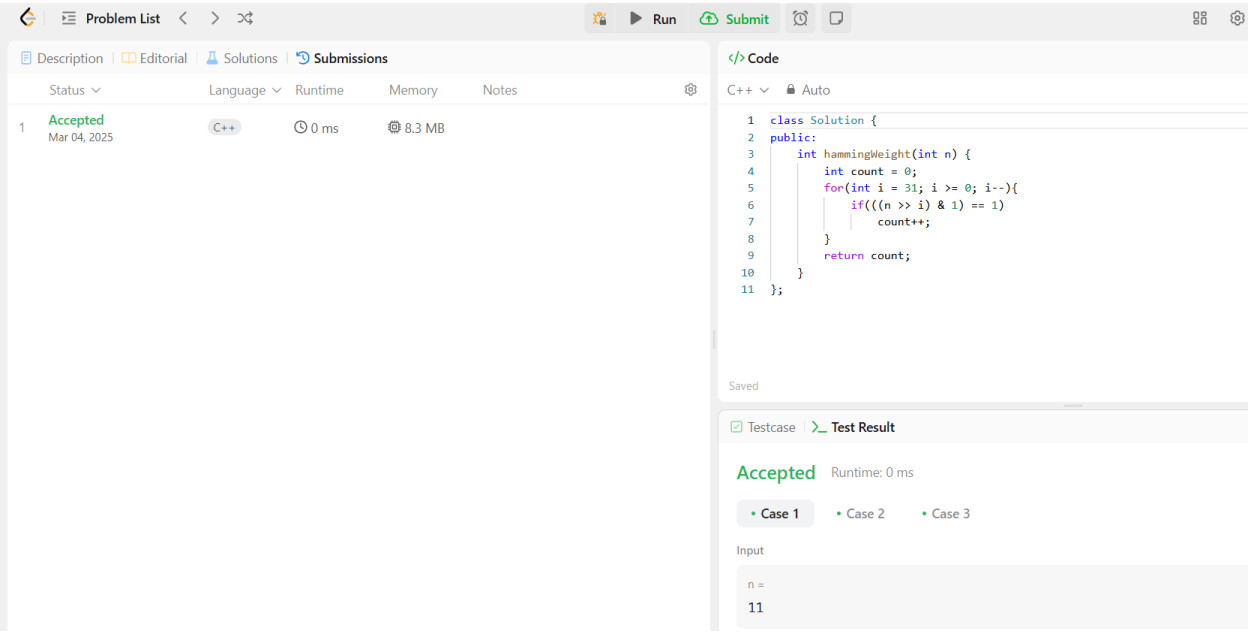
AP Assignment Abhishek Sharma

(22BCS17260)

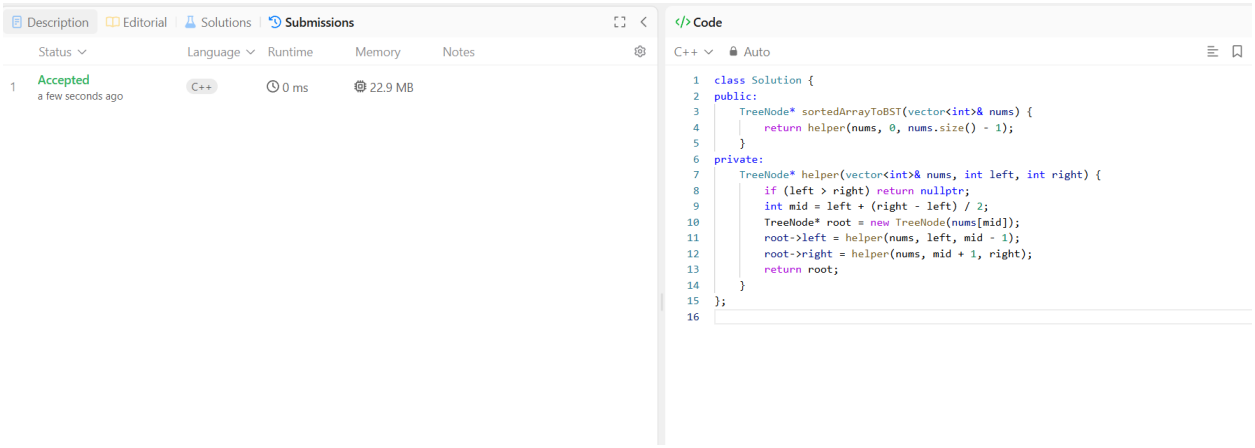
191.Number of 1 Bits

```
class Solution {
public:
    int hammingWeight(int n) {
        int count = 0;
        for(int i = 31; i >= 0; i--){
            if(((n >> i) & 1) == 1)
                count++;
        }
        return count;
    }
};
```



108.Convert Sorted Array to Binary Search Tree

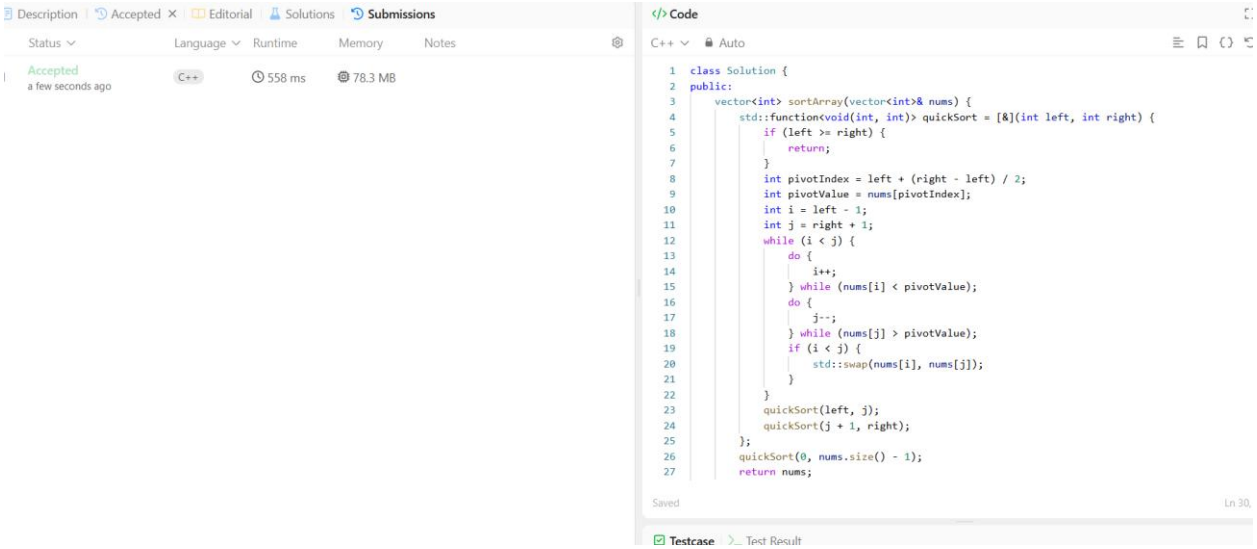
```
#include <vector>
using namespace std;


class Solution {
public:
    TreeNode* sortedArrayToBST(vector<int>& nums) {
        return helper(nums, 0, nums.size() - 1);
    }
private:
    TreeNode* helper(vector<int>& nums, int left, int right) {
        if (left > right) return nullptr;
        int mid = left + (right - left) / 2;
        TreeNode* root = new TreeNode(nums[mid]);
        root->left = helper(nums, left, mid - 1);
        root->right = helper(nums, mid + 1, right);
        return root;
    }
};
```



912.Sort an Array

```
#include <vector>
#include <functional>
class Solution {
public:
  vector<int> sortArray(vector<int>& nums) {
    std::function<void(int, int)> quickSort = [&](int left, int right) {
      if (left >= right) {
        return;
      }
      int pivotIndex = left + (right - left) / 2;
      int pivotValue = nums[pivotIndex];
      int i = left - 1;
      int j = right + 1;
      while (i < j) {
        do {
          i++;
        } while (nums[i] < pivotValue);
        do {
          j--;
        } while (nums[j] > pivotValue);
        if (i < j) {
          std::swap(nums[i], nums[j]);
        }
      }
```

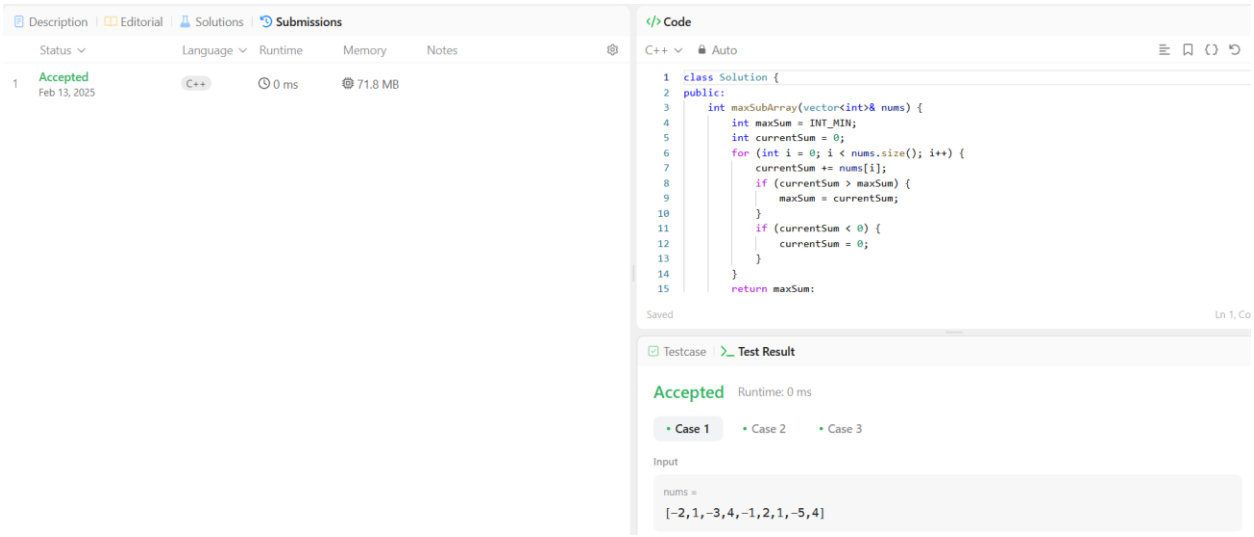
```
        quickSort(left, j);
        quickSort(j + 1, right);
    };
    quickSort(0, nums.size() - 1);
    return nums;
  }
};
```



53.[Maximum Subarray]

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

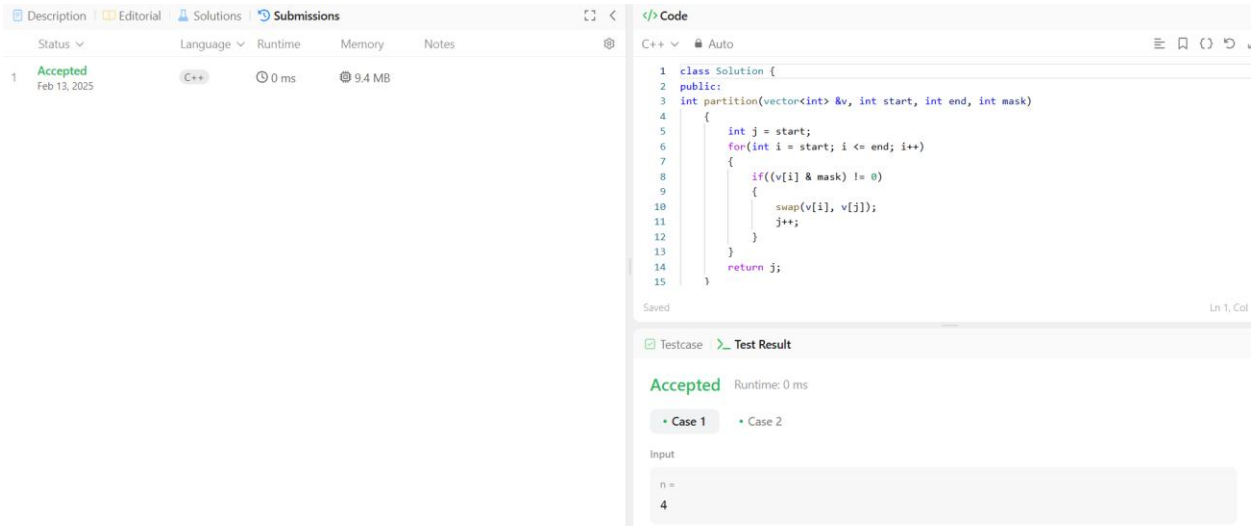
```
        }
        if (currentSum < 0) {
            currentSum = 0;
        }
    }
    return maxSum;
  }
};
```



932.[Beautiful Array]

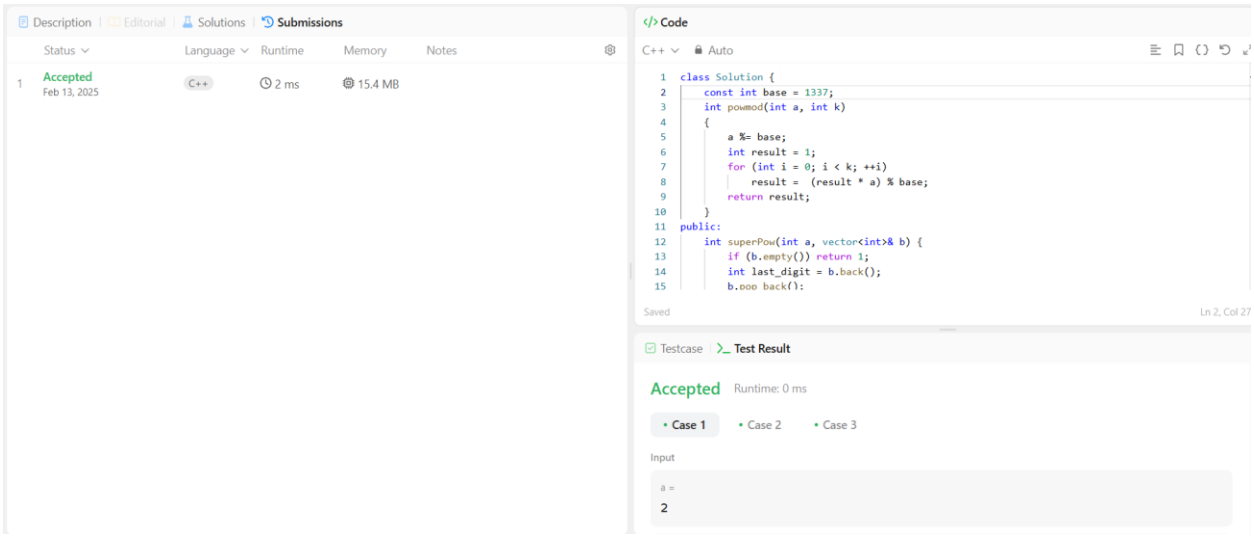
```
class Solution {
public:
int partition(vector<int> &v, int start, int end, int mask)
  {
    int j = start;
    for(int i = start; i <= end; i++)
    {
      if((v[i] & mask) != 0)
      {
        swap(v[i], v[j]);
```

```
            j++;
          }
        }
        return j;
    }

    void sort(vector<int> & v, int start, int end, int mask)
    {
        if(start >= end) return;
        int mid = partition(v, start, end, mask);
        sort(v, start, mid - 1, mask << 1);
        sort(v, mid, end, mask << 1);
    }
    vector<int> beautifulArray(int n) {
        vector<int> ans;
        for(int i = 0; i < n; i++) ans.push_back(i + 1);
        sort(ans, 0, n - 1, 1);
        return ans;

    }
};
```



372.[Super Pow]

```
class Solution {
   const int base = 1337;
   int powmod(int a, int k)
   {
      a %= base;
      int result = 1;
      for (int i = 0; i < k; ++i)
         result =  (result * a) % base;
      return result;
   }
public:
   int superPow(int a, vector<int>& b) {
      if (b.empty()) return 1;
      int last_digit = b.back();
      b.pop_back();
      return powmod(superPow(a, b), 10) * powmod(a, last_digit) % base;
   }
};
```



218.[The Skyline Problem]

```
class Solution {
public:
  vector<vector<int>> getSkyline(vector<vector<int>>& buildings) {
    vector<vector<int>> ans;
    multiset<int> pq{0};

    vector<pair<int, int>> points;

    for(auto b: buildings){
      points.push_back({b[0], -b[2]});
      points.push_back({b[1], b[2]});
    }
    sort(points.begin(), points.end());

    int ongoingHeight = 0;
    for(int i = 0; i < points.size(); i++){
      int currentPoint = points[i].first;
      int heightAtCurrentPoint = points[i].second;
```

```
        if(heightAtCurrentPoint < 0){
            pq.insert(-heightAtCurrentPoint);
        } else {
            pq.erase(pq.find(heightAtCurrentPoint));
        }
        auto pqTop = *pq.rbegin();
        if(ongoingHeight != pqTop){
            ongoingHeight = pqTop;
            ans.push_back({currentPoint, ongoingHeight});
        }
    }
    return ans;
  }
};
```