Assignment 4

Name: Diksha Kumari UID: 22BCS16005

Section: 606 – B Subject: AP LAB

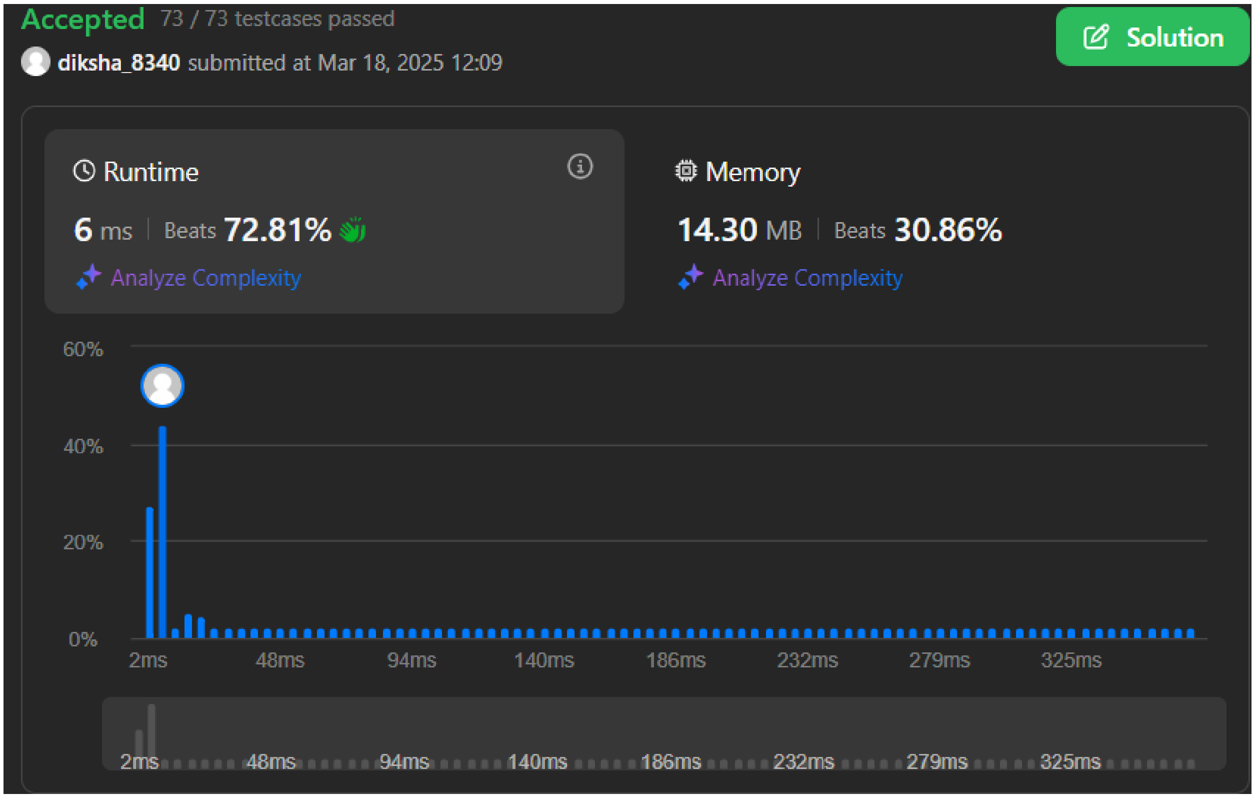
1. **Longest Nice Substring:** A string s is **nice** if, for every letter of the alphabet that s contains, it appears **both** in uppercase and lowercase. For example, "abABB" is nice because 'A' and 'a' appear, and 'B' and 'b' appear. However, "abA" is not because 'b' appears, but 'B' does not.

   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*.

CODE:

```
class Solution {
 public:
   string longestNiceSubstring(string s){
        set<char> st;
      for(char c : s){
         st.insert(c);
      }
      for(int i=0; i<s.length(); i++){
         if(st.count(tolower(s[i])) && st.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



Accepted 73 / 73 testcases passed

diksha_8340 submitted at Mar 18, 2025 12:09

Runtime

6 ms | Beats 72.81%

Memory

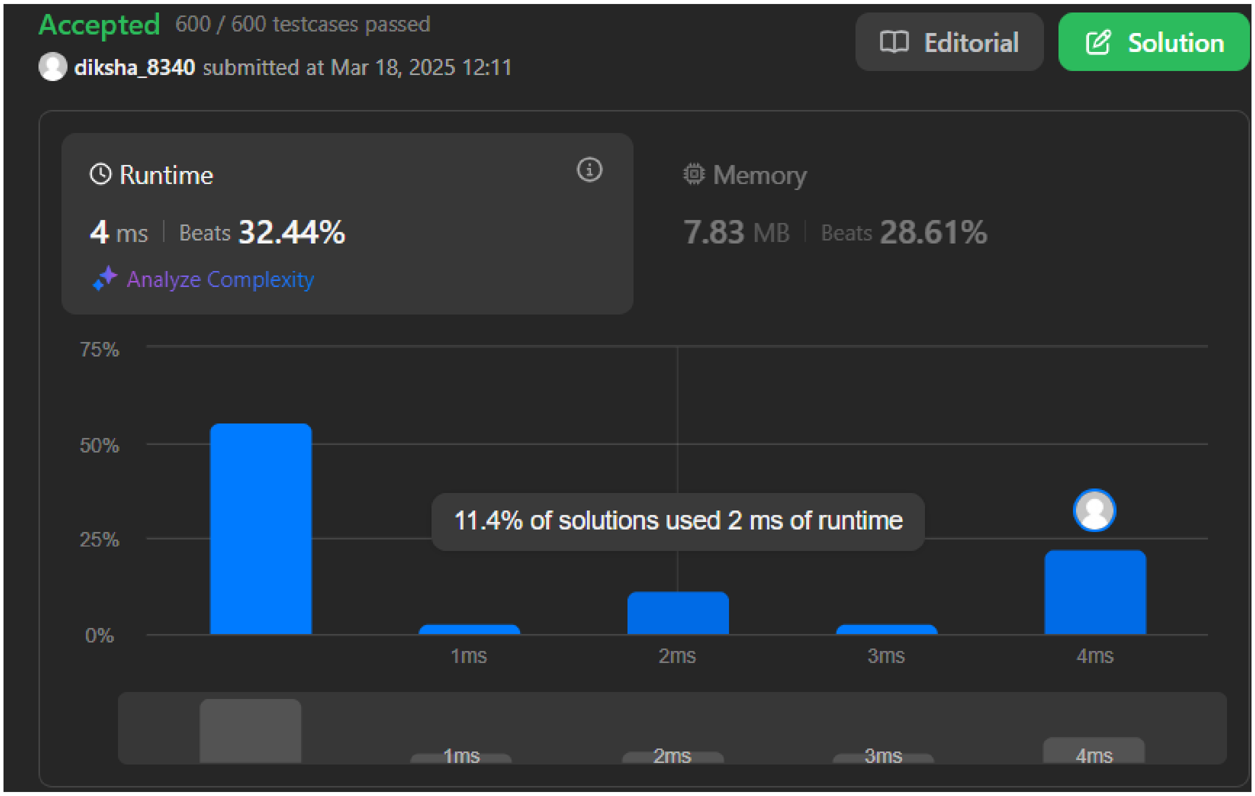
14.30 MB | Beats 30.86%

2. <u>Reverse Bits</u> : Reverse bits of a given 32 bits unsigned integer.

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:


Accepted 600 / 600 testcases passed

diksha_8340 submitted at Mar 18, 2025 12:11

Runtime

4 ms | Beats 32.44%

Memory

7.83 MB | Beats 28.61%

11.4% of solutions used 2 ms of runtime

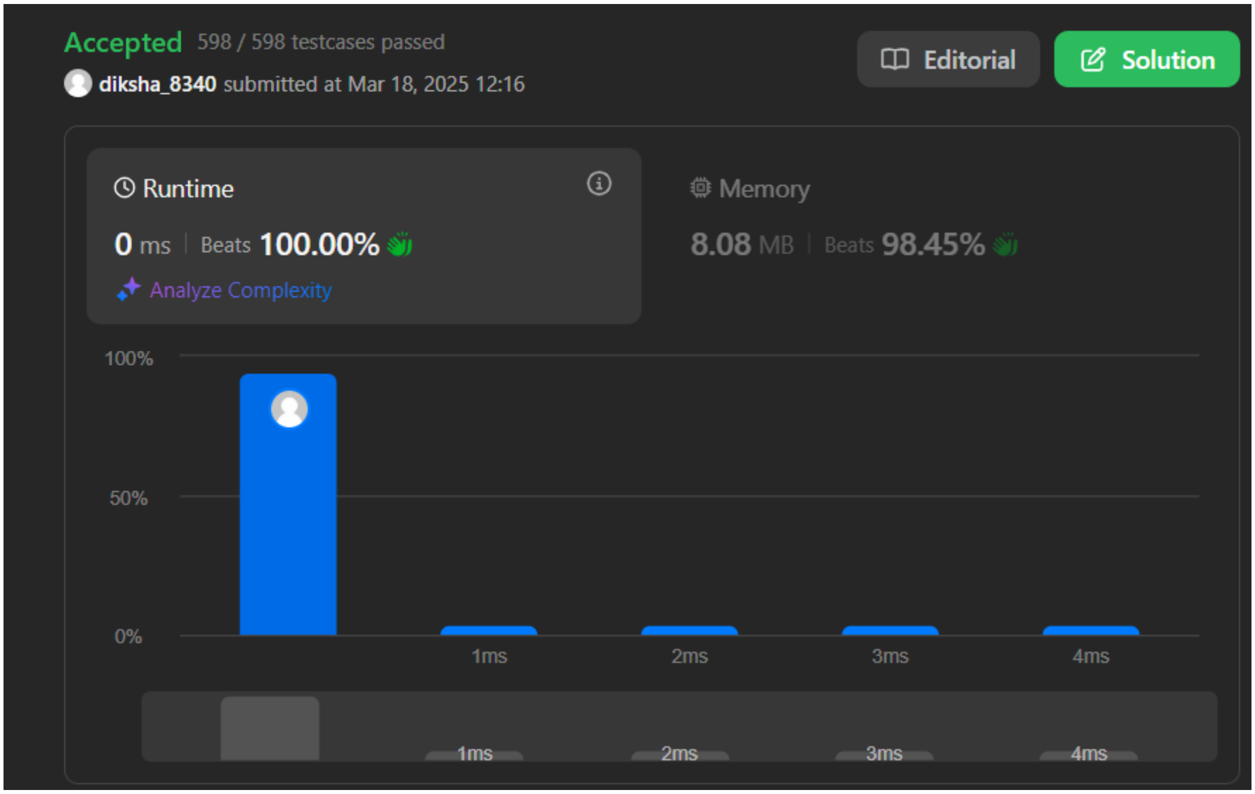
3. Number of 1 Bits:

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(int n) {
        int count = 0;
        while (n) {
            count += (n & 1);
            n >>= 1;
        }
        return count;
    }
};
```

OUTPUT



Accepted 598 / 598 testcases passed

diksha_8340 submitted at Mar 18, 2025 12:16

Runtime

0 ms | Beats 100.00%

Memory

8.08 MB | Beats 98.45%

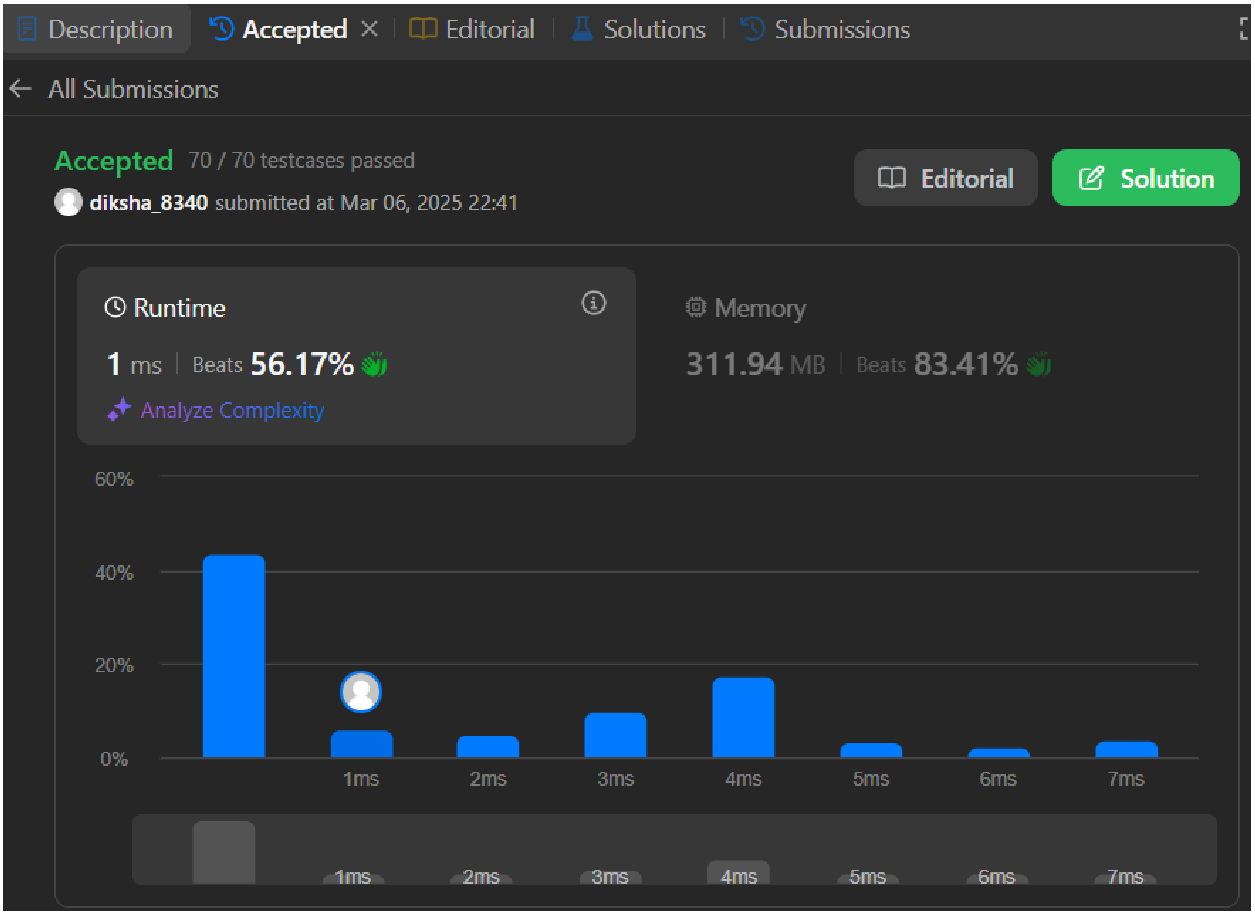
4. Maximum Subarray:

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

CODE:

```
class Solution {
public:
    int maxSubArray(vector<int>& nums) {
        int maxSum = INT_MIN;

        int sum =0;
        for(int i=0; i<nums.size(); i++){
            sum = sum + nums[i];
            if(sum > maxSum){
                maxSum=sum;
            }

            if(sum < 0){
                sum =0;
            }
        }
        return maxSum;
    }
};
```

OUTPUT:



Accepted 70 / 70 testcases passed

diksha_8340 submitted at Mar 06, 2025 22:41

Runtime

1 ms | Beats 56.17%

Memory

311.94 MB | Beats 83.41%

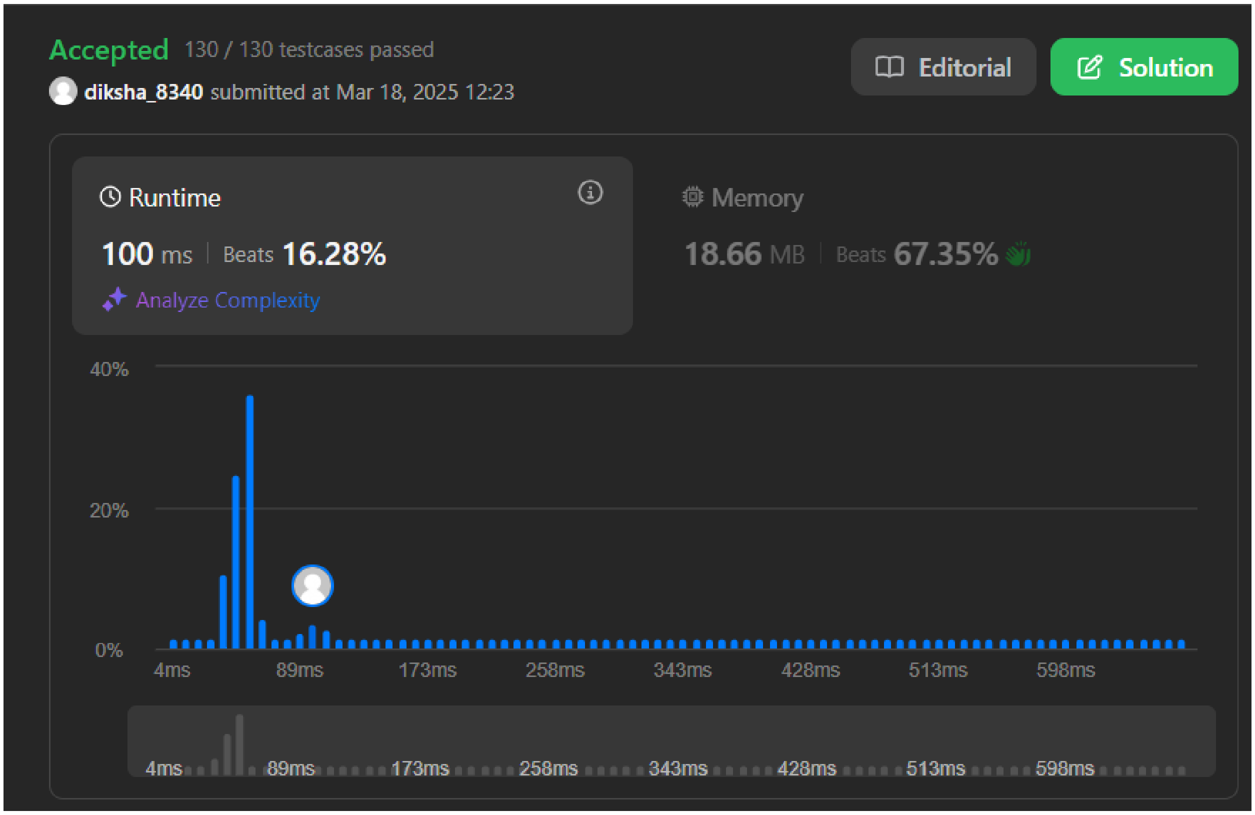
5. **Search a 2D Matrix II**

Write an efficient algorithm that searches for a value target in an m x n integer matrix matrix. This matrix has the following properties:

CODE:

```
class Solution {

public:

    bool binarysearch(vector<int>& rows, int target){

        int low = 0, high = rows.size()-1;

        while(low <= high){
            int mid = low + (high - low)/2;

            if(rows[mid] == target){

                return true;

            }

            else if(rows[mid] > target){

                high = mid - 1;

            }

            else{

                low = mid + 1;

            }

        }

        return false;

    }

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

        int row = matrix.size();

        for(int i=0; i<row; i++){

            if(binarysearch(matrix[i], target)){

                return true;

            }

        }

        return false;

    }

};

};
```

OUTPUT:



Accepted 130 / 130 testcases passed

diksha_8340 submitted at Mar 18, 2025 12:23

Runtime

100 ms | Beats 16.28%

Memory

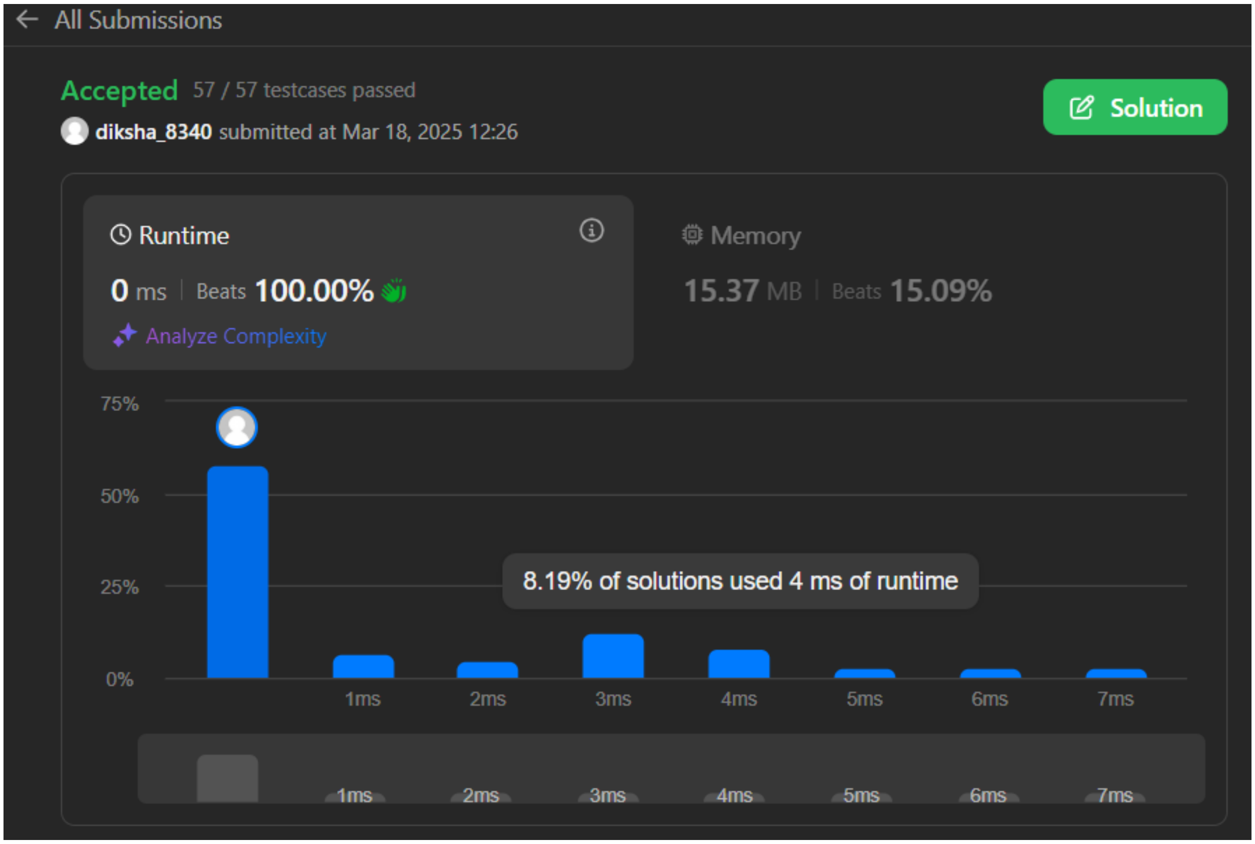
18.66 MB | Beats 67.35%

6. Super pow

CODE:

```
class Solution {
public:
  const int MOD = 1337;

  int powerMod(int a, int b) {
    int result = 1;
    a %= MOD;
    while (b > 0) {
      if (b % 2 == 1) {
        result = (result * a) % MOD;
      }
      a = (a * a) % MOD;
      b /= 2;
    }
    return result;
  }

  int superPow(int a, vector<int>& b) {
    int result = 1;
    for (int digit : b) {
      result = powerMod(result, 10) * powerMod(a, digit) % MOD;
    }
    return result;
  }
};
```

OUTPUT:



Accepted 57 / 57 testcases passed

diksha_8340 submitted at Mar 18, 2025 12:26

0 ms | Beats 100.00%

15.37 MB | Beats 15.09%

8.19% of solutions used 4 ms of runtime

7. Beautiful Array:

An array nums of length n is **beautiful** if:

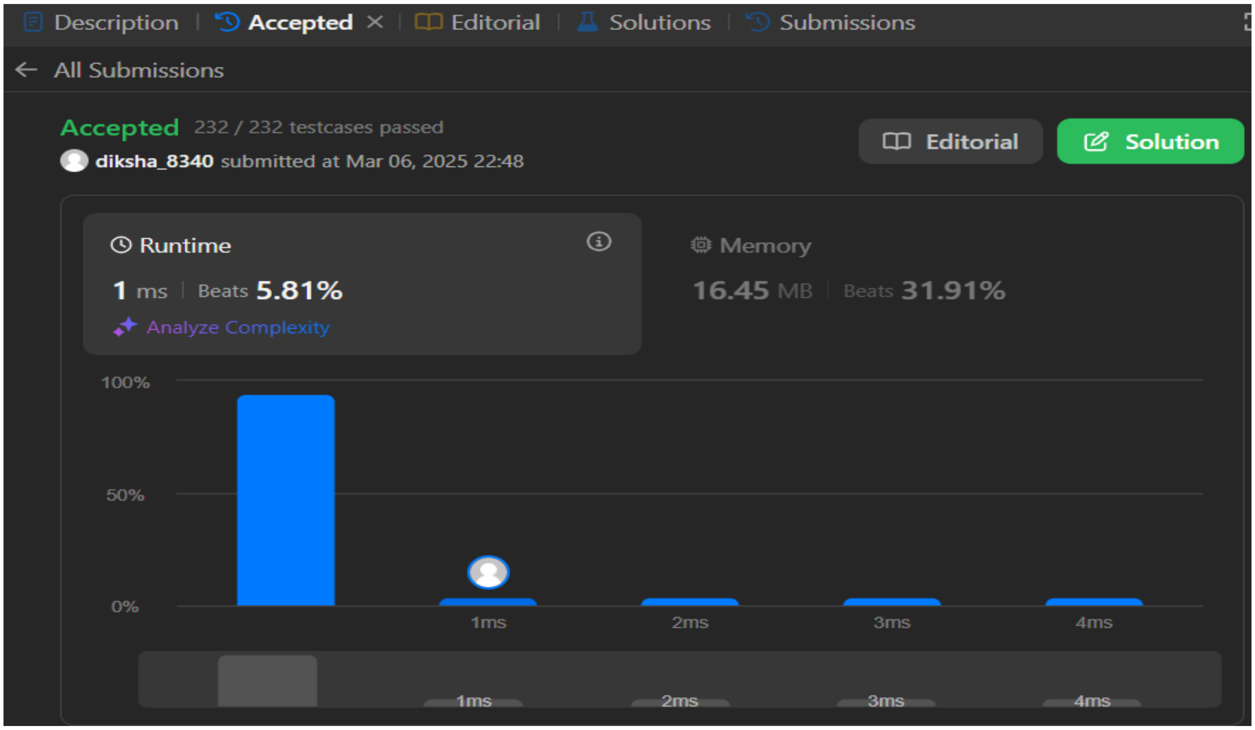
- nums is a permutation of the integers in the range [1, n].
- For every 0 <= i < j < n, there is no index k with i < k < j where 2 * nums[k] == nums[i] + nums[j].

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

CODE:

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

OUTPUT:



Accepted 232 / 232 testcases passed

diksha_8340 submitted at Mar 06, 2025 22:48

Runtime

1 ms | Beats 5.81%

Memory

16.45 MB | Beats 31.91%

8. The Skyline Problem

A city's **skyline** is the outer contour of the silhouette formed by all the buildings in that city when viewed from a distance. Given the locations and heights of all the buildings, return *the **skyline** formed by these buildings collectively*.

The geometric information of each building is given in the array buildings where buildings[i] = [left$_i$, right$_i$, height$_i$]:

- left$_i$ is the x coordinate of the left edge of the ith building.
- right$_i$ is the x coordinate of the right edge of the ith building.
- height$_i$ is the height of the ith building.

You may assume all buildings are perfect rectangles grounded on an absolutely flat surface at height 0.

The **skyline** should be represented as a list of "key points" **sorted by their x-coordinate** in the form [[x$_1$,y$_1$],[x$_2$,y$_2$],...]. Each key point is the left endpoint of some horizontal segment in the skyline except the last point in the list, which always has a y-coordinate 0 and is used to mark the skyline's termination where the rightmost building ends. Any ground between the leftmost and rightmost buildings should be part of the skyline's contour.

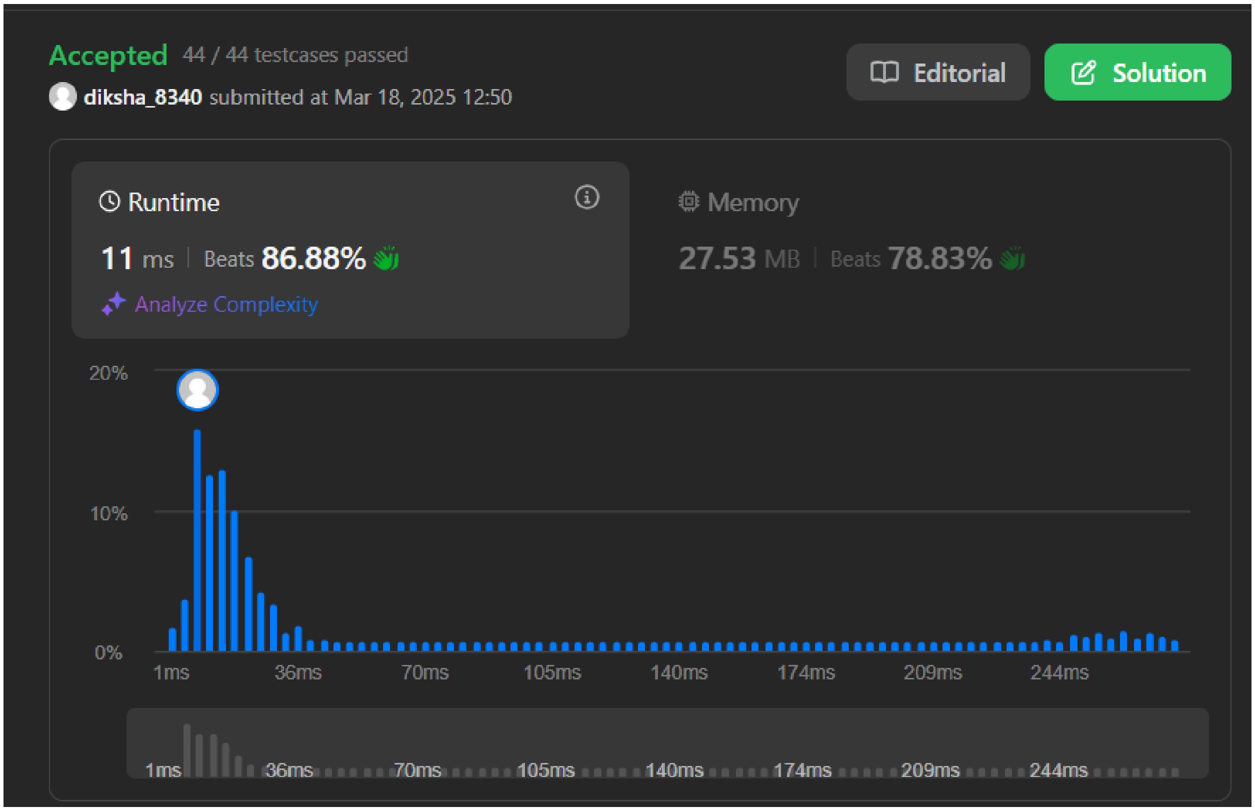
Note: There must be no consecutive horizontal lines of equal height in the output skyline. For instance, [...,[2 3],[4 5],[7 5],[11 5],[12 7],...] is not acceptable; the three lines of height 5 should be merged into one in the final output as such: [...,[2 3],[4 5],[12 7],...]

CODE:

```
class Solution {
public:
  vector<vector<int>> getSkyline(vector<vector<int>>& buildings) {
    vector<pair<int, int>> events;
    vector<vector<int>> result;
    multiset<int> heights = {0};


    for (auto &b : buildings) {
      events.push_back({b[0], -b[2]});
      events.push_back({b[1], b[2]});
    }

    sort(events.begin(), events.end(), [](const pair<int, int> &a, const pair<int, int> &b) {
      return a.first < b.first || (a.first == b.first && a.second < b.second);
    });

    int prevHeight = 0;
    for (auto &e : events) {
      if (e.second < 0) {
        heights.insert(-e.second);
      } else {
        heights.erase(heights.find(e.second));
      }
```

```
            int maxHeight = *heights.rbegin();
            if (maxHeight != prevHeight) {
                result.push_back({e.first, maxHeight});
                prevHeight = maxHeight;
            }
        }
        return result;
    }
};
```

OUTPUT:



Accepted 44 / 44 testcases passed

diksha_8340 submitted at Mar 18, 2025 12:50

Runtime

11 ms | Beats 86.88%

Analyze Complexity

Memory

27.53 MB | Beats 78.83%

9. Reverse Pairs:

Given an integer array nums, return *the number of reverse pairs in the array*.

A reverse pair is a pair (i, j) where:

<= i < j < nums.length and

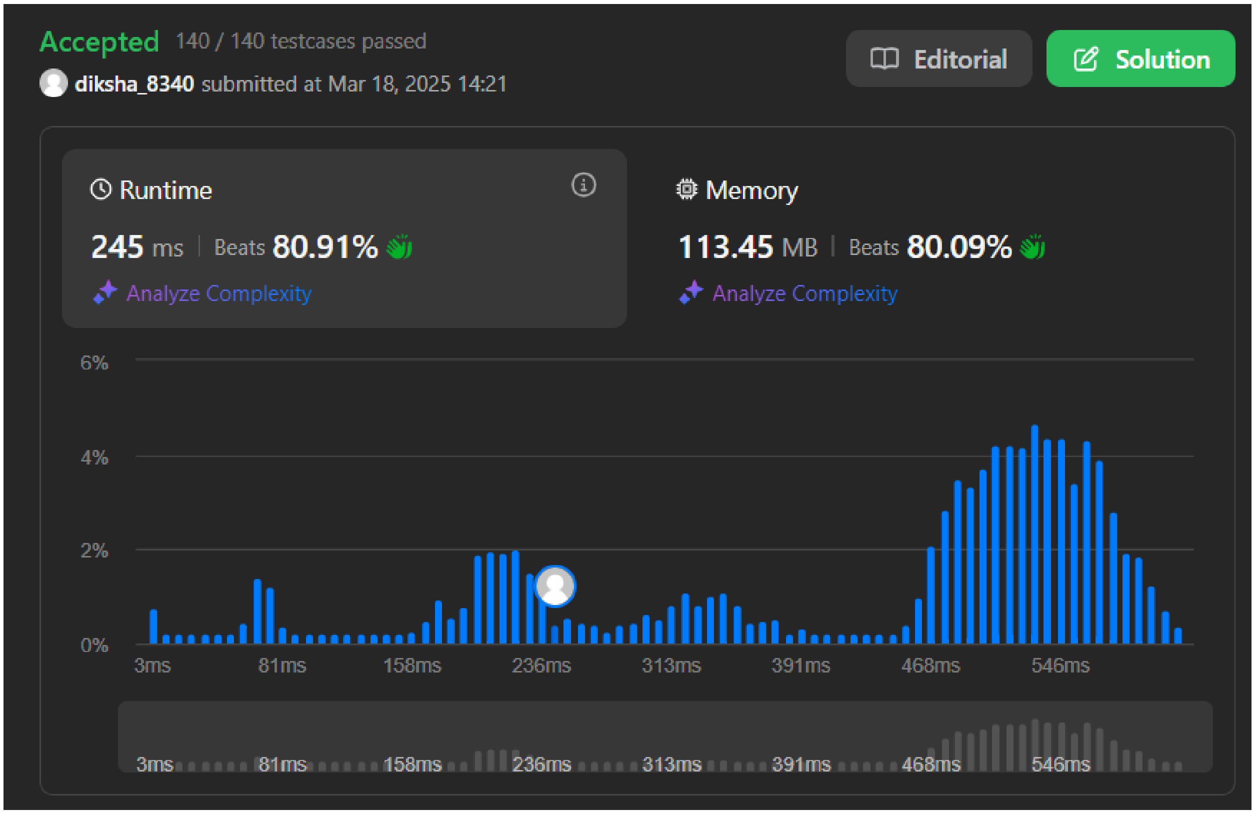
nums[i] > 2 * nums[j].

CODE:

```
class Solution {
public:
int mergeSort(vector<int>& nums, int start, int mid, int end){
    int count = 0;
    int j = mid + 1;

  // Count the reverse pairs
  for (int i = start; i <= mid; i++) {
    while (j <= end && nums[i] > 2LL * nums[j]) {
      j++;
    }
    count += (j - mid - 1);
  }

    vector<int> temp(end - start + 1);
    int left = start, right = mid + 1, idx = 0;

    while (left <= mid && right <= end) {
      if (nums[left] <= nums[right]) {
        temp[idx++] = nums[left++];
      } else {
        temp[idx++] = nums[right++];
      }
    }
    while (left <= mid) temp[idx++] = nums[left++];
    while (right <= end) temp[idx++] = nums[right++];

    // Copy back the sorted elements
    for (int i = start; i <= end; i++) {
      nums[i] = temp[i - start];
    }

    return count;
  }

  int mergeSortAndCount(vector<int>& nums, int start, int end) {
  if (start >= end) return 0;

  int mid = start + (end - start) / 2;
```

```
    int count = 0;

    count += mergeSortAndCount(nums, start, mid);
    count += mergeSortAndCount(nums, mid + 1, end);
    count += mergeSort(nums, start, mid, end);

    return count;
}
    int reversePairs(vector<int>& nums) {
        return mergeSortAndCount(nums, 0, nums.size()-1);
    }
};
```

OUTPUT:



Accepted 140 / 140 testcases passed

diksha_8340 submitted at Mar 18, 2025 14:21

Runtime

245 ms | Beats 80.91%

Memory

113.45 MB | Beats 80.09%

10. Longest Increasing Subsequence II

You are given an integer array nums and an integer k.

Find the longest subsequence of nums that meets the following requirements:

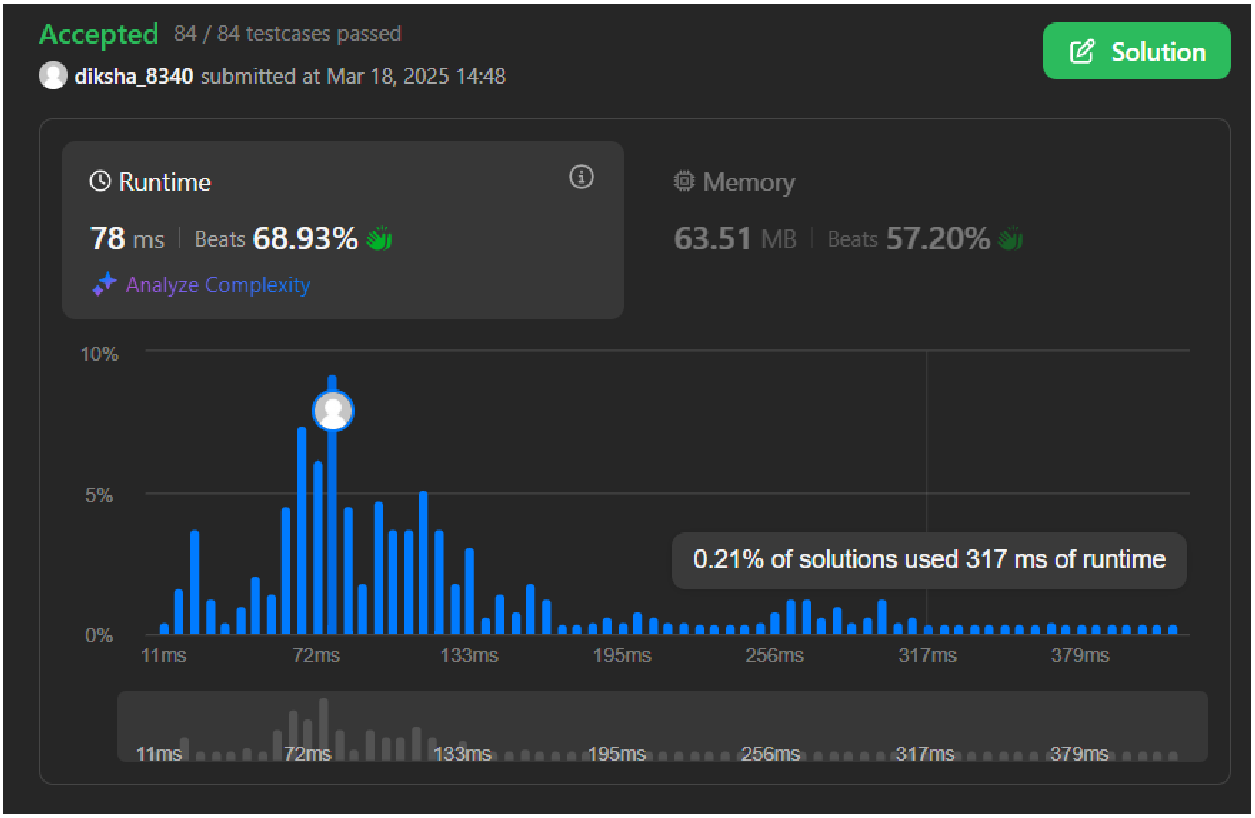
- The subsequence is **strictly increasing** and
- The difference between adjacent elements in the subsequence is **at most** k.

Return *the length of the **longest subsequence** that meets the requirements.*
A **subsequence** is an array that can be derived from another array by deleting some or no elements without changing the order of the remaining elements.

CODE

```
#include <algorithm>
class SegmentTree {
    vector<int> tree;
    int size;

public:
    SegmentTree(int n) {
        size = n;
        tree.resize(4 * n, 0);
    }

    int query(int node, int start, int end, int left, int right) {
        if (left > end || right < start) return 0; // Out of range
        if (left <= start && end <= right) return tree[node]; // Fully within range
        int mid = (start + end) / 2;
        return max(query(2 * node, start, mid, left, right),
                   query(2 * node + 1, mid + 1, end, left, right));
    }

    void update(int node, int start, int end, int index, int value) {
        if (start == end) {
            tree[node] = value;
            return;
        }
        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 maxLen = 0;

    for (int num : nums) {
      int bestPrev = segTree.query(1, 0, maxVal, max(0, num - k), num - 1);
      int currLen = bestPrev + 1;
      segTree.update(1, 0, maxVal, num, currLen);
      maxLen = max(maxLen, currLen);
    }

    return maxLen;
  }
};
```

OUTPUT



Accepted 84 / 84 testcases passed

diksha_8340 submitted at Mar 18, 2025 14:48

Runtime

78 ms | Beats 68.93%

Memory

63.51 MB | Beats 57.20%

0.21% of solutions used 317 ms of runtime