Experiment 4

Student Name: Diya Parmar
UID: 22BET10210
Branch: IT
Section/Group: 22BET_IOT-702/A
Semester: 6th
Date of Performance: 14/02/25
Subject Name: Advance Programming-II
Subject Code: 22ITP-367

Problem: 1.4.1: Longest Nice Substring

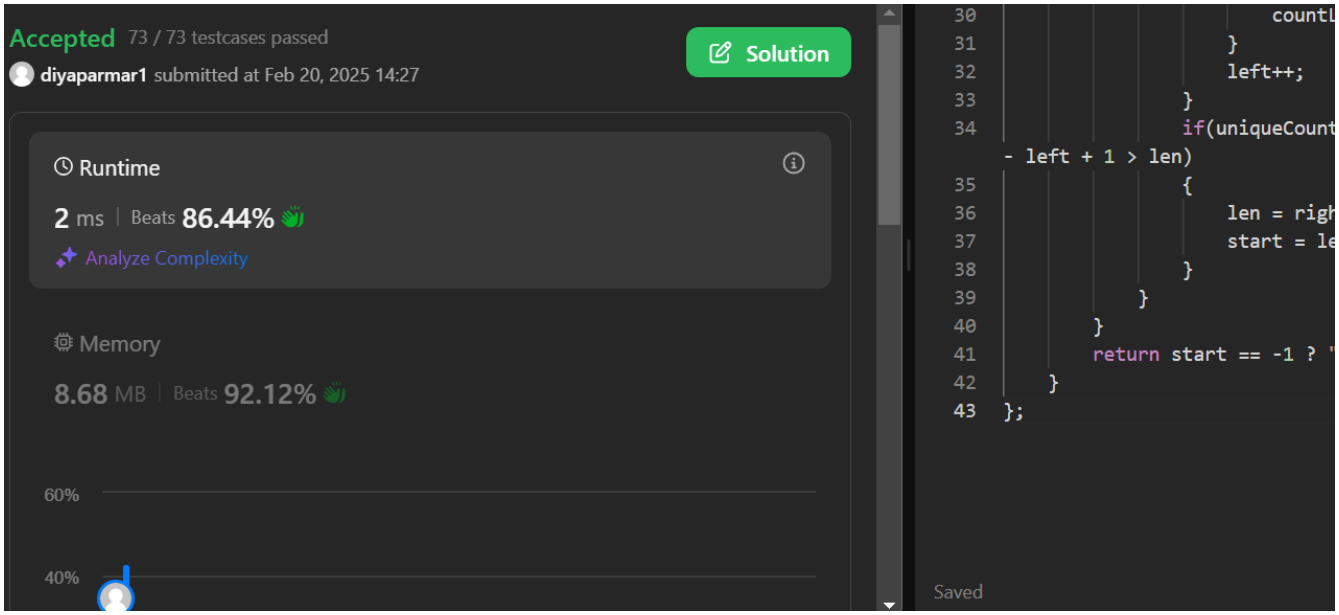
Problem Statement: A string s is considered **nice** if, for every character in the string, the character's uppercase and lowercase forms both exist in the string.

1. **Objective:** Find the longest contiguous substring where every character has both its uppercase and lowercase counterpart present.

2. **Code:**

```
class Solution:
def longestNiceSubstring(self, s: str) -> str:
    # Base case: if the string is empty or has only one character, return ""
    if len(s) < 2:
        return ""

    # Check for invalid characters
    for i, ch in enumerate(s):
        if ch.swapcase() not in s:
            # Split around the invalid character and check both parts
            left = self.longestNiceSubstring(s[:i])
            right = self.longestNiceSubstring(s[i+1:])
            # Return the longer substring
            return left if len(left) >= len(right) else right

    # If all characters are valid, return the entire string
    return s
```

3. Result:


Accepted 73 / 73 testcases passed
diyaparmar1 submitted at Feb 20, 2025 14:27
Solution
Runtime
2 ms | Beats 86.44%
Analyze Complexity
Memory
8.68 MB | Beats 92.12%
60%
40%
countL
}
left++;
}
if(uniqueCount
- left + 1 > len)
{
len = righ
start = le
}
}
}
return start == -1 ?
}
};
Saved

Problem 1.4.2: Reverse Bits

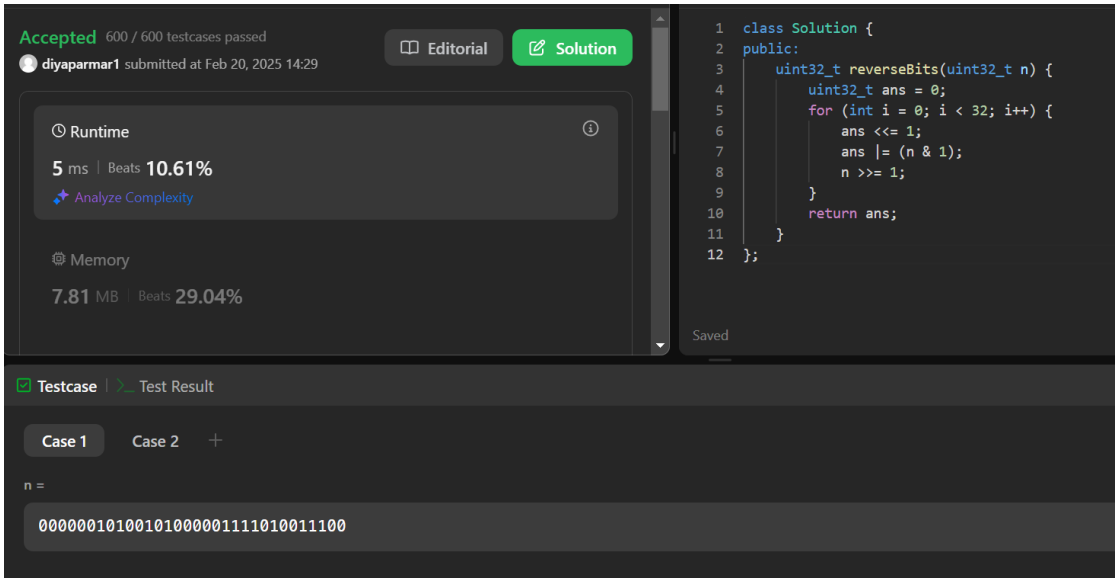
Problem Statement: You are given a 32-bit unsigned integer n. Your task is to reverse the bits of n and return the result as an unsigned integer.

1. **Objective:** Reverse the order of bits in the given 32-bit unsigned integer.

2. **Code:**

```
class Solution {
public:
    uint32_t reverseBits(uint32_t n) {
        // Swap adjacent bits
        n = ((n >> 1) & 0x55555555) | ((n & 0x55555555) << 1);
        // Swap pairs of bits
        n = ((n >> 2) & 0x33333333) | ((n & 0x33333333) << 2);
        // Swap nibbles (4-bit groups)
        n = ((n >> 4) & 0x0F0F0F0F) | ((n & 0x0F0F0F0F) << 4);
        // Swap bytes
        n = ((n >> 8) & 0x00FF00FF) | ((n & 0x00FF00FF) << 8);
        // Swap 16-bit halves
        n = (n >> 16) | (n << 16);
        return n;
    }
};
```

3. **Result:**



Problem 1.4.3: Number of 1 bits

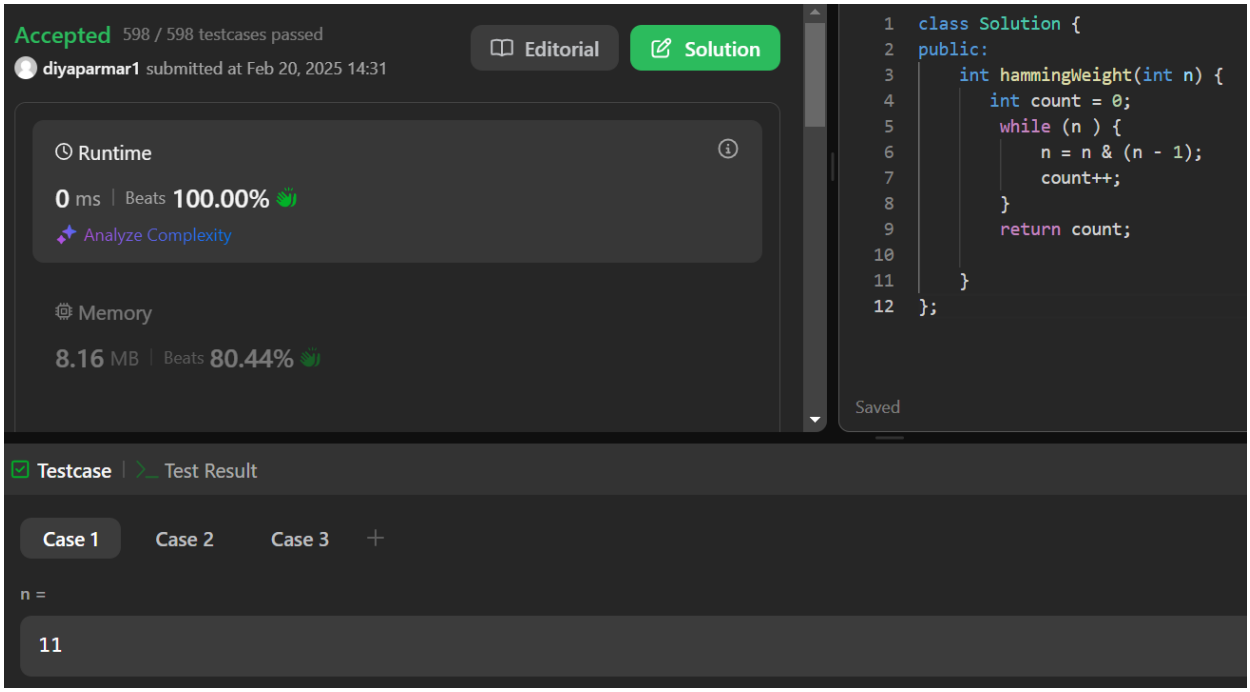
Problem Statement: You are given a 32-bit unsigned integer n. Your task is to return the number of '1' bits it has, also known as the **Hamming Weight**.

1. **Objective:** Count the number of '1' bits in the 32-bit binary representation of n.
2. **Code:**

```
class Solution {
public:
    int hammingWeight(uint32_t n) {
        int count = 0;
        while (n) {
            n &= (n - 1);  // Remove the lowest set bit
```

```
            count++;
        }
        return count;
    }
};
```

3. **Result:**



Problem 1.3.4: Max Subarray

Problem Statement: Given an integer array nums, find the contiguous subarray (containing at least one number) which has the largest sum and return its sum.

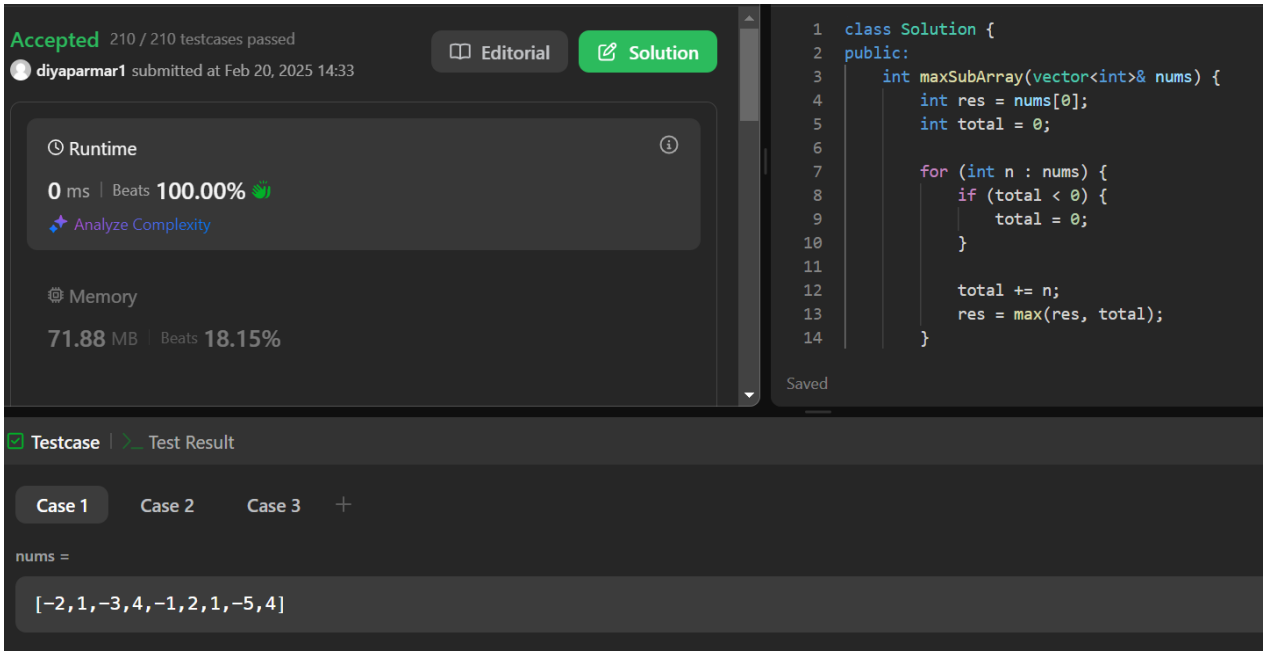
1. **Objective:** Identify the contiguous subarray within the given array that has the maximum sum.

2. **Code:**

```
class Solution {
public:
    int maxSubArray(vector<int>& nums) {
        int maxSum = nums[0];
        int currentSum = nums[0];
        for (int i = 1; i < nums.size(); i++)
```

```
currentSum = max(nums[i], currentSum + nums[i]); maxSum =
max(maxSum, currentSum);
                }
                return maxSum;
            }
        };
```

3. Result:



Problem 1.4.5: The Skyline Problem

Problem Statement: Given a list of buildings, where each building is represented as a triplet [L,R,H][L, R, H][L,R,H] (with LLL as the left x-coordinate, RRR as the right x-coordinate, and HHH as the height), your task is to output the skyline
formed by these buildings. The skyline is a list of "key points" [x,y][x, y][x,y] that represent where the height of the skyline changes. Key points should be output in sorted order by the x-coordinate.

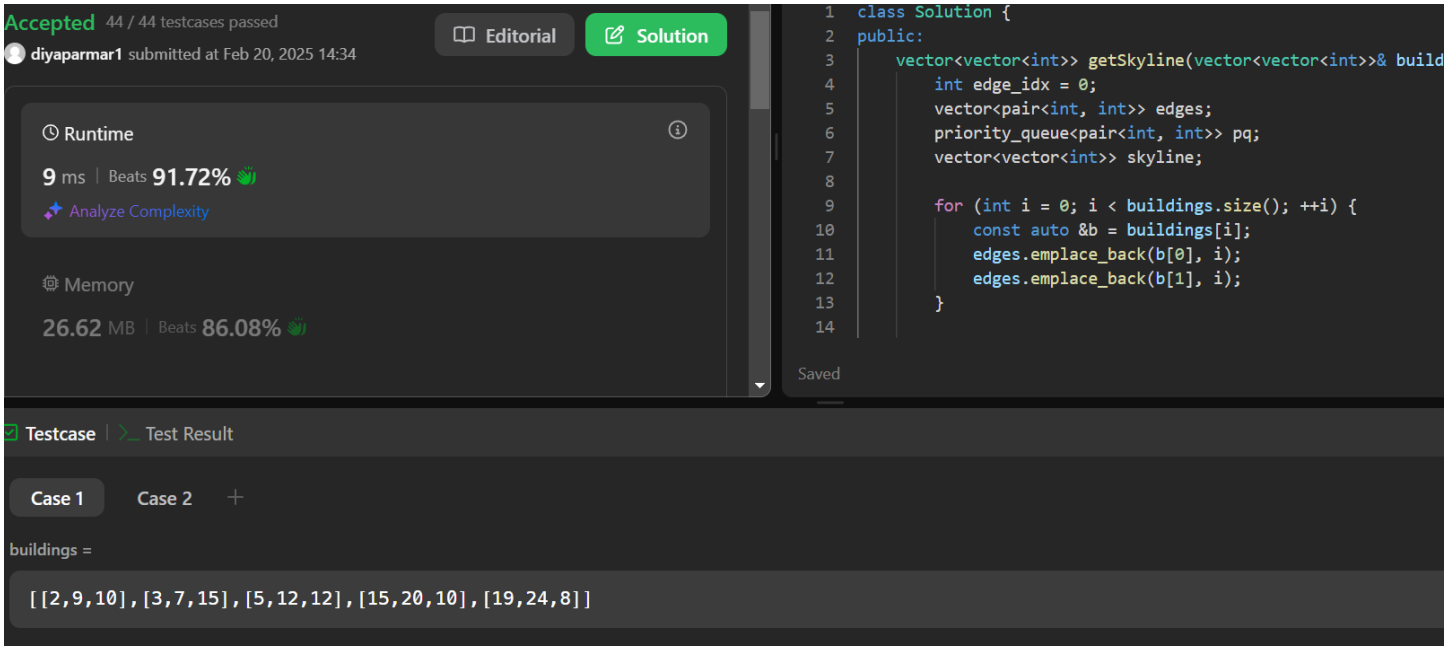
1. **Objective:** Determine the key points that form the outer contour (skyline) when the buildings are viewed from a distance.

2. **Code:**

```
#include <vector>
#include <queue>
#include <algorithm>
#include <climits>
using namespace std;

// Custom comparator: orders by first element ascending.
struct cmp {
    bool operator()(const pair<int,int>& a, const pair<int,int>& b) {
        return a.first > b.first; // smaller (more negative) first element has higher priority.
    }
};

class Solution {
public:
    vector<vector<int>> getSkyline(vector<vector<int>>& buildings) {
        // Create events: for each building [L, R, H]:
        // - Start event: (L, -H, R)
        // - End event: (R, 0, 0)
        vector<vector<int>> events;
        for (const auto& b : buildings) {
            int L = b[0], R = b[1], H = b[2];
            events.push_back({L, -H, R});
            events.push_back({R, 0, 0});
        }

        // Sort events by x-coordinate.
        // If two events share the same x, the one with smaller second value (i.e. start
events with higher heights) comes first.
        sort(events.begin(), events.end(), [](const vector<int>& a, const vector<int>& b)
{
            if(a[0] != b[0])
```

```
            return a[0] < b[0];
        return a[1] < b[1];
    });

    // Priority queue (min-heap using custom comparator) to track active buildings.
    // Each element is a pair (height, right), where height is stored as a negative
value.
    priority_queue<pair<int,int>, vector<pair<int,int>>, cmp> live;
    // Add a dummy building with height 0 lasting indefinitely.
    live.push({0, INT_MAX});

    vector<vector<int>> result;
    int i = 0, n = events.size();
    while (i < n) {
        int x = events[i][0];
        // Process all events at the same x-coordinate.
        while (i < n && events[i][0] == x) {
            if (events[i][1] < 0) {  // start event
                live.push({events[i][1], events[i][2]});
            }
            // End events are implicitly handled by removing expired buildings.
            i++;
        }

        // Remove buildings from the heap that have ended.
        while (!live.empty() && live.top().second <= x)
            live.pop();

        // The current skyline height is the negative of the top element's first value.
        int currHeight = -live.top().first;
        // If the height has changed, record a new key point.
        if (result.empty() || result.back()[1] != currHeight)
            result.push_back({x, currHeight});
    }

    return result;
  }
};
```

3. **Result:**


Accepted 44 / 44 testcases passed
diyaparmar1 submitted at Feb 20, 2025 14:34
Editorial
Solution
Runtime
9 ms | Beats 91.72%
Analyze Complexity
Memory
26.62 MB | Beats 86.08%
class Solution {
public:
vector<vector<int>> getSkyline(vector<vector<int>>& build
int edge_idx = 0;
vector<pair<int, int>> edges;
priority_queue<pair<int, int>> pq;
vector<vector<int>> skyline;
for (int i = 0; i < buildings.size(); ++i) {
const auto &b = buildings[i];
edges.emplace_back(b[0], i);
edges.emplace_back(b[1], i);
}
Saved
Testcase
Test Result
Case 1
Case 2
buildings =
[[2,9,10],[3,7,15],[5,12,12],[15,20,10],[19,24,8]]