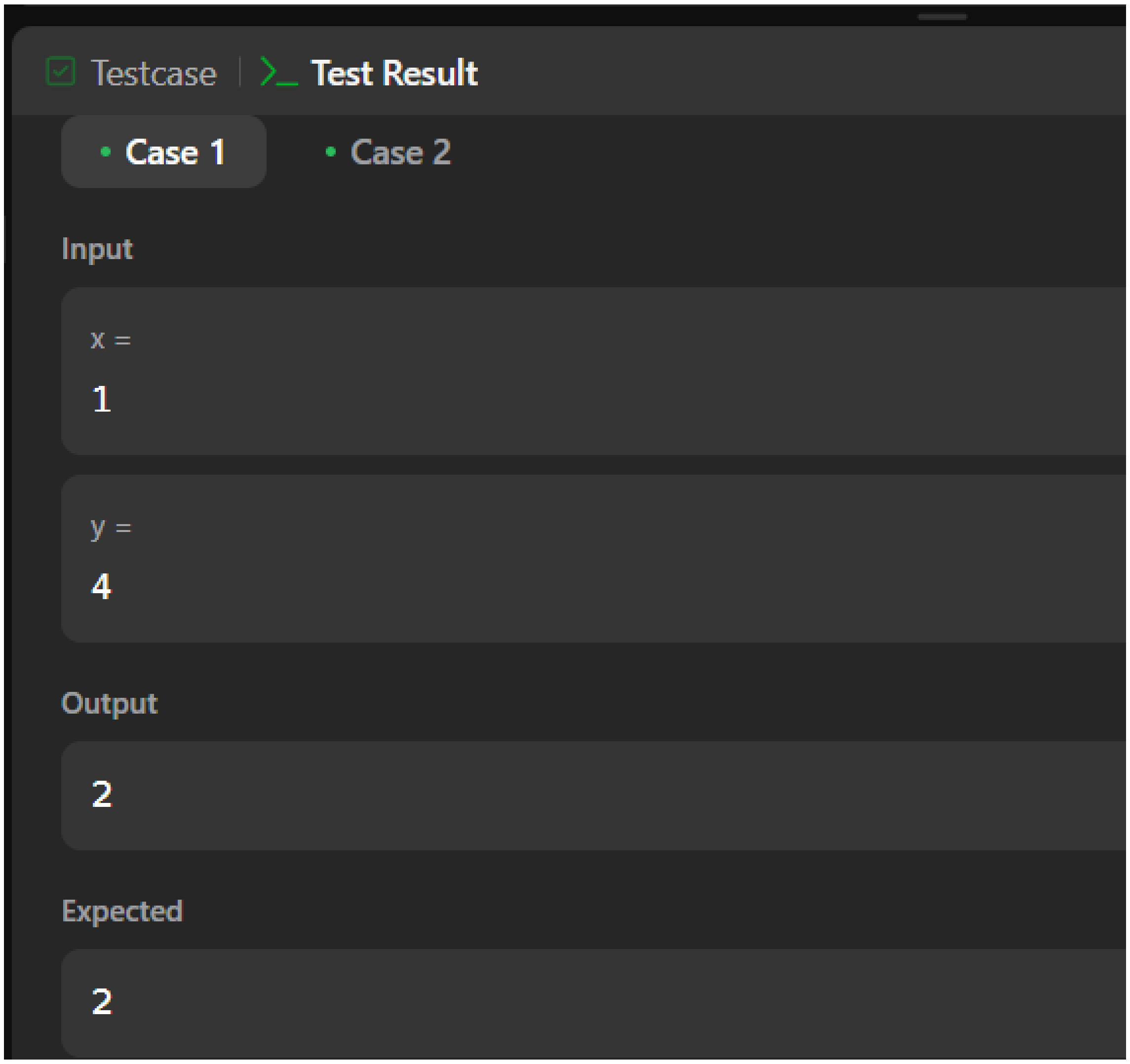
Given two integers x and y, return *the Hamming distance between them*.

Code)

```
class Solution {
 public:
  int hammingDistance(int x, int y) {
    int ans = 0;
    while (x > 0 || y > 0) {
      ans += (x & 1) ^ (y & 1);
      x >>= 1;
      y >>= 1;
    }

    return ans;
  }
};
```

Output)



Input

x =
1

y =
4

Output

2

Expected

2

3) Task Scheduler

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)

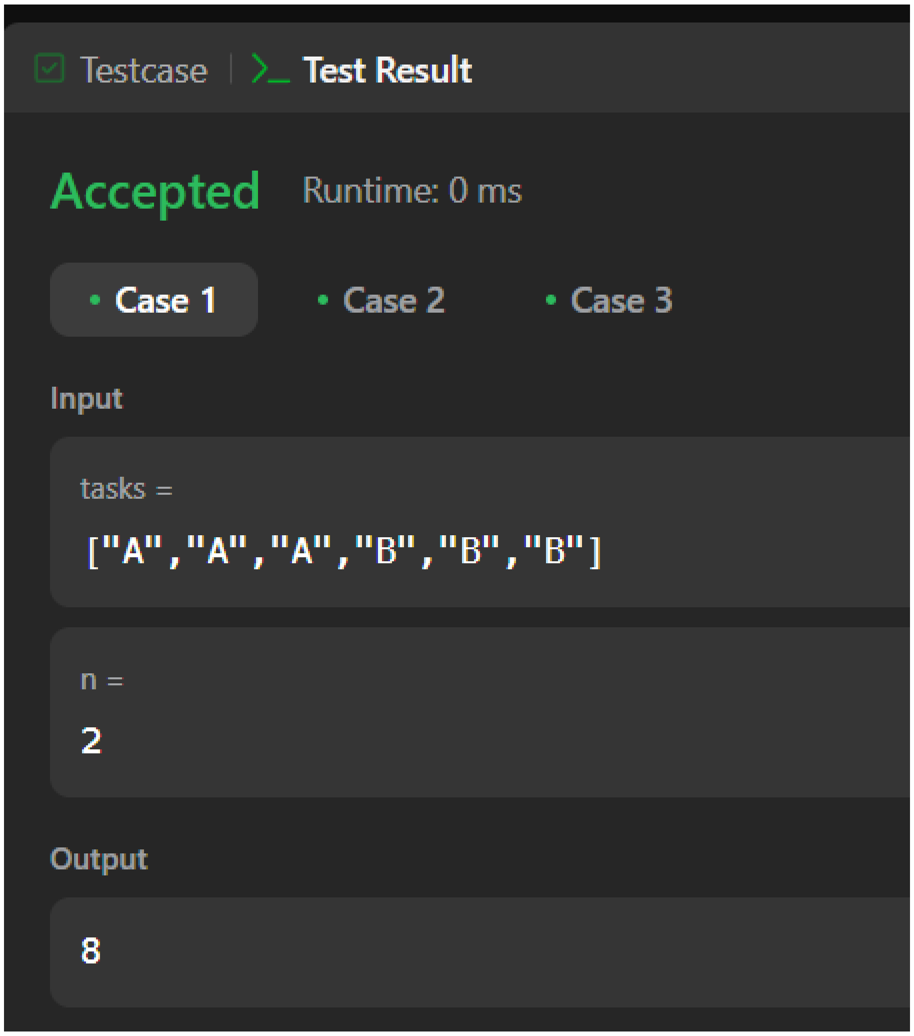
```
class Solution {
 public:
  int leastInterval(vector<char>& tasks, int n) {
    if (n == 0)
      return tasks.size();

    vector<int> count(26);

    for (const char task : tasks)
      ++count[task - 'A'];

    const int maxFreq = ranges::max(count);
    // Put the most frequent task in the slot first.
    const int maxFreqTaskOccupy = (maxFreq - 1) * (n + 1);
    // Get the number of tasks with the same frequency as `maxFreq`, we'll
    // append them after `maxFreqTaskOccupy`.
    const int nMaxFreq = ranges::count(count, maxFreq);
    // max(
    //   the most frequent task is frequent enough to force some idle slots,
    //   the most frequent task is not frequent enough to force idle slots
    // )
    return max(maxFreqTaskOccupy + nMaxFreq, static_cast<int>(tasks.size()));
  }
};
```

Output)



```
Testcase | >_ Test Result

Accepted   Runtime: 0 ms

• Case 1   • Case 2   • Case 3

Input

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

n =
2

Output

8
```

4) Number of 1 Bits

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).

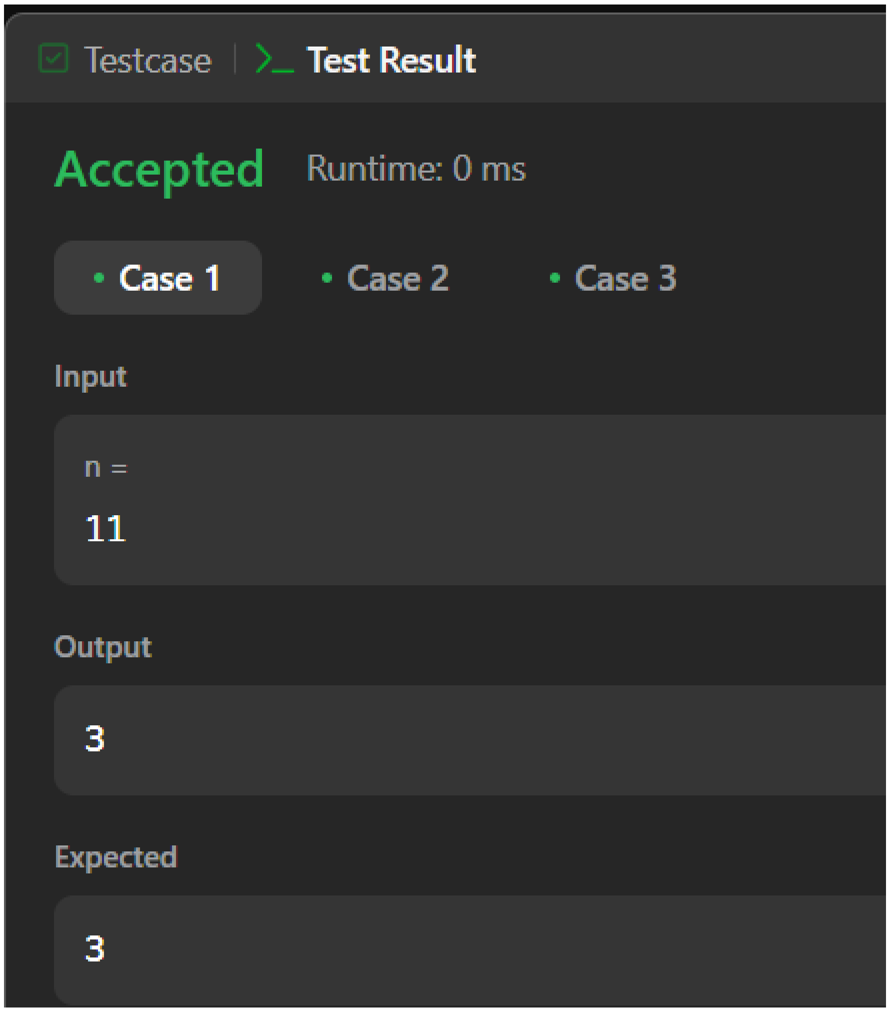
Code)

```
class Solution {
 public:
  int hammingWeight(uint32_t n) {
    int ans = 0;

    for (int i = 0; i < 32; ++i)
      if ((n >> i) & 1)
        ++ans;

    return ans;
  }
};
```

Output)



Testcase | Test Result

Accepted Runtime: 0 ms

• Case 1 • Case 2 • Case 3

Input

```
n =
11
```

Output

```
3
```

Expected

```
3
```

5) **Divide Two Integers**

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)

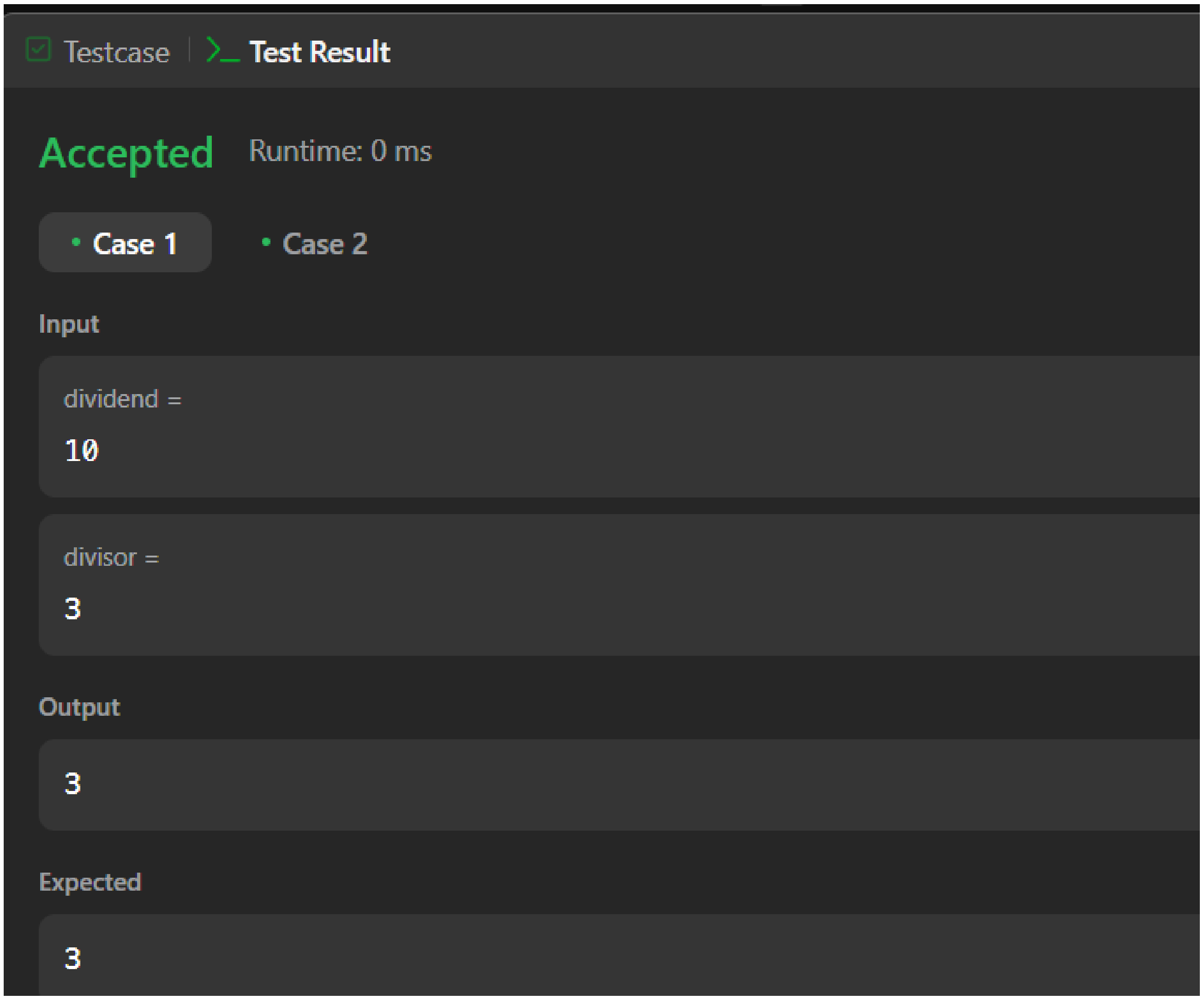
```
class Solution {
 public:
  int divide(int dividend, int divisor) {
    // -2^{31} / -1 = 2^31 will overflow, so return 2^31 - 1.
    if (dividend == INT_MIN && divisor == -1)
      return INT_MAX;

    const int sign = dividend > 0 ^ divisor > 0 ? -1 : 1;
    long ans = 0;
    long dvd = labs(dividend);
    long dvs = labs(divisor);

    while (dvd >= dvs) {
      long k = 1;
      while (k * 2 * dvs <= dvd)
        k *= 2;
      dvd -= k * dvs;
      ans += k;
    }

    return sign * ans;
  }
};
```

Output)



```
Accepted    Runtime: 0 ms

Case 1    Case 2

Input
dividend =
10
divisor =
3

Output
3

Expected
3
```