**Name – Ansh Kumar Singh UID – 22BCS11057 Subject – AP LAB**

**1.Longest Nice Substring.**

class Solution {

public:

    string longestNiceSubstring(string s) {

        int n=s.length();

        if (s.length()<2) {

            return "";

        }

        bool lower[26]={false};

        bool upper[26]={false};

        for(char c:s){

            if(islower(c)){

                lower[c-'a']=true;

            }

            else{

                upper[c-'A']=true;

            }

        }

        for(int i=0;i<n;i++){

            char c=s[i];

            if(islower(c)&&!upper[c-'a']){

                string left=longestNiceSubstring(s.substr(0,i));

                string right=longestNiceSubstring(s.substr(i+1));

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

            }

            if(isupper(c)&&!lower[c-'A']){

                string left=longestNiceSubstring(s.substr(0,i));

                string right=longestNiceSubstring(s.substr(i+1));

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

            }

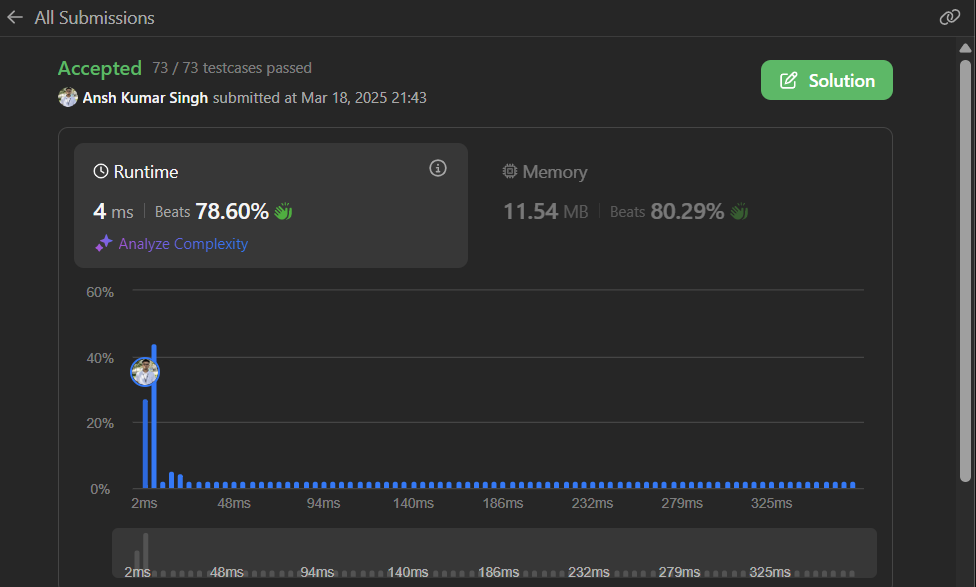
        }

        return s;

    }

};

**OUTPUT:-**

****

**2.Reverse Bits:**

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); // Shift result left and add the last bit of n

n >>= 1; // Shift n right to process the next bit

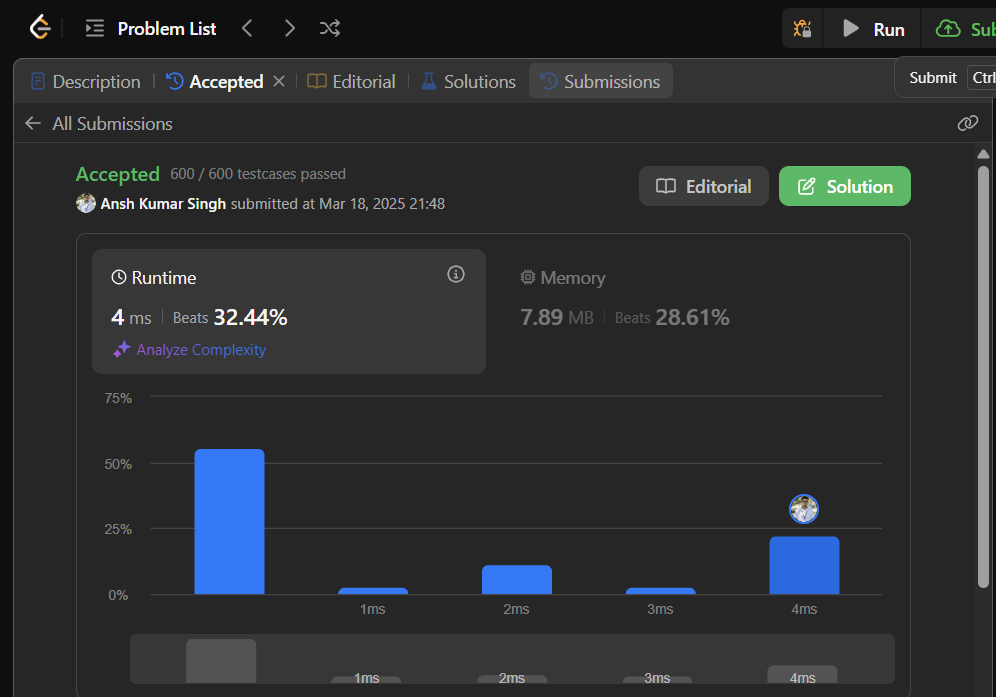
}

return result;

}

};

**OUTPUT:-**

****

**3.Number of 1 Bits**

class Solution {

public:

int hammingWeight(int n) {

int count = 0;

while (n) {

count += (n & 1); // Add 1 if the last bit is 1

n >>= 1; // Right shift n to check the next bit

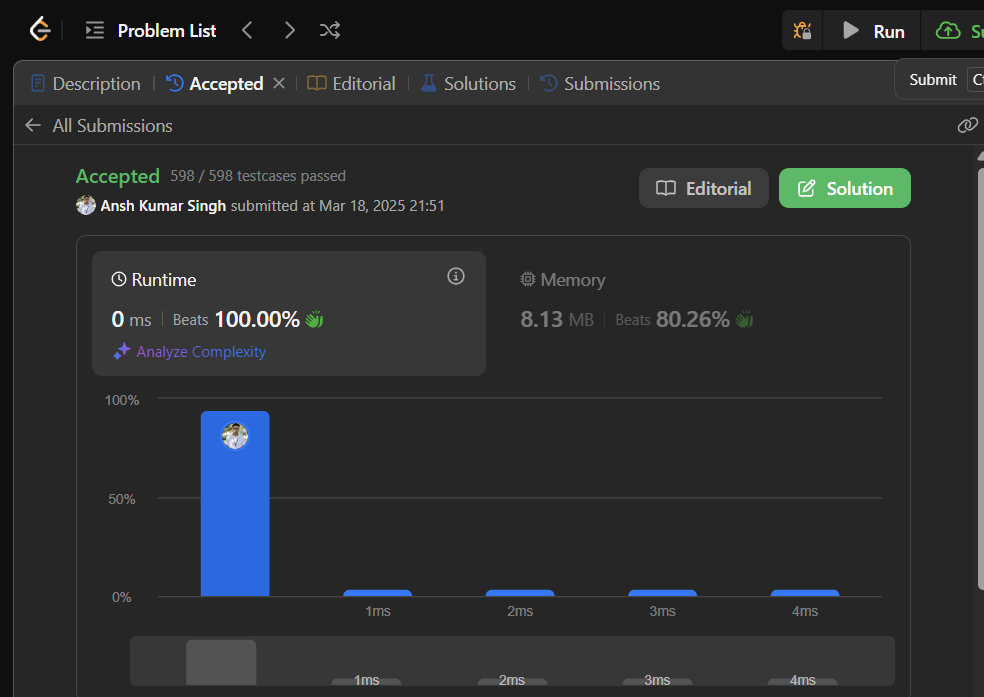
}

return count;

}

};

**OUTPUT:-**

****

**4.Maximum Subarray:**

class Solution {

public:

int maxSubArray(vector<int>& nums) {

int current\_sum = 0;

int max\_sum = nums[0]; // Initialize max\_sum with the first element

for (int num : nums) {

current\_sum = max(num, current\_sum + num); // Take the maximum of starting new subarray or extending current subarray

max\_sum = max(max\_sum, current\_sum); // Update max\_sum if current\_sum is greater

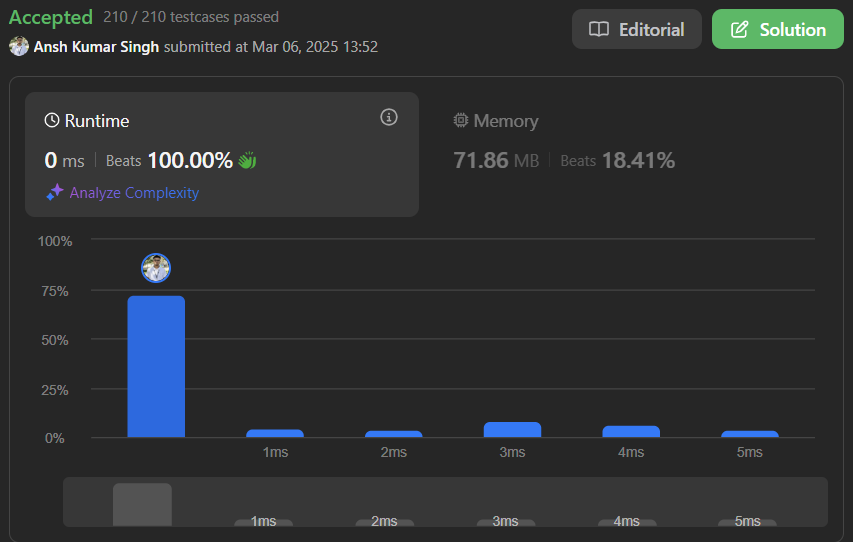
}

return max\_sum; // Return the maximum subarray sum found

}

};

**OUTPUT:-**

****

**5.Search a 2D Matrix II**

class Solution {

public:

bool searchMatrix(vector<vector<int>>& matrix, int target) {

int rows = matrix.size();

if (rows == 0) return false;

int cols = matrix[0].size();

int row = 0, col = cols - 1; // Start from top-right corner

while (row < rows && col >= 0) {

if (matrix[row][col] == target)

return true;

else if (matrix[row][col] > target)

col--; // Move left

else

row++; // Move down

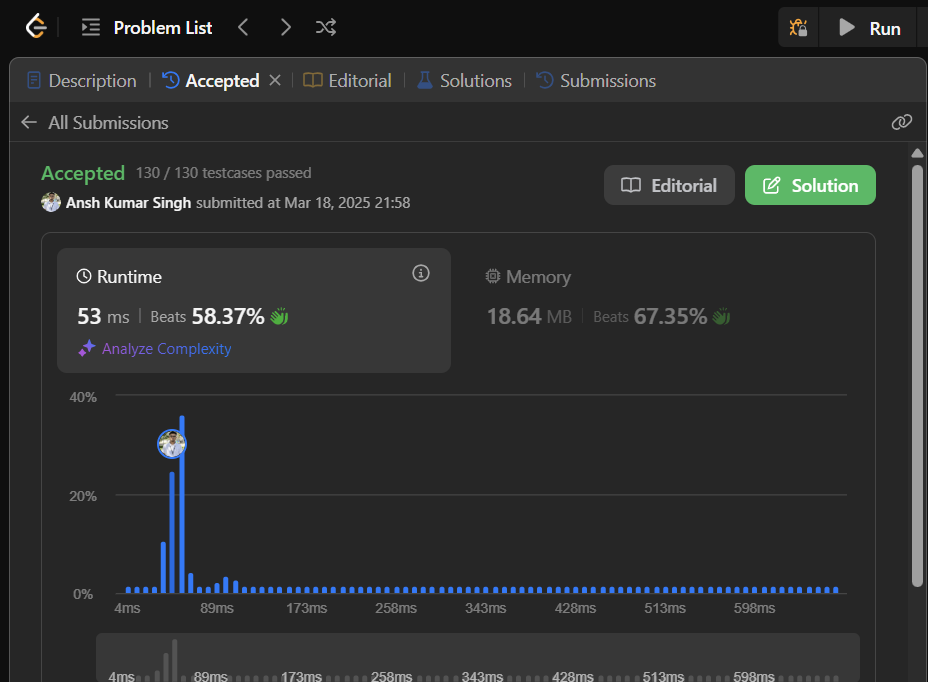
}

return false;

}

};

**OUTPUT:-**

****

**6.Super Pow:-**

class Solution {

public:

const int MOD = 1337;

int powmod(int x, int y) {

int result = 1;

x %= MOD; // Take mod at start to avoid large numbers

while (y) {

if (y % 2 == 1) // If y is odd, multiply by x

result = (result \* x) % MOD;

x = (x \* x) % MOD; // Square x and take mod

y /= 2; // Reduce y by half

}

return result;

}

int superPow(int a, vector<int>& b) {

int result = 1;

for (int digit : b) {

result = powmod(result, 10) \* powmod(a, digit) % MOD;

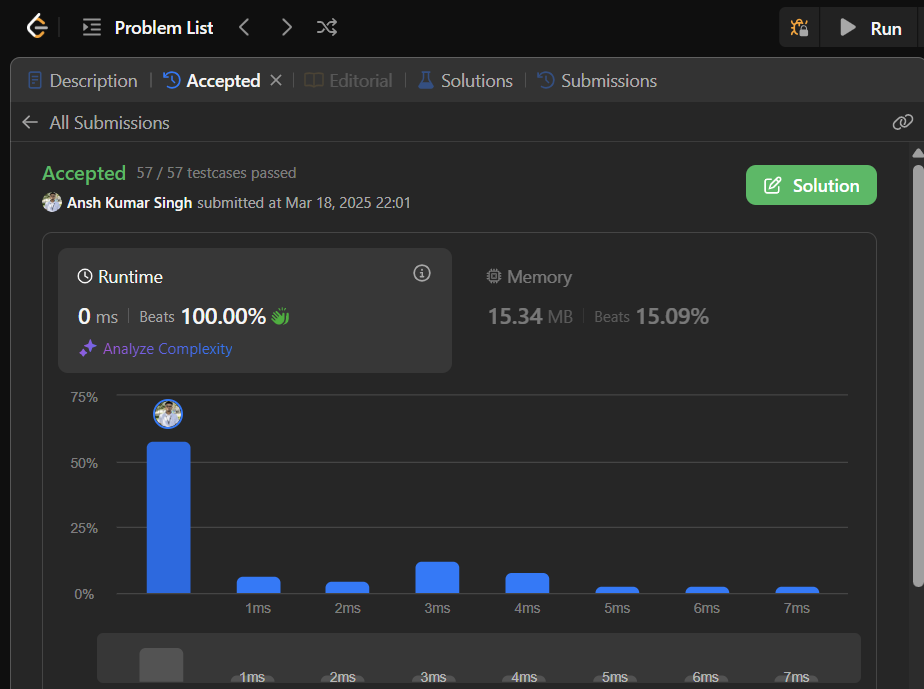
}

return result;

}

};

**OUTPUT:-**

****

**7.Beautiful Array:**

class Solution {

public:

vector<int> beautifulArray(int n) {

vector<int> result = {1}; // Start with base case

while (result.size() < n) {

vector<int> temp;

// Generate odd values

for (int num : result) {

if (num \* 2 - 1 <= n)

temp.push\_back(num \* 2 - 1);

}

for (int num : result) {

if (num \* 2 <= n)

temp.push\_back(num \* 2);

}

result = temp; // Update result

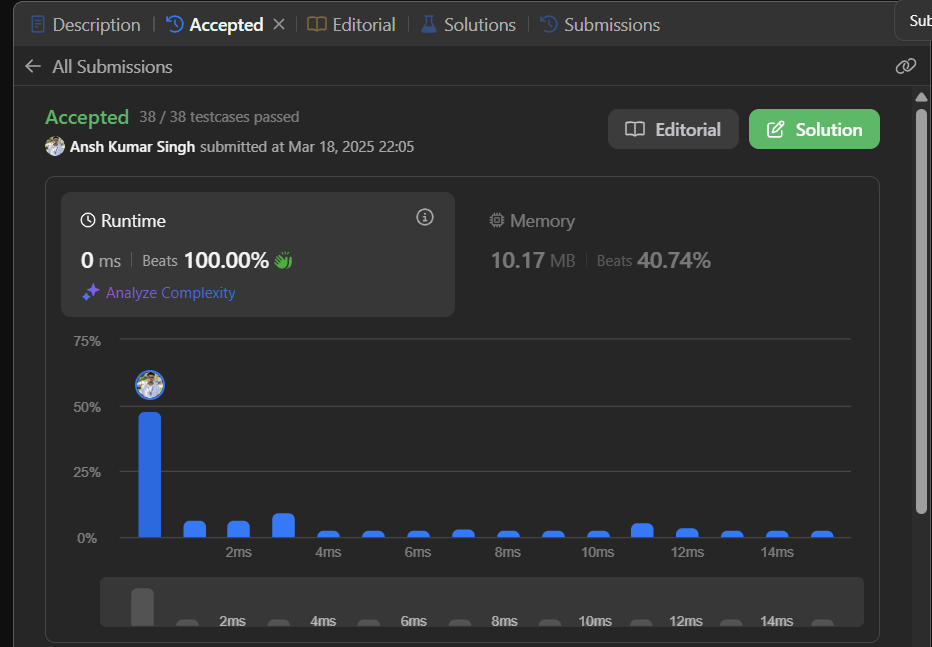
}

return result;

}

};

**OUTPUT:-**

****

**8.The Skyline Problem:**

class Solution {

public:

vector<vector<int>> getSkyline(vector<vector<int>>& buildings) {

vector<pair<int, int>> events;

for (const auto& b : buildings) {

events.emplace\_back(b[0], -b[2]); // Start event (negative height)

events.emplace\_back(b[1], b[2]); // End event (positive height)

}

sort(events.begin(), events.end());

multiset<int> heights = {0};

int prevMaxHeight = 0;

vector<vector<int>> skyline;

// Process events

for (const auto& e : events) {

int x = e.first, h = e.second;

if (h < 0) {

heights.insert(-h);

} else {

// End of a building: remove height

heights.erase(heights.find(h));

}

int currMaxHeight = \*heights.rbegin();

if (currMaxHeight != prevMaxHeight) {

skyline.push\_back({x, currMaxHeight});

prevMaxHeight = currMaxHeight;

}

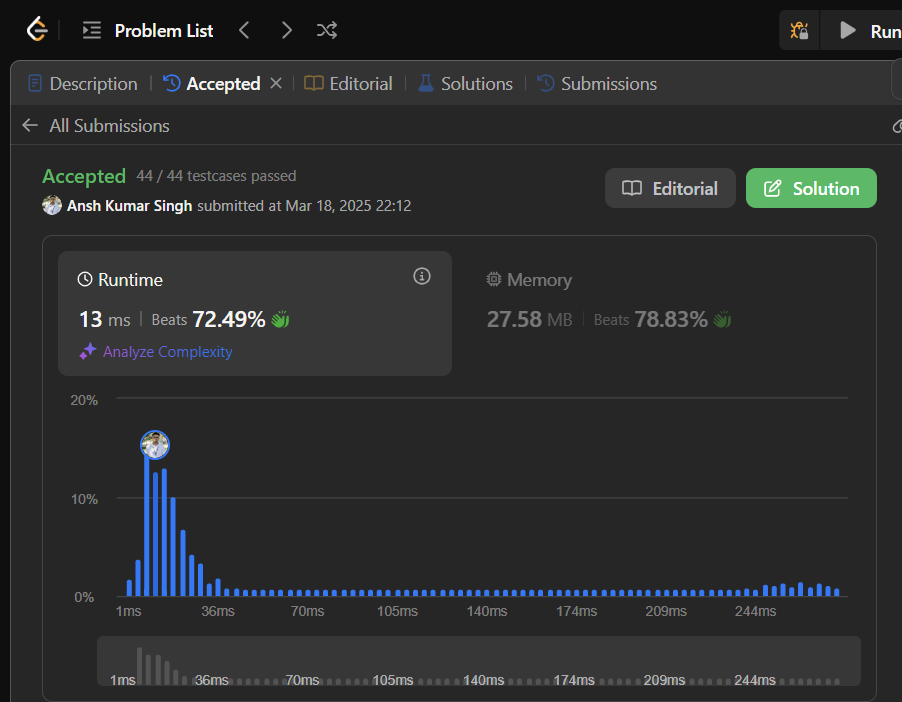
}

return skyline;

}

};

**OUTPUT:-**

****

**9.Reverse Pairs**

class Solution {

public:

    int reversePairs(vector<int>& nums) {

        return mergeSort(nums, 0, nums.size() - 1);

    }

private:

    int mergeSort(vector<int>& nums, int left, int right) {

        if (left >= right) return 0;

        int mid = left + (right - left) / 2;

        int count = mergeSort(nums, left, mid) + mergeSort(nums, mid + 1, right);

        int j = mid + 1;

        for (int i = left; i <= mid; i++) {

            while (j <= right && nums[i] > 2LL \* nums[j]) j++;

            count += (j - (mid + 1));

        }

        merge(nums, left, mid, right);

        return count;

    }

    void merge(vector<int>& nums, int left, int mid, int right) {

        vector<int> temp;

        int i = left, j = mid + 1;

        while (i <= mid && j <= right) {

            if (nums[i] <= nums[j]) temp.push\_back(nums[i++]);

            else temp.push\_back(nums[j++]);

        }

        while (i <= mid) temp.push\_back(nums[i++]);

        while (j <= right) temp.push\_back(nums[j++]);

        for (int k = left; k <= right; k++) {

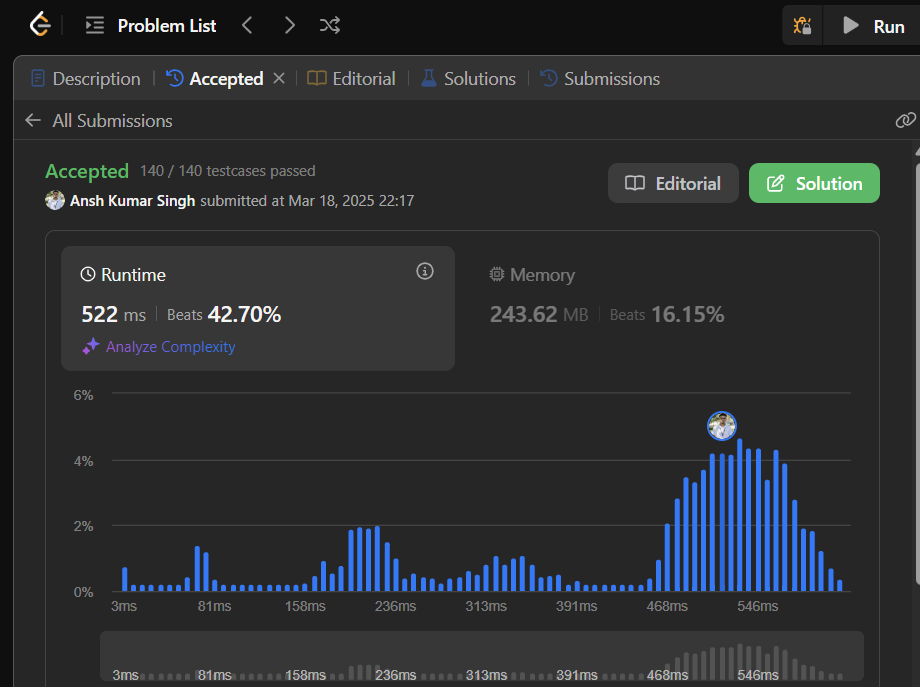
            nums[k] = temp[k - left];

        }

    }

};

**OUTPUT:-**

****

**10: Longest Increasing Subsequence II**

#include <vector>

#include <algorithm>

using namespace std;

class SegmentTree {

public:

vector<int> tree;

int size;

SegmentTree(int n) {

size = n;

tree.assign(4 \* n, 0);

}

int query(int node, int start, int end, int L, int R) {

if (R < start || end < L) return 0; // Out of range

if (L <= start && end <= R) return tree[node]; // Inside range

int mid = (start + end) / 2;

return max(query(2 \* node, start, mid, L, R),

query(2 \* node + 1, mid + 1, end, L, R));

}

void update(int node, int start, int end, int index, int value) {

if (start == end) {

tree[node] = value;

} else {

int mid = (start + end) / 2;

if (index <= mid) update(2 \* node, start, mid, index, value);

else update(2 \* node + 1, mid + 1, end, index, value);

tree[node] = max(tree[2 \* node], tree[2 \* node + 1]);

}

}

};

class Solution {

public:

int lengthOfLIS(vector<int>& nums, int k) {

int maxVal = \*max\_element(nums.begin(), nums.end());

SegmentTree segTree(maxVal + 1);

int maxLIS = 0;

for (int num : nums) {

int bestPrevLIS = segTree.query(1, 0, maxVal, max(0, num - k), num - 1);

int newLIS = bestPrevLIS + 1;

segTree.update(1, 0, maxVal, num, newLIS);

maxLIS = max(maxLIS, newLIS);

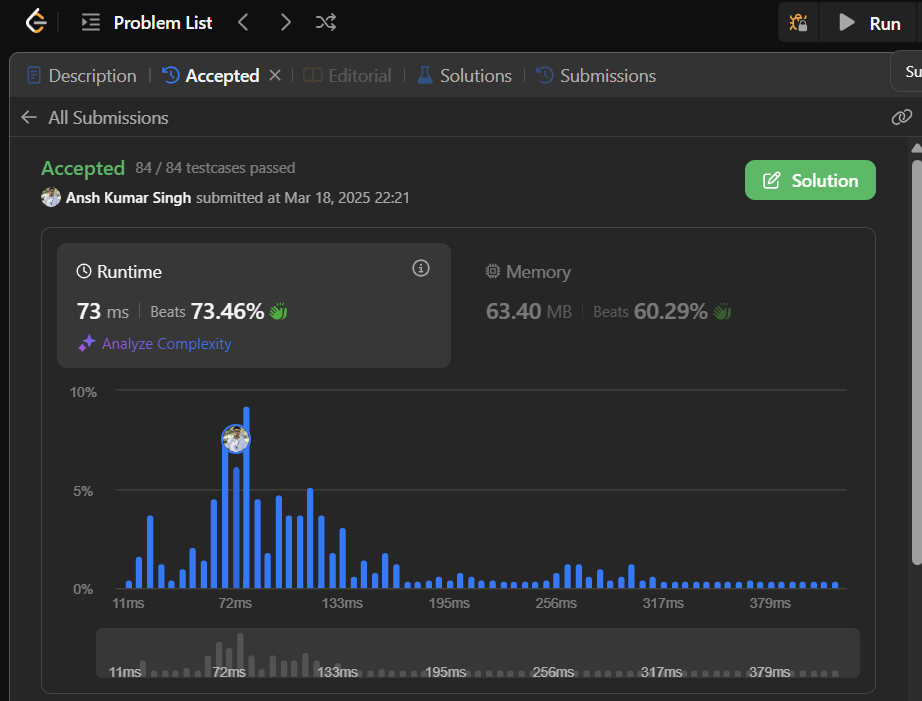
}

return maxLIS;

}

};

**OUTPUT:-**

****