



Experiment 4

Student Name: Rohan

Branch: BE-IT

Semester: 6th

Subject Name: Advanced Programming Lab-2

UID: 22BET10169

Section/Group: 22BET_IOT_701/A

Date of Performance: 14-2-25

Subject Code: 22ITP-351

Problem 1. Given a string *s*, return the longest substring of *s* that is nice. If there are multiple, return the substring of the earliest occurrence. If there are none, return an empty string.

Algorithm:

1. Base Case:

- If the length of *s* is less than 2, return an empty string ("") because a single character cannot be "nice".

2. Create a Set of Characters in *s*:

- Store all characters of *s* in a hash set for quick lookup.

3. Find a Split Point:

- Traverse through *s*. If a character appears in only one case (i.e., either uppercase or lowercase but not both), this means the substring cannot be nice.
- Use this character as a **split point** and break *s* into two substrings:
 - left = *s* [0: *i*]
 - right = *s* [*i*+1:]
- Recursively find the longest "nice" substring in both parts.

4. Compare Substrings:

- Return the longer of the two substrings found in step 3.

5. If No Split Occurs:

- If no character was found that requires splitting, return *s* itself as it is already "nice".

Code:

```
class Solution {
public:
    string longestNiceSubstring(string s) {
        if (s.length() < 2) return "";

        unordered_set<char> charSet(s.begin(), s.end());

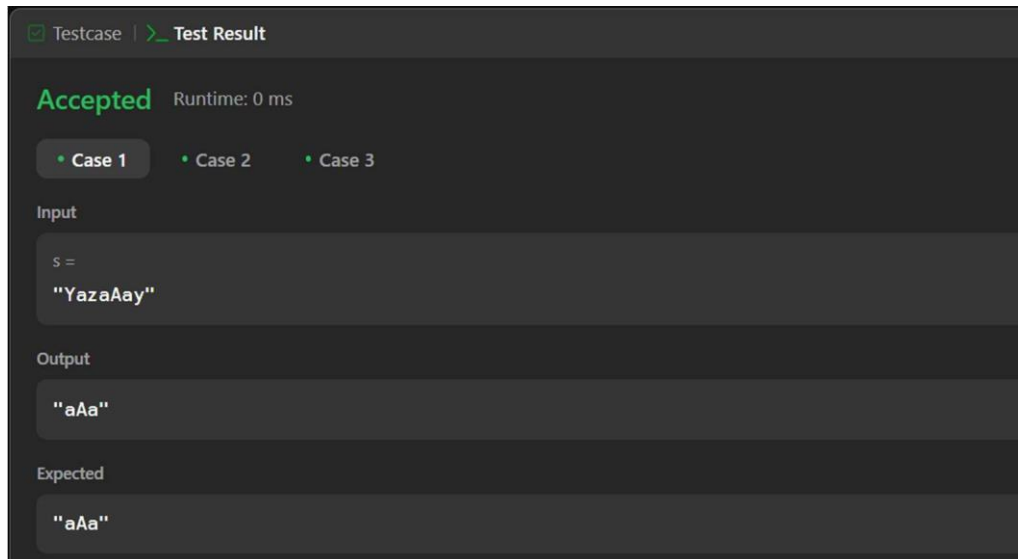
        for (int i = 0; i < s.length(); i++) {
            if (charSet.count(tolower(s[i])) && charSet.count(toupper(s[i]))) {
                continue;
            }

            string left = longestNiceSubstring(s.substr(0, i));
            string right = longestNiceSubstring(s.substr(i + 1));
```

```
        return (left.length() >= right.length()) ? left : right;
    }

    return s;
}
};
```

Output:



Problem 2. Reverse bits of a given 32 bits unsigned integer.

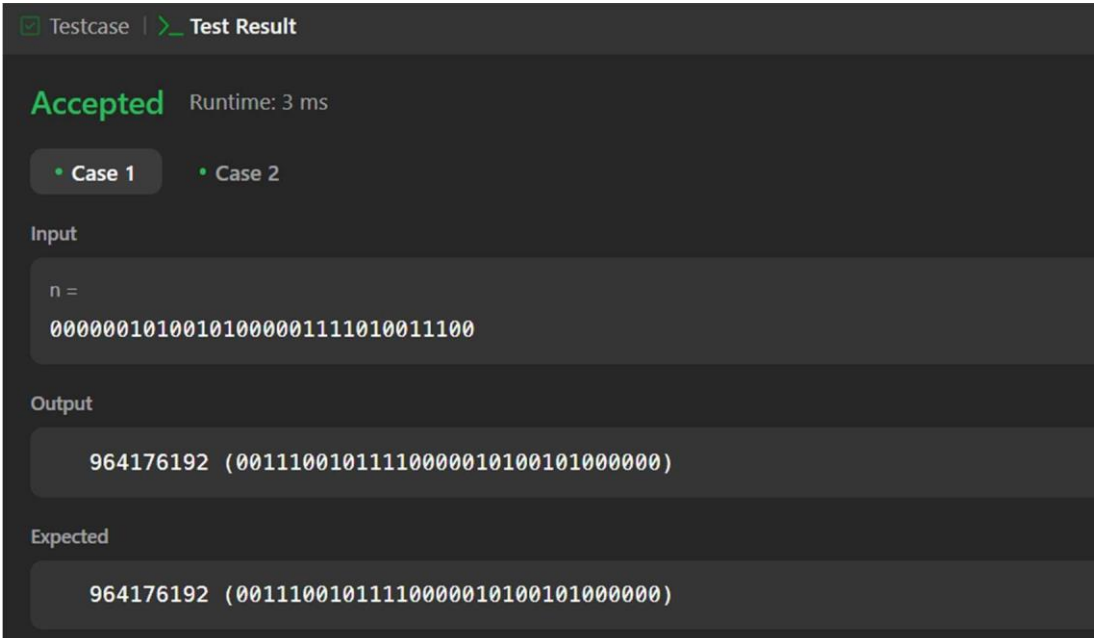
Algorithm:

1. Initialize result to 0
 - This variable will store the reversed bits.
2. Iterate 32 times (since it's a 32-bit integer)
 - Extract the least significant bit (LSB) of n using $n \& 1$.
 - Shift result left by 1 to make space for the new bit.
 - Add the extracted bit to result.
 - Right shift n by 1 to process the next bit.
3. Return result
 - The final value of result contains the reversed bit sequence.

Code:

```
class Solution {
public:
    uint32_t reverseBits(uint32_t n) {
        uint32_t result = 0;
        for (int i = 0; i < 32; i++) {
            result = (result << 1) | (n & 1);
            n >>= 1;
        }
        return result;
    }
};
```

Output:



Testcase | Test Result

Accepted Runtime: 3 ms

• Case 1 • Case 2

Input

n =
00000010100101000001111010011100

Output

964176192 (00111001011110000010100101000000)

Expected

964176192 (00111001011110000010100101000000)

Problem 3. 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](#)).

Algorithm:

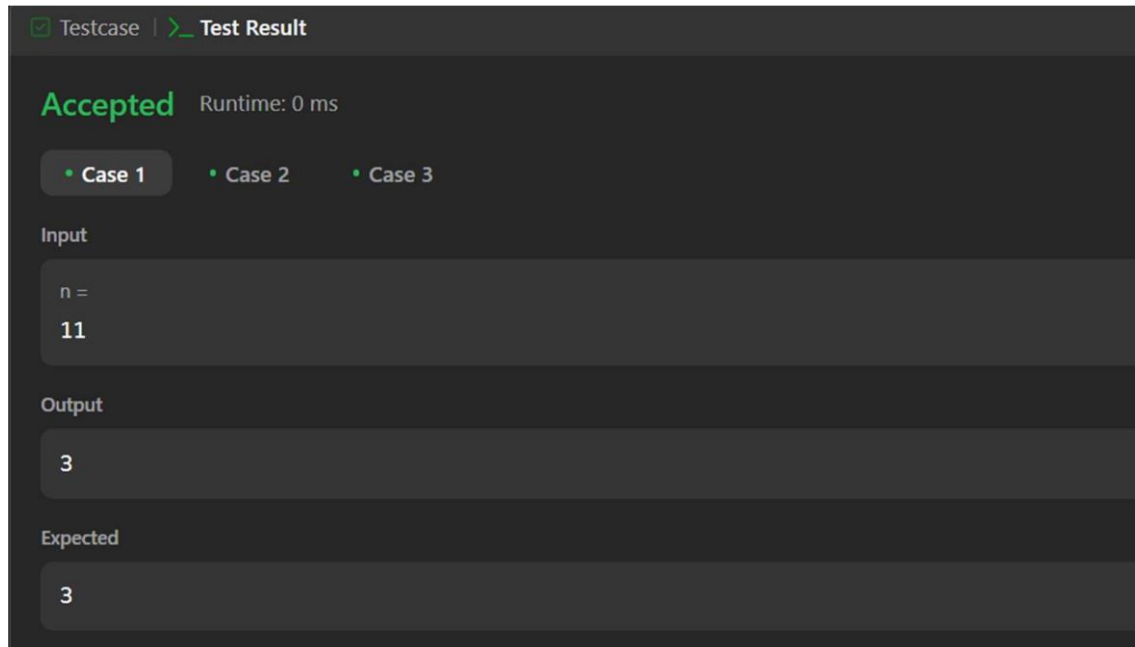
Steps

1. Initialize count = 0 to store the number of 1 bits.
2. Iterate through all 32 bits of the integer:
 - Check if the last bit is 1 using $(n \& 1)$.
 - If true, increment count.
 - Right shift n ($n = n \gg 1$) to check the next bit.
3. Return count after processing all bits.

Code:

```
class Solution {
public:
    int hammingWeight(int n) {
        int count = 0;
        while (n != 0) {
            count += (n & 1);
            n >>= 1;
        }
        return count;
    }
};
```

Output:



Problem 4. Given an integer array `nums`, find the subarray with the largest sum, and return *its sum*.

Algorithm:

1. Initialize Variables

- `maxsum = nums[0]` → Stores the maximum subarray sum found so far.
- `currsum = 0` → Tracks the sum of the current subarray.

2. Iterate Through the Array

- If `currsum` becomes negative, reset it to 0 (since a negative sum reduces the potential maximum).
- Add the current element `num` to `currsum`.
- Update `maxsum = max(maxsum, currsum)`.

4. Return **maxsum**, which stores the maximum subarray sum.

Code:

```
class Solution {
public:
    int maxSubArray(vector<int>& nums) {
        int maxsum = nums[0];
        int currsum = 0;

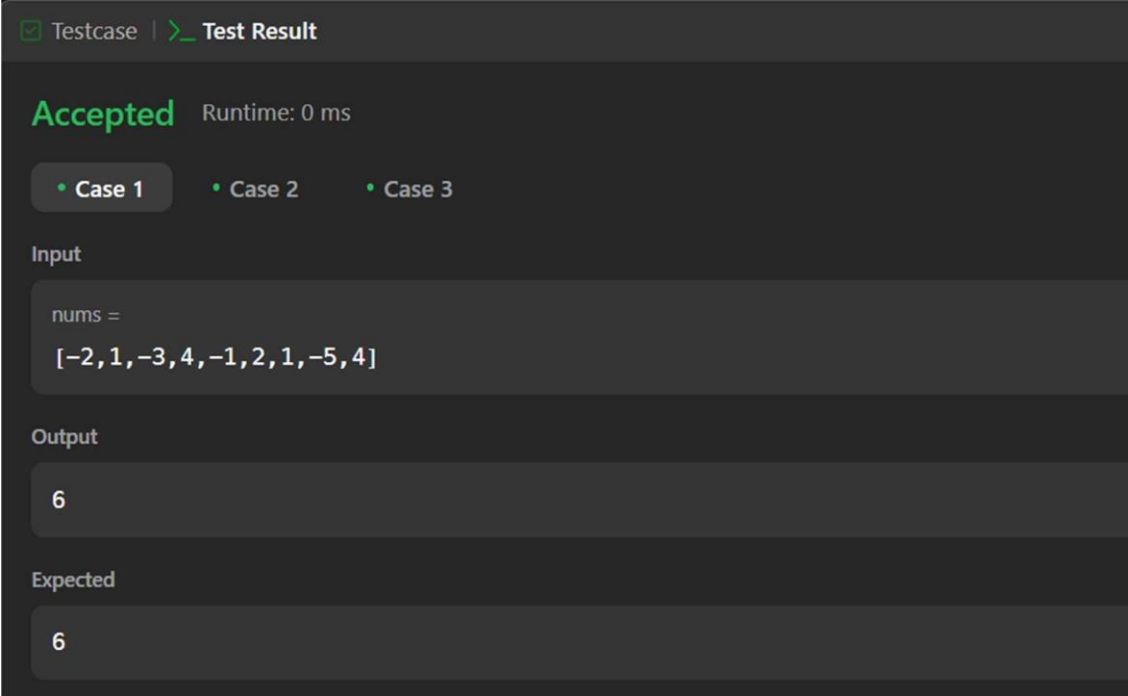
        for (int num : nums) {

            if (currsum < 0) {
                currsum = 0;
            }

            currsum += num;
```

```
        maxsum = max(maxsum, currsum);  
    }  
  
    return maxsum;  
}  
};
```

Output:



The screenshot shows a test result interface with a dark theme. At the top, there are tabs for 'Testcase' and 'Test Result', with 'Test Result' being the active tab. Below the tabs, the status 'Accepted' is displayed in green, followed by 'Runtime: 0 ms'. There are three tabs for test cases: 'Case 1', 'Case 2', and 'Case 3', with 'Case 1' being the active tab. Under the 'Case 1' tab, there are three sections: 'Input', 'Output', and 'Expected'. The 'Input' section shows 'nums = [-2, 1, -3, 4, -1, 2, 1, -5, 4]'. The 'Output' section shows the value '6'. The 'Expected' section also shows the value '6'.

Problem 5. Write an efficient algorithm that searches for a value target in an $m \times n$ integer matrix. This matrix has the following properties:

- Integers in each row are sorted in ascending from left to right.
- Integers in each column are sorted in ascending from top to bottom.

Algorithm:

1. Define the Search Space
 - Treat the matrix as a flattened 1D array of size $m * n$.
 - Define search boundaries:
 - left = 0 (first element)
 - right = $(m * n) - 1$ (last element)
2. Perform Binary Search
 - Compute the middle index:
 - $mid = (left + right) / 2$
 - Convert this 1D index into a 2D index:
 - $row = mid / n$
 - $col = mid \% n$
 - Compare $matrix[row][col]$ with target:
 - If equal, return true.
 - If less than target, search the right half ($left = mid + 1$).

- If greater than target, search the left half (right = mid - 1).
3. Return false if target is not found.

Code:

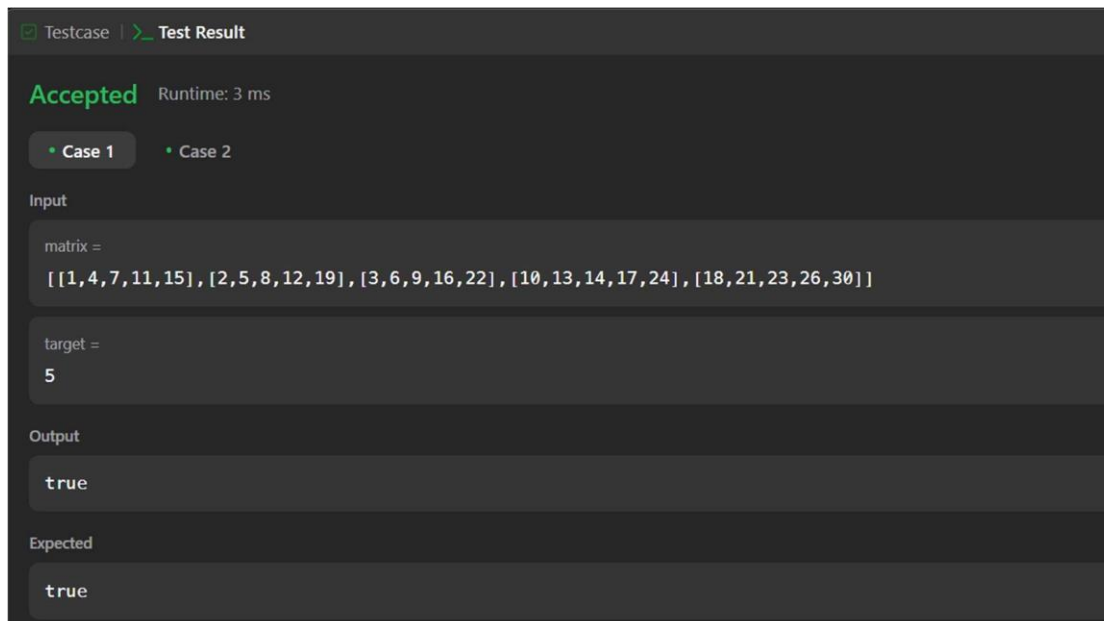
```
class Solution {
public:
    bool searchMatrix(vector<vector<int>>& matrix, int target) {
        if (matrix.empty() || matrix[0].empty()) return false;

        int rows = matrix.size();
        int cols = matrix[0].size();
        int left = 0, right = rows * cols - 1;

        while (left <= right) {
            int mid = left + (right - left) / 2;
            int row = mid / cols;
            int col = mid % cols;

            if (matrix[row][col] == target)
                return true;
            else if (matrix[row][col] < target)
                left = mid + 1;
            else
                right = mid - 1;
        }
        return false;
    }
};
```

Output:



The screenshot shows a code execution environment with a dark theme. At the top, there are tabs for 'Testcase' and 'Test Result'. Below the tabs, the word 'Accepted' is displayed in green, followed by 'Runtime: 3 ms'. There are two tabs for 'Case 1' and 'Case 2', with 'Case 1' being the active one. Under the 'Input' section, there are two text boxes: 'matrix =' containing a 5x5 array of integers, and 'target =' containing the integer 5. Under the 'Output' section, there is a text box containing the boolean value 'true'. At the bottom, under the 'Expected' section, there is a text box containing the boolean value 'true'.

```
Testcase | > Test Result

Accepted Runtime: 3 ms

• Case 1 • Case 2

Input

matrix =
[[1,4,7,11,15],[2,5,8,12,19],[3,6,9,16,22],[10,13,14,17,24],[18,21,23,26,30]]

target =
5

Output

true

Expected

true
```

Problem 6. Write an efficient algorithm that searches for a value target in an $m \times n$ integer matrix matrix. This matrix has the following properties:

- Integers in each row are sorted in ascending from left to right.
- Integers in each column are sorted in ascending from top to bottom.

Algorithm:

1. **Define a helper function modPow(a, exp, mod)** that computes:
(a^{exp} mod mod) (a^{exp} \mod mod) (aexpmodmod)
efficiently using **fast exponentiation**.
2. **Process the exponent list b using recursion:**
 - Extract the **last digit** from b (lastDigit = b.pop_back()).
 - Compute: part1= (modPow(a, lastDigit,1337))part1 = (modPow(a, lastDigit, 1337))part1=(modPow(a,lastDigit,1337)) part2=(modPow(superPow(a,b),10,1337))part2 = (modPow(superPow(a, b), 10, 1337))part2=(modPow(superPow(a,b),10,1337))
 - Combine results: result=(part1×part2) mod 1337result = (part1 \times part2) \mod 1337result=(part1×part2) mod1337

Code:

```
class Solution {
public:
    const int MOD = 1337;
    int modPow(int a, int exp, int mod) {
        int result = 1;
        a %= mod;
        while (exp > 0) {
            if (exp % 2 == 1)
                result = (result * a) % mod;
            a = (a * a) % mod;
            exp /= 2;
        }
        return result;
    }

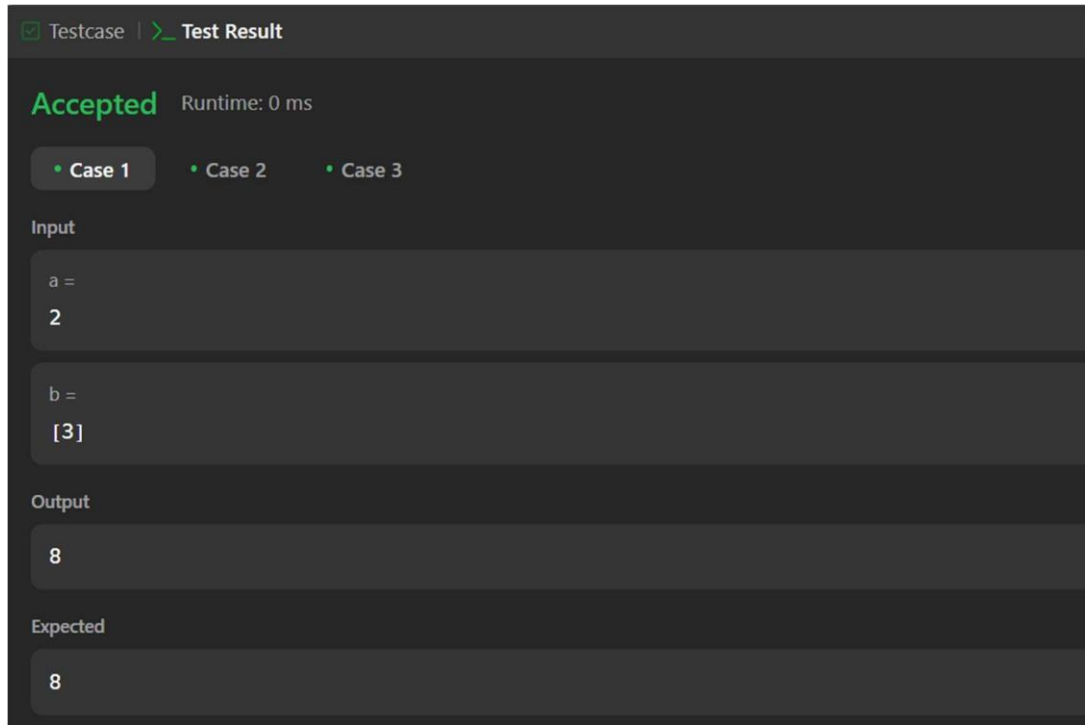
    int superPow(int a, vector<int>& b) {
        if (b.empty()) return 1;

        int lastDigit = b.back();
        b.pop_back();

        int part1 = modPow(a, lastDigit, MOD);
        int part2 = modPow(superPow(a, b), 10, MOD);

        return (part1 * part2) % MOD;
    }
};
```

Output:



Problem 7. Given the integer n , return *any beautiful array* *nums of length* n . There will be at least one valid answer for the given n .

Algorithm:

1. Start with **base case**: $n = 1 \rightarrow \{1\}$.
2. Recursively generate two sequences:
 - **Odd values**: $2 * x - 1$ (from `beautifulArray(n/2)`)
 - **Even values**: $2 * x$ (from `beautifulArray(n/2)`)
3. Combine the two sequences to maintain the required properties.

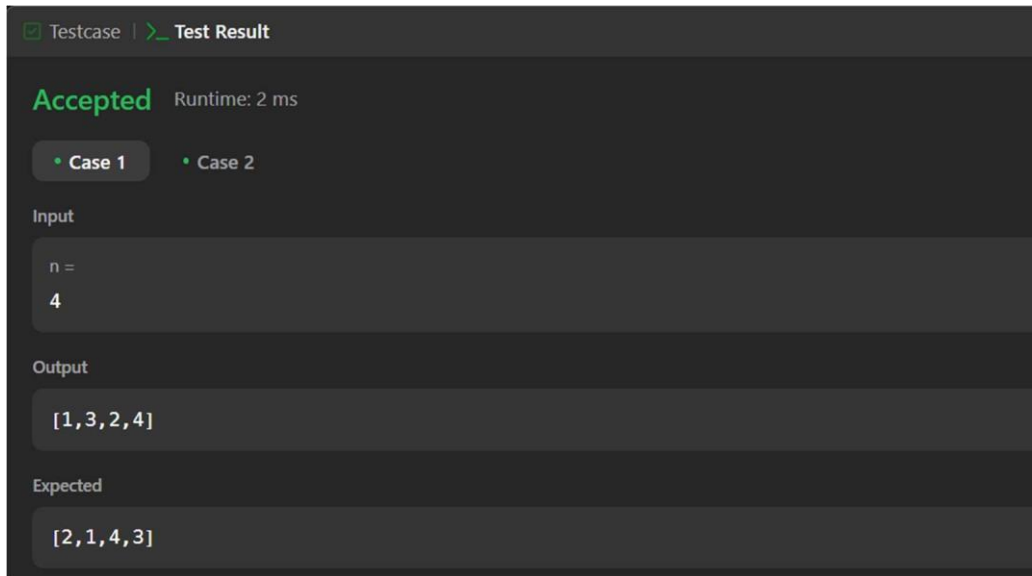
Code:

```
class Solution {
public:
    vector<int> beautifulArray(int n) {
        vector<int> result = {1};
        while (result.size() < n) {
            vector<int> temp;
            for (int num : result) {
                if (2 * num - 1 <= n)
                    temp.push_back(2 * num - 1);
            }
            for (int num : result) {
                if (2 * num <= n)
                    temp.push_back(2 * num);
            }
        }
    }
};
```



```
        result = temp;
    }
    return result;
}
};
```

Output:



Problem 8. 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](#)).

Algorithm:

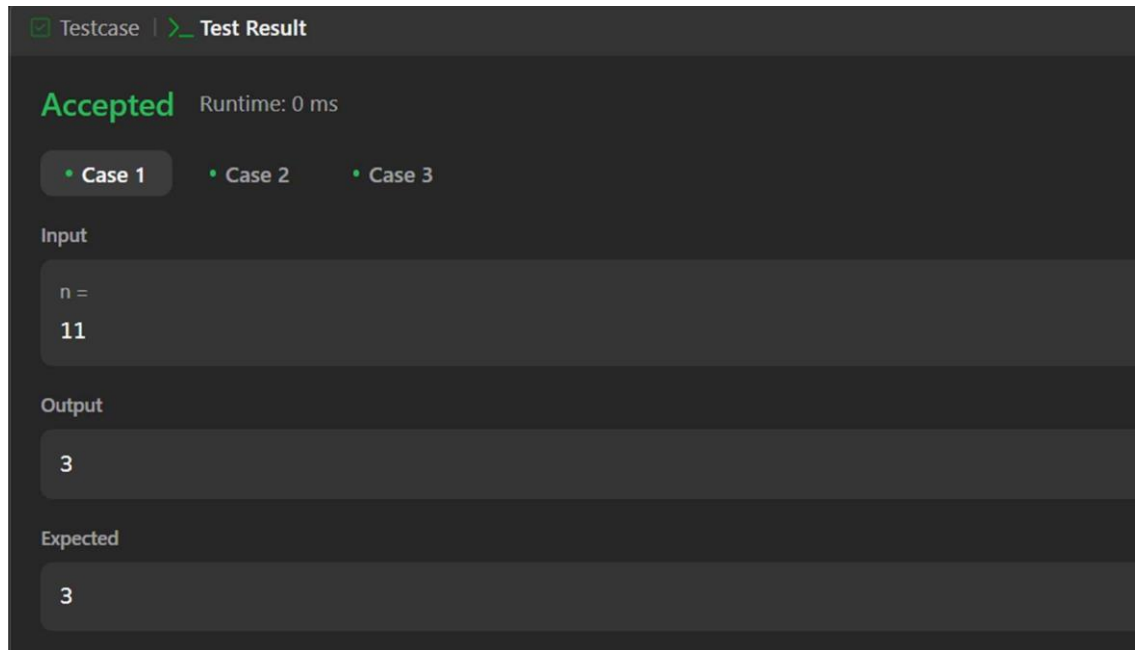
Steps

5. Initialize $\text{count} = 0$ to store the number of 1 bits.
6. Iterate through all 32 bits of the integer:
 - o Check if the last bit is 1 using $(n \& 1)$.
 - o If true, increment count.
 - o Right shift n ($n = n \gg 1$) to check the next bit.
7. Return count after processing all bits.

Code:

```
class Solution {
public:
    int hammingWeight(int n) {
        int count = 0;
        while (n != 0) {
            count += (n & 1);
            n >>= 1;
        }
        return count;
    }
};
```

Output:



Problem 9. Given an integer array `nums`, find the subarray with the largest sum, and return *its sum*.

Algorithm:

1. Initialize Variables

- `maxsum = nums[0]` → Stores the maximum subarray sum found so far.
- `currsum = 0` → Tracks the sum of the current subarray.

2. Iterate Through the Array

- If `currsum` becomes negative, reset it to 0 (since a negative sum reduces the potential maximum).
- Add the current element `num` to `currsum`.
- Update `maxsum = max(maxsum, currsum)`.

8. **Return `maxsum`**, which stores the maximum subarray sum.

Code:

```
class Solution {
public:
    int maxSubArray(vector<int>& nums) {
        int maxsum = nums[0];
        int currsum = 0;

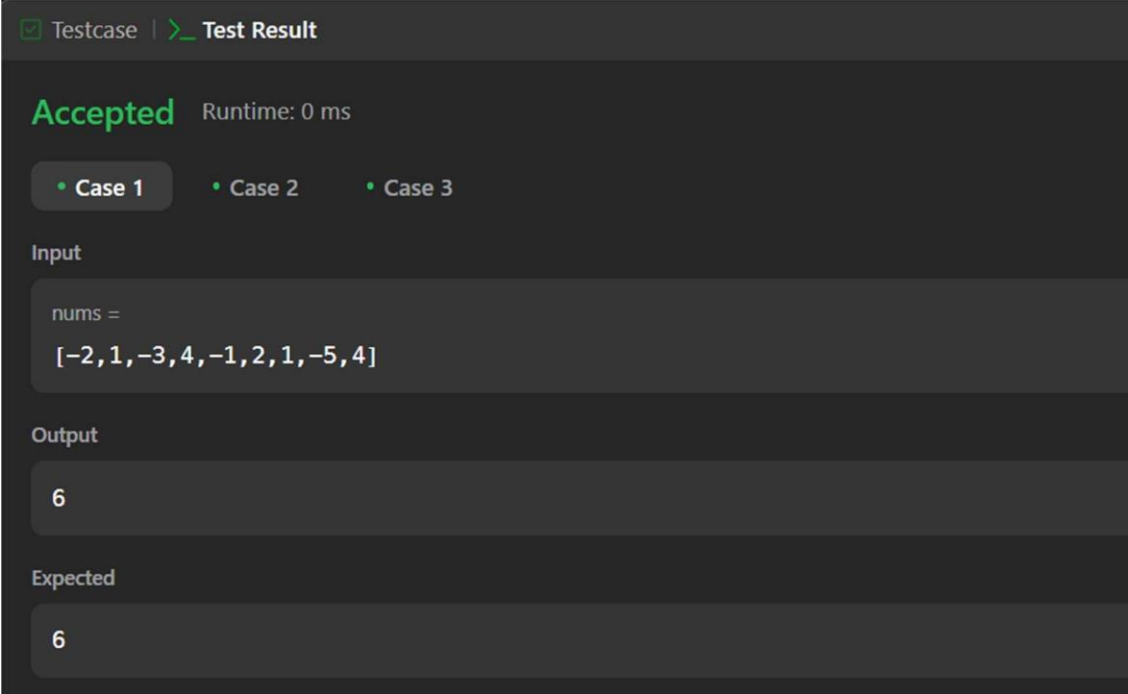
        for (int num : nums) {

            if (currsum < 0) {
                currsum = 0;
            }

            currsum += num;
```

```
        maxsum = max(maxsum, currsum);  
    }  
  
    return maxsum;  
}  
};
```

Output:



The screenshot shows a web-based code execution environment. At the top, there are tabs for 'Testcase' and 'Test Result', with 'Test Result' being the active tab. Below the tabs, the word 'Accepted' is displayed in green, followed by 'Runtime: 0 ms'. There are three buttons labeled 'Case 1', 'Case 2', and 'Case 3', with 'Case 1' being the selected one. Under the 'Input' section, the code 'nums = [-2, 1, -3, 4, -1, 2, 1, -5, 4]' is shown. Under the 'Output' section, the value '6' is displayed. Under the 'Expected' section, the value '6' is also displayed, indicating a successful match.

Problem 10. Write an efficient algorithm that searches for a value target in an $m \times n$ integer matrix. This matrix has the following properties:

- Integers in each row are sorted in ascending from left to right.
- Integers in each column are sorted in ascending from top to bottom.

Algorithm:

1. Define the Search Space
 - Treat the matrix as a flattened 1D array of size $m * n$.
 - Define search boundaries:
 - left = 0 (first element)
 - right = $(m * n) - 1$ (last element)
2. Perform Binary Search
 - Compute the middle index:
 - mid = $(\text{left} + \text{right}) / 2$
 - Convert this 1D index into a 2D index:
 - row = mid / n

Discover. Learn. Empower.

- $col = mid \% n$
- Compare `matrix[row][col]` with target:
 - If equal, return true.
 - If less than target, search the right half ($left = mid + 1$).
 - If greater than target, search the left half ($right = mid - 1$).
- 3. Return false if target is not found.

Code:

```
class Solution {
public:

    bool searchMatrix(vector<vector<int>>& matrix, int target) {
        if (matrix.empty() || matrix[0].empty()) return false;

        int rows = matrix.size();
        int cols = matrix[0].size();
        int left = 0, right = rows * cols - 1;

        while (left <= right) {
            int mid = left + (right - left) / 2;
            int row = mid / cols;
            int col = mid % cols;

            if (matrix[row][col] == target)
                return true;
            else if (matrix[row][col] < target)
                left = mid + 1;
            else
                right = mid - 1;
        }
        return false;
    }
};
```



DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

Discover. Learn. Empower.

Output:

☒ Testcase

[Test Result](#)

Accepted Runtime: 3 ms

Case 1

Case 2

Input

matrix =
[[1,4,7,11,15],[2,5,8,12,19],[3,6,9,16,22],[10,13,14,17,24],[18,21,23,26,30]]

target =
5

Output

true

Expected

true